

CC, Spring 2014
Exam Project, part 4

| | |
|-------|---|
| Group | 4 |
|-------|---|

| | |
|------|----------------|
| Date | March 12, 2018 |
|------|----------------|

| | |
|-----------|----------------------------------|
| Name | Andreas Aagaard Lynge |
| Birthday | 17-12-1987 |
| Logins | anlyn11 - anlyn11@student.sdu.dk |
| Signature | |

| | |
|-----------|----------------------------------|
| Name | Martin Møller Andersen |
| Birthday | 26-1-1992 |
| Logins | maan511 - maan511@student.sdu.dk |
| Signature | |

This report contains a total of 661 pages.

Contents

| | | |
|----------|--|-----------|
| 1 | Introduction | 8 |
| 2 | The Vitaly Programming Language | 8 |
| 2.1 | Basic Vitaly Programming | 8 |
| 2.1.1 | Hello World | 8 |
| 2.1.2 | Primitive Data Types | 9 |
| 2.1.3 | Expressions | 11 |
| 2.1.4 | Control Flow | 11 |
| 2.1.5 | Arrays | 12 |
| 2.1.6 | Type Cast | 14 |
| 2.1.7 | Records | 15 |
| 2.1.8 | Type Definitions | 16 |
| 2.1.9 | Functions | 18 |
| 2.1.10 | Overloading functions | 21 |
| 2.1.11 | Scope | 22 |
| 2.1.12 | Import | 24 |
| 2.1.13 | Organizing a Vitaly Project | 28 |
| 2.2 | Advanced Vitaly Programming | 29 |
| 2.2.1 | Finalize | 29 |
| 2.2.2 | Inheritance | 30 |
| 2.2.3 | Structural Equivalence | 33 |
| 2.2.4 | Functions In Records | 36 |
| 2.2.5 | Record Constructors | 38 |
| 2.2.6 | Records Destructors | 42 |
| 2.2.7 | Interfacing with C | 43 |
| 2.3 | Vitaly Standard Library | 44 |
| 2.3.1 | std.c | 44 |
| 2.3.2 | std.object | 44 |
| 2.3.3 | std.string | 44 |
| 2.3.4 | std.errno | 45 |
| 2.3.5 | std.stdio | 46 |
| 2.3.6 | Standard out - stdo | 46 |
| 2.3.7 | Standard in - stdi | 47 |
| 2.3.8 | std.math | 47 |
| 2.3.9 | std.indexable | 47 |
| 2.3.10 | std.array | 48 |
| 2.3.11 | std.vector | 48 |
| 2.3.12 | std.comparator | 48 |
| 2.3.13 | std.sort | 49 |
| 2.3.14 | std.thread | 50 |
| 2.3.15 | std.lock | 52 |
| 3 | Compiler flags | 53 |
| 3.1 | General flags | 54 |
| 3.2 | Optimize flags | 55 |
| 3.3 | Warning flags | 55 |
| 3.4 | Dump flags | 56 |

| | | |
|-----------|--|------------|
| 4 | Compiler Library | 57 |
| 4.1 | String | 57 |
| 4.2 | Linked Lists | 58 |
| 4.3 | Vector | 59 |
| 4.4 | Dynamic Hash Map | 59 |
| 4.5 | Red Black Tree | 60 |
| 4.6 | Debugging Facilities | 60 |
| 5 | The Vitaly Grammar | 61 |
| 6 | Parsing | 65 |
| 6.1 | Scanning | 65 |
| 6.2 | Parsing | 66 |
| 6.3 | Abstract Syntax Tree | 70 |
| 6.4 | Error Recovery | 72 |
| 7 | Symbol Table | 73 |
| 7.1 | Name Mangling | 75 |
| 8 | Imports | 76 |
| 8.1 | The Merge Table | 78 |
| 8.2 | Nested imports | 80 |
| 8.3 | Import collisions | 80 |
| 8.4 | Viti, Vitaly Interface Files | 80 |
| 8.5 | Recursive compilation | 81 |
| 9 | Type Checking | 82 |
| 9.1 | Structural Equivalence | 82 |
| 9.2 | Overloaded Function Selection | 83 |
| 10 | Intermediate Code Generation | 84 |
| 10.1 | Nesting Functions | 86 |
| 10.2 | Records and Multiple Inheritance | 88 |
| 11 | Optimization | 92 |
| 11.1 | Constant Propagation | 93 |
| 11.2 | Instruction Elimination | 95 |
| 11.3 | Register Variables | 95 |
| 12 | Register allocation | 96 |
| 13 | Unused Move Elimination | 98 |
| 14 | Testing | 99 |
| A | Source Code | 102 |
| A.1 | src/main.c | 102 |
| A.2 | src/main.h | 122 |
| A.3 | Scanning/Parsing | 123 |
| A.3.1 | src/ast/ast.c | 123 |
| A.3.2 | src/ast/ast.h | 126 |

| | | |
|--------|--|-----|
| A.3.3 | src/ast/ast_string.c | 135 |
| A.3.4 | src/ast/ast_string.h | 137 |
| A.3.5 | src/ast/ast_visitor.h | 137 |
| A.3.6 | src/ast/ast_visitor_print.c | 140 |
| A.3.7 | src/ast/ast_visitor_print.h | 148 |
| A.3.8 | src/ast/ast_visitor_print_graph.c | 148 |
| A.3.9 | src/ast/ast_visitor_print_graph.h | 155 |
| A.3.10 | src/ast/ast_visitor_symbol_table.c | 155 |
| A.3.11 | src/ast/ast_visitor_symbol_table.h | 167 |
| A.3.12 | src/ast/symbol_table.c | 167 |
| A.3.13 | src/ast/symbol_table.h | 217 |
| A.3.14 | src/parser.c | 228 |
| A.3.15 | src/parser.h | 229 |
| A.3.16 | src/parser/parser.y | 229 |
| A.3.17 | src/parser/scanner.l | 252 |
| A.4 | Imports | 255 |
| A.4.1 | src/ast/ast_visitor_import.c | 255 |
| A.4.2 | src/ast/ast_visitor_import.h | 263 |
| A.4.3 | src/import_handler.c | 263 |
| A.4.4 | src/import_handler.h | 279 |
| A.5 | Type Checking | 280 |
| A.5.1 | src/ast/ast_visitor_type_check.c | 280 |
| A.5.2 | src/ast/ast_visitor_type_check.h | 319 |
| A.6 | Generate AIA | 320 |
| A.6.1 | src/ast/ast_visitor_aia.c | 320 |
| A.6.2 | src/ast/ast_visitor_aia.h | 378 |
| A.6.3 | src/ast/ast_visitor_delete.c | 378 |
| A.6.4 | src/ast/ast_visitor_delete.h | 383 |
| A.6.5 | src/ast/ast_visitor_dependency.c | 383 |
| A.6.6 | src/ast/ast_visitor_dependency.h | 392 |
| A.7 | Intermediate Representation | 393 |
| A.7.1 | src/aia/aia.c | 393 |
| A.7.2 | src/aia/aia.h | 404 |
| A.7.3 | src/aia/aia_functions_return.c | 417 |
| A.7.4 | src/aia/aia_functions_return.h | 418 |
| A.7.5 | src/aia/aia_instr.c | 419 |
| A.7.6 | src/aia/aia_instr.h | 424 |
| A.7.7 | src/aia/aia_normalize_addr.c | 431 |
| A.7.8 | src/aia/aia_normalize_addr.h | 434 |
| A.7.9 | src/aia/aia_operand.c | 434 |
| A.7.10 | src/aia/aia_operand.h | 439 |
| A.7.11 | src/aia/aia_operand_map.c | 443 |
| A.7.12 | src/aia/aia_operand_map.h | 444 |
| A.7.13 | src/aia/aia_operand_set.c | 445 |
| A.7.14 | src/aia/aia_operand_set.h | 446 |
| A.7.15 | src/aia/aia_warn_undefined.c | 447 |
| A.7.16 | src/aia/aia_warn_undefined.h | 448 |
| A.8 | Optimization | 448 |
| A.8.1 | src/aia/aia_block_elim.c | 448 |
| A.8.2 | src/aia/aia_block_elim.h | 450 |

| | | |
|---------|---|-----|
| A.8.3 | src/aia/aia_const_prop.c | 450 |
| A.8.4 | src/aia/aia_const_prop.h | 460 |
| A.8.5 | src/aia/aia_def_to_use.c | 461 |
| A.8.6 | src/aia/aia_def_to_use.h | 465 |
| A.8.7 | src/aia/aia_func_access.c | 465 |
| A.8.8 | src/aia/aia_func_access.h | 470 |
| A.8.9 | src/aia/aia_func_kills.c | 471 |
| A.8.10 | src/aia/aia_func_kills.h | 475 |
| A.8.11 | src/aia/aia_instr_elim.c | 476 |
| A.8.12 | src/aia/aia_instr_elim.h | 479 |
| A.8.13 | src/aia/aia_instr_live_sets.c | 479 |
| A.8.14 | src/aia/aia_instr_live_sets.h | 488 |
| A.8.15 | src/aia/aia_optimize.c | 489 |
| A.8.16 | src/aia/aia_optimize.h | 490 |
| A.8.17 | src/aia/aia_unused_set.c | 490 |
| A.8.18 | src/aia/aia_unused_set.h | 494 |
| A.8.19 | src/x86_32/x86_32_regs.c | 494 |
| A.8.20 | src/x86_32/x86_32_regs.h | 503 |
| A.8.21 | src/x86_32/x86_32_reg_alloc.h | 505 |
| A.8.22 | src/x86_32/x86_32_reg_alloc_color.c | 505 |
| A.8.23 | src/x86_32/x86_32_reg_alloc_color.h | 531 |
| A.8.24 | src/x86_32/x86_32_reg_vars.c | 531 |
| A.8.25 | src/x86_32/x86_32_reg_vars.h | 541 |
| A.9 | Code Generation | 541 |
| A.9.1 | src/vit/end.s | 541 |
| A.9.2 | src/vit/ini.s | 542 |
| A.9.3 | src/vit/lib.c | 542 |
| A.9.4 | src/vit/lib.s | 543 |
| A.9.5 | src/vit/retmain.s | 546 |
| A.9.6 | src/vit/vitmain.s | 547 |
| A.9.7 | src/x86_32/x86_32.c | 547 |
| A.9.8 | src/x86_32/x86_32.h | 548 |
| A.9.9 | src/x86_32/x86_32_emit.c | 548 |
| A.9.10 | src/x86_32/x86_32_emit.h | 559 |
| A.9.11 | src/x86_32/x86_32_func_normalize.c | 559 |
| A.9.12 | src/x86_32/x86_32_func_normalize.h | 569 |
| A.9.13 | src/x86_32/x86_32_normalize.c | 569 |
| A.9.14 | src/x86_32/x86_32_normalize.h | 577 |
| A.10 | Compiler Library | 578 |
| A.10.1 | src/alloc.c | 578 |
| A.10.2 | src/alloc.h | 578 |
| A.10.3 | src/bit_vector.h | 578 |
| A.10.4 | src/debug.c | 579 |
| A.10.5 | src/debug.h | 581 |
| A.10.6 | src/dot_printer.c | 582 |
| A.10.7 | src/dot_printer.h | 584 |
| A.10.8 | src/double_list.c | 586 |
| A.10.9 | src/double_list.h | 586 |
| A.10.10 | src/file_location.c | 588 |
| A.10.11 | src/file_location.h | 588 |

| | |
|---|-----|
| A.10.12src/hash_map.c | 589 |
| A.10.13src/hash_map.h | 591 |
| A.10.14src/help_msg.c | 594 |
| A.10.15src/help_msg.h | 597 |
| A.10.16src/io.c | 597 |
| A.10.17src/io.h | 600 |
| A.10.18src/pointer_hash.c | 601 |
| A.10.19src/pointer_hash.h | 601 |
| A.10.20src/rb_tree.c | 601 |
| A.10.21src/rb_tree.h | 606 |
| A.10.22src/report.c | 608 |
| A.10.23src/report.h | 612 |
| A.10.24src/single_list.c | 613 |
| A.10.25src/single_list.h | 613 |
| A.10.26src/std_define.h | 614 |
| A.10.27src/std_include.h | 615 |
| A.10.28src/str.c | 616 |
| A.10.29src/str.h | 619 |
| A.10.30src/string_builder.c | 626 |
| A.10.31src/string_builder.h | 626 |
| A.10.32src/test/test_hash_map.c | 627 |
| A.10.33src/test/test_include.h | 630 |
| A.10.34src/test/test_lists.c | 630 |
| A.10.35src/test/test_rb_tree.c | 632 |
| A.10.36src/timer.c | 634 |
| A.10.37src/timer.h | 634 |
| A.10.38src/vector.c | 635 |
| A.10.39src/vector.h | 635 |
| A.11 Makefile System | 639 |
| A.11.1 src/aia/Makefile | 639 |
| A.11.2 src/ast/Makefile | 639 |
| A.11.3 src/Makefile | 639 |
| A.11.4 src/parser/gen/Makefile | 640 |
| A.11.5 src/parser/Makefile | 640 |
| A.11.6 src/test/Makefile | 641 |
| A.11.7 src/vit/Makefile | 641 |
| A.11.8 src/vit/std/c/Makefile | 642 |
| A.11.9 src/vit/std/Makefile | 642 |
| A.11.10src/vit/_vit_thread/Makefile | 643 |
| A.11.11src/vit/_vit_thread/std/Makefile | 644 |
| A.11.12src/x86_32/Makefile | 645 |
| A.12 Vitaly Standard Library | 645 |
| A.12.1 src/vit/std/array.vit | 645 |
| A.12.2 src/vit/std/cerrno.c | 646 |
| A.12.3 src/vit/std/c/stdio.c | 646 |
| A.12.4 src/vit/std/c/ctype.vit | 647 |
| A.12.5 src/vit/std/c/stdio.vit | 647 |
| A.12.6 src/vit/std/c/string.vit | 647 |
| A.12.7 src/vit/std/comparator.vit | 647 |
| A.12.8 src/vit/std/errno.vit | 648 |

| | |
|---|-----|
| A.12.9 src/vit/std/indexable.vit | 650 |
| A.12.10src/vit/std/math.vit | 650 |
| A.12.11src/vit/std/object.vit | 650 |
| A.12.12src/vit/std/sort.vit | 651 |
| A.12.13src/vit/std/stdio.vit | 651 |
| A.12.14src/vit/std/stdlib.vit | 653 |
| A.12.15src/vit/std/string.vit | 653 |
| A.12.16src/vit/std/vector.vit | 657 |
| A.12.17src/vit/_vit_thread/std/clock.c | 658 |
| A.12.18src/vit/_vit_thread/std/cthread.c | 659 |
| A.12.19src/vit/_vit_thread/std/lock.vit | 659 |
| A.12.20src/vit/_vit_thread/std/thread.vit | 660 |

1 Introduction

This paper describes the implementation of a compiler for the vitality programming language, which is largely extended with object oriented programming capabilities, a standard library including support for multi threading, among other extensions. We begin by an introduction to the extended vitality programming language, where all features of the language will be discussed along with examples. Afterwards the implementation of the compiler will be discussed in the following sections.

2 The Vitaly Programming Language

2.1 Basic Vitaly Programming

2.1.1 Hello World

By tradition the "Hello world" program is used when introducing a new programming language. Here is the vitality version:

```
1 write "Hello world";
```

Saving that in a text file called `hw.vit` and compiling with

```
1 vitality hw.vit
```

will generate an executable file `a.out` which prints

```
1 Hello world
```

when it's run.

Alternatively you might want to compile with

```
1 vitality -o hello-world hw.vit
```

which will generate an executable called `hello-world` instead of the default `a.out`.

The global scope of a vitality program is treated somewhat different from what we are used to from languages like Java and C.

In vitality, statements and expressions are allowed in the global scope outside of functions. When a vitality program is executed, the code in the global scope is executed first. If we define a `main` function, and compile the vitality program with the `-m` option enabled, the `main` function gets called after the code in the global scope is executed.

We will discuss the `main` function, and functions in general in a later subsection.

As in most other programming languages, it is possible to insert comments which are ignored by the compiler. Line comments begin with `#` and block comments begin with `(*` and ends with `*)`. E.g.:

```
1 # This is a line comment
2 (* This is a
3    multi line
4    comment *)
```

2.1.2 Primitive Data Types

The vitality programming language knows about the following primitive data types:

- `string`, string literals like `"Hello world"`.
- `char`, character literals like `'a'` and `'C'`.
- `int`, integers like `42` and `-1`.
- `bool`, the boolean values `true` or `false`.

For example:

```
1 var s:string = "my string"; # declare and initialize the string s.
2 var c:char = 'T';           # declare and initialize the char c.
3 var i:int = 300;            # declare and initialize the int i.
4 var b:bool = true;          # declare and initialize the bool b.
5 write s;
6 write c;
7 write i;
8 write b;
```

will output:

```
1 my string
2 T
3 300
4 true
```

Identifiers consist of the characters a-z, A-Z, 0-9, and underscore (`_`). Variable names, and identifiers in general, must begin with one of the characters a-z, A-Z or underscore (`_`), however.

Also, identifiers in vitality are case sensitive. So, for example, the identifier `count` is different from the identifier `Count`.

We might be tempted to try to initialize a `string` with a `char`, initialize an `int` with a `bool` or maybe initialize a `char` with an `int`. E.g.:

```
1 var myString:string = 'S';
2 var myInt:int = true;
3 var myChar:char = 20;
```

Saving this in a file called `test.vit` and compiling with:

```
1 vitality test.vit
```

produces the following output:

```
1 test.vit:1:21: (error) incompatible assignment of 'string' type from 'char' type
2 test.vit:2:15: (warning) implicit cast from 'bool' to 'int'
3 vitality: 1 error, 1 warning
```

The first line of the output is an error and says that a `string` is not assignment compatible with a `char` value.

The second line is "only" a warning, it says that the `bool` value is implicitly casted to an `int`. This might come as a surprise to a Java programmer, when casted to an integer the `bool` value `true` is 1 and `false` is 0.

The other way around is also possible, when an `int` different from 0 is casted to `bool` the result is `true` otherwise the result is `false`.

2.1 Basic Vitaly Programming 2 THE VITALY PROGRAMMING LANGUAGE

The third line of source code is not an error nor a warning. The vitality programming language implicitly casts back and forth between `char` and `int` values.

A `char` variable is a 1 byte ASCII code, however when used in arithmetic `char` is implicitly casted to `int`.

For example:

```
1 var o:char = '1';
2 var t:char = '2';
3 var a:char = o + t;
```

Will cast `o` to an `int`, cast `t` to an `int` and afterwards add the numbers 48 (ASCII code for `'1'`) and 49 (ASCII code for `'2'`) producing the integer result 97. In the end the `int` result is casted to the `char` `'a'`, since 97 is the ASCII code for `'a'`, and saved in the variable `a`.

We may initialize a `string` with the special value `null`, which is used to indicate that the `string` doesn't contain anything. E.g.:

```
1 var s:string = null;
```

We are going to talk more about `null` in later sections.

Vitaly supports 4 special escape sequences we can use when initializing `char` or `string` variables:

```
1 var str:string = "\'hello\\'\\n\\t\\'world\\'";
2 write str;
```

Outputs:

```
1 'hello'
2 "world"
```

The special sequence `\'` is replaced with a single `'`, the sequence `\n` is replaced with a new line character, `\t` is replaced by a tab character and `\"` is replaced with a `"`. We use the `\"` notation inside `string` literals when we want a `"` inside the literal instead of terminating the string literal. When we want a `'` inside a `char` literal we can use the `\'` escape sequence:

```
1 var c:char = '\';
2 write c;
```

output:

```
1 '
```

Notice that the `var` keyword is optional, and we don't have to initialize variables right away. E.g.:

```
1 a:int;
2 a = 10;
```

is equivalent to:

```
1 var a:int = 10;
```

and:

```
1 a:int = 10;
```

2.1.3 Expressions

Vitaly provides a number of binary and unary operators. The following lists the operators from lowest precedence to highest, with the operators in the same box having the same precedence.

The comment to the left shows which type the operator expects its operands to have and which result type (`->`) the operator has (in the following example `type` means any type):

```
1 expr1 || expr2; # bool || bool -> bool
```

```
1 expr1 && expr2; # bool && bool -> bool
```

```
1 expr1 == expr2; # type == type -> bool
2 expr1 != expr2; # type != type -> bool
3 expr1 > expr2; # int > int -> bool
4 expr1 < expr2; # int < int -> bool
5 expr1 >= expr2; # int <= int -> bool
6 expr1 <= expr2; # int <= int -> bool
```

```
1 expr1 + expr2; # int + int -> int
2 expr1 - expr2; # int - int -> int
```

```
1 expr1 * expr2; # int * int -> int
2 expr1 / expr2; # int / int -> int
```

```
1 !expr; # ! bool -> bool
```

```
1 |expr|; # | int | -> int
2 (expr); # ( type ) -> type
```

In most cases it is possible to feed the operator with a different type than it expects. However doing so will often produce an implicit cast warning.

It is possible to disable the implicit cast warnings with the `-w ign-implicit-cast` command line option, however doing so is not recommended. In a later subsection we will discuss another way to get rid of the implicit cast warnings.

Most of the operators supported by vitaly are equivalent to the operators found in the C-like languages. The only real surprise might be the `|expr|` operator which takes the absolute value of its `int` operand. The `||` operator is also used for another purpose, but we will talk about that in a later subsection.

2.1.4 Control Flow

The vitaly programming language also supports the `if` statement:

```
1 if cond then {
2     write "cond is true";
3 }
```

2.1 Basic Vitaly Programming 2 THE VITALY PROGRAMMING LANGUAGE

Note that when there is only one statement inside the `if` body you may omit the opening `{` and closing `}`. So the following is equivalent:

```
1 if cond then
2   write "cond is true";
```

The `if-else` statement:

```
1 if cond then
2   write "cond is true";
3 else
4   write "cond is false";
```

If `cond` is not a `bool` variable the vitality compiler will by default output a warning about an implicit cast to `bool`.

`cond` does not have to be a variable it may also be an expression. For example:

```
1 if i + j < 10 then {
2   write i;
3   write j;
4 }
```

Notice that the `<` operator produces a `bool` result.

The vitality programming language also supports the `while` statement:

```
1 i:int = 0;
2 while i < 3 do {
3   write i;
4   i = i + 1;
5 }
```

Instead of `i < 3` the condition may be any valid expression.

2.1.5 Arrays

Any type you can use in vitality you can put inside an `array`. For example:

```
1 a:array of int;
2 b:array of bool;
3 c:array of string;
4 d:array of char;
5 e:array of array of int;
6 f:array of array of array of string;
7 # You got the point ...
```

The first line declares an `array` containing `int` values, the second line an `array` with `bool` values, and so on, until line 5 and 6 which declares arrays containing arrays.

The array differs from the primitive types in that we need to allocate memory for the array before we can use it. Example follows:

```
1 ary:array of int;
2 allocate ary of length 3;
3 i:int = 0;
4 while i < 3 do {
5   ary[i] = i + 10;
6   i = i + 1;
7 }
8 while i > 0 do {
9   i = i - 1;
10  write ary[i];
11 }
```

2.1 Basic Vitaly Programming 2 THE VITALY PROGRAMMING LANGUAGE

```
12 delete ary;
```

Note that the `[]`-operator has higher precedence than the other operators introduced this far. The program outputs:

```
1 12
2 11
3 10
```

Line 2 allocates memory for an `int` array with length 3. Line 5 initializes the array elements, note that a vitality array always is 0-indexed, that is the first element of a vitality array is located at index 0.

Line 10 prints the output and the last line deallocates memory used by the array.

Indexing an array with a negative index or indexing an array with an index larger than or equal to the length of the array results in undefined behaviour and should be avoided.

We can try to allocate an array with negative length, but doing so does not allocate any memory, it simply initializes the array with the special `null` value. E.g.:

```
1 a:array of int;
2 allocate a of length -1;
3 write a;
4 delete a;
```

Produces output:

```
1 null
```

Notice that it was not an error to `delete` the array when it's initialized with `null`.

A `string` is much like an `array` in that we index the string in much the same way. E.g.:

```
1 s:string = "hello";
2 i:int = 0;
3 while i < 5 do {
4   c:char = s[i]; # Note that s[i] is a char
5   write c;
6   i = i + 1;
7 }
```

Output:

```
1 h
2 e
3 l
4 l
5 o
```

Next we might be tempted to do the following:

```
1 s:string = "hello";
2 s[0] = 'H';
```

However when compiling this program the vitality compiler produces an error telling us that the `string` elements are immutable, meaning we are not allowed to modify the `string` like that.

If we need to modify the `string` elements we must use an `array of char` instead. Example:

```
1 a:array of char;
2 allocate a of length 6;
3 s:string = "hello";
4 i:int = 0;
5 while i < 6 do {
6     a[i] = s[i];
7     i = i + 1;
8 }
9 a[0] = 'H'; # fine
10 delete a; # remember to deallocate the array
```

Notice that we allocate an array of length 6 and not 5. We do this because vitality strings, like C-strings, are 0-terminated. That means the string "hello" has a size of 6 chars and that the last char is the NULL char with ASCII code 0.

Later we will talk about how to convert an array of char to a string.

If we would like to obtain the length of an array we can use the `| |`-operator.

E.g.:

```
1 a:array of char;
2 b:array of string;
3 allocate a of length 0;
4 allocate b of length 10;
5 write |a|;
6 write |b|;
7 delete a;
8 delete b;
```

Which first outputs the length of `a` and afterwards outputs the length of `b`:

```
1 0
2 10
```

Again the primitive `string` data type differs from the `array` in that we cannot obtain the length of a `string` using the `| |`-operator. If we want to compute the length of a string we can search for the terminating NULL character as follows:

```
1 s:string = "hello";
2 len:int = 0;
3 while s[len] != 0 do
4     len = len + 1;
5 write len;
```

which outputs the length of the string "hello":

```
1 5
```

Vitaly supplies other ways to find the length of a 0-terminated `string`. We are going to talk more about finding the length of a strings in a later subsection.

Like with strings it is possible to explicitly assign arrays with `null`:

```
1 a:array of bool = null;
2 s:string = null;
3 # a[0] = true; program will crash.
4 # write s[0]; program will crash.
```

2.1.6 Type Cast

The vitality programming language offers the opportunity to explicitly cast from one type to another. For example:

2.1 Basic Vitaly Programming 2 THE VITALY PROGRAMMING LANGUAGE

```
1 i:int = 0;
2 b:bool = true;
3 write cast(bool) i;
4 write cast(int) b;
```

Output:

```
1 false
2 1
```

We can also use the `cast` to cast from array of char to string as follows:

```
1 a:array of char;
2 allocate a of length 4;
3 a[0] = 'a'; a[1] = 'r'; a[2] = 'y'; a[3] = 0;
4 s:string = cast(string) a;
5 write s;
6 delete a; # do not deallocate a before we are done using s.
```

Output:

```
1 ary
```

What's important to note in the above example is that we can `cast` from array of char to string, but we must be careful not to delete the array of char before we are done using the string.

Type casting one array type to another is not possible, however it is possible to cast an array or string to bool. When the array or string is null the cast evaluates into false otherwise it evaluates into true.

Note that we can get the same effect with `ary != null` and `str != null`.

2.1.7 Records

The `record` is used for creating our own user defined data types. It is vitaly's answer to Java's classes and C's structs.

We declare a record as follows:

```
1 var point:record of {
2   x:int,
3   y:int
4 };
```

Notice that the `var` keyword is optional and that you can use the `;` instead of `,` inside the record. The following declarations are equivalent:

```
1 var point:record of {
2   var x:int,
3   var y:int
4 };
```

```
1 point:record of {
2   x:int;
3   y:int;
4 };
```

Like with arrays we `allocate` a record before using it. If we would like to use the `point` record we would allocate it as follows:

```
1 allocate point;
2 point.x = 1;
3 point.y = 2;
4 write point.x + point.y;
5 delete point;
```

Output:

```
1 3
```

Notice the `.`-operator used to access the records fields `x` and `y`. Like arrays, records may contain any data type. So we can declare arrays of records and we can declare records with arrays. Example:

```
1 outer:record of {
2     innerArray:array of record of {
3         s:string;
4         c:char;
5     };
6     innerRec:record of {
7         b:bool;
8         i:int;
9     };
10 };
11 allocate outer;
12 allocate outer.innerArray of length 1;
13 allocate outer.innerArray[0];
14 allocate outer.innerRec;
15
16 outer.innerArray[0].s = "Inner string";
17 outer.innerArray[0].c = 'C';
18 outer.innerRec.b = false;
19 outer.innerRec.i = 42;
20
21 delete outer.innerRec;
22 delete outer.innerArray[0];
23 delete outer.innerArray;
24 delete outer;
```

The `.` and `[]` operators have the same precedence and evaluates from left to right. So, for example, `outer.innerArray[0].s` means get `innerArray` from `outer`, get record at index `0` from `innerArray`, in the end, get field `s` from that record.

Like with array and string you can assign a record with null if needed.

Records are much more powerful than what we have seen this far. We will take a look at what vitaly records really are capable of in a later subsection.

2.1.8 Type Definitions

Say that we have two records `p1` and `p2` of the same type. E.g.:

```
1 p1:record of {
2     x:int;
3     y:int;
4 };
5 p2:record of {
6     x:int;
7     y:int;
8 };
```

Instead of typing the same record fields and their types over and over whenever we want to declare a record we can use a type definition, as follows:

2.1 Basic Vitaly Programming 2 THE VITALY PROGRAMMING LANGUAGE

```
1 type Point = record of {
2   x:int;
3   y:int;
4 };
5 p1:Point;
6 p2:Point;
```

Which is equivalent to the previous example, however in this example we have defined the type `Point` we can reuse whenever we want to declare a `record` of that type.

If we for some reason want to change something in the record, adding a new field for example, we only have to do this one place instead of having to locate all the places where we have declared a record of that type such that we can add the field to the declaration.

It also makes the code better self-documenting since we are forced to give the record a type name.

In general it is considered good practice to always give the records a type name.

Type names are not reserved for records. We can give any type we want a new type name. E.g.:

```
1 type A = array of bool;
2 type C = char;
3 type I = int;
4
5 a:A;
6 allocate a of length 2;
7 a[0] = true;
8 a[1] = false;
9 c:C = 'A';
10 i:I = 10;
11
12 write a[0];
13 write a[1];
14 write c;
15 write i;
16
17 delete a;
```

Which outputs:

```
1 true
2 false
3 A
4 10
```

In general vitality is quite relaxed with what you are allowed to do with type definitions:

```
1 type A1 = A2;
2 type A2 = A1;
3 type A1 = int;
4
5 type P = record of { s:string };
6 type Q = record of { s:string };
7 type P = Q;
8 type P = record of { s:string };
9 type T = Q;
```

All of this is fine. Types `A1` and `A2` are aliases for type `int` and `P`, `Q` and `T` are of the same type `record of { s:string }`.

The following example does not compile:

```
1 type I = int;
2 type I = bool; # error I was previously defined as int.
3
```

2.1 Basic Vitaly Programming 2 THE VITALY PROGRAMMING LANGUAGE

```
4 type A = record of { i:int };
5 type B = record of { b:bool };
6 type A = B; # error A was previously defined as another record type.
7
8 type P = Q;
9 type Q = P; # Cycle in type definitions.
```

Although there was a cycle in the type definitions in the previous example involving A1 and A2, the compiler was able to resolve the type of A1 and A2 because of the A1 = int; line of code.

There is no such line in this last example with P and Q, and thus vitality is unable to determine the type of P and Q.

2.1.9 Functions

We can define a function as follows:

```
1 func foo():int
2   return 0;
3 end foo
```

There are quite some things to notice in this example.

First we start the function definition of the function called foo with the func keyword and we specify that the function produces an int result with the :int syntax.

Inside the body of the function we return the int value 0, and afterwards we end the function definition with the end keyword followed by the name of the function foo.

Thus this function always produces the result 0 when called. E.g.:

```
1 i:int = foo();
2 write i;
3 write foo();
```

Outputs:

```
1 0
2 0
```

A function can take any number of arguments:

```
1 func a(i:int, b:bool, a:array of int):int
2   write i;
3   write b;
4   if a == null then
5     return 0;
6   return a[0];
7 end a
8 write a(10, true, null);
```

Outputs:

```
1 10
2 true
3 0
```

We can nest functions inside other functions:

```
1 func outer(outParm:bool):string
2   ret:string = middle();
```

2.1 Basic Vitaly Programming 2 THE VITALY PROGRAMMING LANGUAGE

```
3
4     func middle():string
5         func inner():bool
6             return outParm;
7         end inner
8
9         if inner() then
10             return "yes";
11         else
12             return "no";
13         end middle
14
15     return ret;
16 end outer
17 s1:string = outer(true);
18 s2:string = outer(false);
19 write s1;
20 write s2;
```

Produces output:

```
1 yes
2 no
```

An important thing to notice here is that the `inner` function is able to access the `outer` function's parameter `outParm`.

The rules for declaring variables inside functions are the same as outside functions. A nested function can access the containing function's variables and the global variables not declared inside a function. Example:

```
1 type a = int;
2 a(1);
3 func a(a:a):int # function, parameter and type with the same name is fine
4     func b():int
5         a = 100;
6         glob = 1000;
7         return 10;
8     end b
9     write a;
10    b:int = b();
11    write b;
12    write a;
13
14    return 0;
15 end a
16 glob:int;
17 write glob;
```

Which outputs:

```
1 1
2 10
3 100
4 1000
```

Also notice that we can declare variables, functions and type definitions with the same name if we want.

Functions can return any type we can use in the vitality programming language including a special type `void` (which is not really a type).

It is used when defining a function that does not return any result. Example:

```
1 f();
2 func f():void
3     write "Hello from f()";
4 end f
```

2.1 Basic Vitaly Programming 2 THE VITALY PROGRAMMING LANGUAGE

Which prints:

```
1 Hello from f()
```

What we should notice here is that the function `f` doesn't have a `return` statement, which is required when the function's return type is different from `void`.

We cannot use `void` for anything else than specifying that a function doesn't return anything.

Now that we have introduced functions another version of the "Hello world" program can be made using a `main` function:

```
1 func main():int
2     write "Hello world";
3     return 0;
4 end main
```

Saving this to a file called `main_hw.vit` and compiling with:

```
1 vitality -m main_hw.vit
```

outputs an executable `a.out` which prints:

```
1 Hello world
```

when it's executed.

Note that we used the `-m` option when we invoked `vitality` to tell the compiler there was a `main` function.

The `main` function must have return type `int` and the value returned from `main` will become the exit status of the executed program, like we are used to with C.

An alternative definition of the `main` function follows:

```
1 func main(args:array of string):int
2     i:int = 0;
3     while i < |args| do {
4         write args[i];
5         i = i + 1;
6     }
7     return 0;
8 end main
```

Saving in a file called `main_args.vit` and compiling with:

```
1 vitality -m main_args.vit
```

Executing the `a.out` executable with command line arguments:

```
1 ./a.out hello outside world
```

will output:

```
1 ./a.out
2 hello
3 outside
4 world
```

Notice that the name of the executable is the first element of the `args` array.

The `main` function must have empty parameter list or one array of string parameter. Anything else is a compile time error.

2.1.10 Overloading functions

In the vitaly language it is possible to overload functions, hence the following is allowed:

```
1 func foo():int
2   return 0;
3 end foo
4
5 func foo(i:int):int
6   return i;
7 end foo
8
9 write foo();
10 write foo(10);
```

Which will output

```
1 0
2 10
```

When calling an overloaded function the function with parameters "best matching" the arguments is called. For example:

```
1 func foo(i:int):void
2   write "1";
3 end foo
4 {
5   func foo(c:char):void
6     write "2";
7   end foo
8
9   foo(1);
10 }
```

Outputs:

```
1 1
```

Without the outermost definition of the function `foo`, the argument `1` would get casted to a `char` and the output would have been `2`.

If multiple overloaded versions of a functions match the arguments equally well the closest definition is called.

To determine which function is closer the distance in terms of scopes is considered. Note that we are going to discuss scopes more carefully in a later subsection.

If multiple functions match equally well and are equally close, an error is reported.

Functions are not allowed to be overloaded based on return type. Hence the following vitaly program is illegal:

```
1 func foo():int # multiple definitions of foo
2   return 0;
3 end foo
4
5 func foo():bool # multiple definitions of foo
6   return false;
7 end foo
8
9 i:int = foo();
10 b:bool = foo();
```

Since function overloading ambiguities can arise. In the following example the two function definitions match equally well, therefore neither can be identified as the

correct one and the compiler reports an error:

```

1 func foo(x:int, y:char):void
2 end foo
3
4 func foo(y:char, x:int):void
5 end foo
6
7 a:int, b:int;
8 foo(a, b); # Ambiguous call to foo

```

If one of the variables in the above example, say `a`, had been of type `char` instead of `int`, then there would be no ambiguity since function `foo(char, int)` would be an exact match and vice versa.

The ambiguity would also be resolved if `foo` was overloaded with the types `foo(int, int)`.

2.1.11 Scope

The body of a function and the body of a record opens a new scope. In fact, whenever we write an opening `{` we open a new scope. Thus in the following example there are 4 different scopes:

```

1 # Global scope.
2 {
3     # Scope inside opening { and closing }.
4 }
5 func foo():void
6     # Scope inside function foo.
7     r:record of {
8         # Scope inside record r.
9     };
10 end foo

```

We are not allowed to declare a variable with the same name twice in the same scope. Although `i` is declared with the same type, the following is an error:

```

1 i:int;
2 i:int;

```

This is also an error:

```

1 j:string;
2 j:char;

```

The following is not an error:

```

1 i:int = 1;
2 if i == 1 then { # New scope between { and }.
3     write i;
4     i:int = 2;
5     write i;
6 }
7 write i;

```

Output:

```

1 1
2 2
3 1

```

The inner scope inside the `if` statement body refers to the `i` variable in the outer global scope until the `i` variable inside the body of the `if` statement is declared.

The outer global scope refer to the `i` variable from it's own scope. We must declare variables before we use them.

Almost the same thing applies to type definitions and function definitions. However, if there is a type definition of type `T` in the current scope, we always refer to that `T` and never a `T` from an outer scope. The same thing applies to functions with equally matching parameter lists. E.g.:

```
1 type T = int;
2 i:T;
3 func f():void
4     i = 1;      # outer i
5
6     i:T;        # inner T (bool)
7
8     f();        # inner f()
9
10    func f():void
11        i = false; # inner i
12    end f
13
14    type T = bool;
15
16    write i;     # inner i
17end f
18f();
19write i;
```

The program outputs:

```
1 false
2 1
```

If `vitaly` is compiled with the `NEST_ACCESS=after` option enabled, nested functions can access variables declared after the function definition. For example:

```
1 i:int;
2 func outer():void
3     func inner():void
4         i = 33;
5     end inner
6     inner();
7
8     i:int;
9
10    write i;
11end outer
```

Output:

```
1 33
```

The noticeable thing here is that the inner function `inner` is accessing the `i` variable declared inside `outer` and not the global variable `i`.

This is probably not intuitive for most programmers, and is not the default behaviour. By default, when `vitaly` is not compiled with the `NEST_ACCESS=after` makefile option, `inner` will refer to the global `i` and give a warning about the variable `i` in function `outer` not being initialized before use.

The global scope is special regarding initialization of variables. Consider the following example:

```

1 global:int;
2 write global;
3 func f():void
4     local:int;
5     write local;
6 end f
7 f();

```

Compiling this example the vitality compiler gives a warning telling us that the `local` variable is uninitialized before use, however it does not say anything about the `global` variable.

This is so because the variables in the global scope are automatically initialized to `0` or `null` when they are declared.

2.1.12 Import

It is possible to import global variables, functions and types from other vitality source files using the `import` keyword. In the following simple example two files are considered `file1.vit` and `file2.vit`. In this subsection the files are assumed to be placed within the same directory, and compiled with:

```

1 vitality -r file1.vit

```

which will recursively compile and link the files into one executable.

| file1.vit: | file2.vit: | |
|------------------------------|--|---|
| 1 <code>import file2;</code> | 1 <code>func foo(i:int):void</code> | 1 |
| 2 | 2 <code> write i;</code> | 2 |
| 3 <code>a:A;</code> | 3 <code>end foo</code> | 3 |
| 4 <code>allocate a;</code> | 4 | 4 |
| 5 | 5 <code>type A = record of { i:int };</code> | 5 |
| 6 <code>a.i = 10;</code> | 6 <code>b:int = 10;</code> | 6 |
| 7 <code>foo(a.i);</code> | | 7 |
| 8 <code>b = b * 2;</code> | | |
| 9 <code>foo(b);</code> | | |
| 10 | | |
| 11 <code>delete a;</code> | | |

By line 1 of `file1.vit`, it is now possible to use everything declared or defined in the global scope of `file2.vit`. Note that only the base name of `file2.vit` is used to indicate which file to use, hence the `.vit` file extension is omitted. The imported symbols are considered as residing in a parent scope to where they are specified. This implies that local symbols suppress imported symbols if name collisions occur, just like regular scope rules. Hence the following example can compile to a valid vitality program:

| file1.vit | file2.vit: | |
|-------------------------------------|-------------------------------------|---|
| 1 <code>import file2;</code> | 1 <code>import file1;</code> | 1 |
| 2 <code>a = 10; # imported a</code> | 2 <code>a = 30; # imported a</code> | 2 |
| 3 <code>a:int;</code> | 3 <code>a:int;</code> | 3 |
| 4 <code>a = 20; # local a</code> | 4 <code>a = 40 # local a</code> | 4 |
| 5 <code>write a; # prints 20</code> | 5 <code>write a; # prints 40</code> | 5 |

As mentioned earlier the code of the global scope is executed before a `main` function. There is no guarantee however in which order the global scopes will be executed in the compiled program. Thus it is undefined whether the program above will output `20` followed by `40` or the other way around. Therefore we cannot rely on which files are

2.1 Basic Vitaly Programming 2 THE VITALY PROGRAMMING LANGUAGE

initialized first. An exception to this is the vitaly standard library which is guaranteed to be initialized before the normal global scope we have seen this far.

Thus variables imported from the standard library can be used in the global scope. We will introduce the vitaly standard library in the following subsection.

Cycles among import statements, as in the example above, are allowed. Even type definitions can be defined with cycles across imported files, by the same rules as for normal type definitions. Hence the following is a valid vitaly program, since all types can be resolved to `int`:

file1.vit

```
1 import file2;
2 type A = B;
3 type C = D;
4 type C = int;
```

file2.vit:

```
1 import file1;
2 type B = C;
3 type D = A;
4 b:B = 10;
5 write b;
```

Outputs:

```
1 10
```

As mentioned there are no guarantee to which order the global scopes of imported files are executed. Only that they will all be executed before an optional call to a main function. Consider the following example:

file1.vit

```
1 import file2;
2 import file3;
3
4 (* We don't know whether a is
5   initialized to 20 or whether
6   b is initialized to 10 here *)
7
8 func main(args:array of string):int
9   write a + b;
10   i:int = 1;
11   while i < |args| do {
12     write args[i];
13     i = i + 1;
14   }
15   return 0;
16 end main
```

file2.vit:

```
1 import file3;
2 b:int = 10;
```

file3.vit:

```
1 import file2;
2 a:int = 20;
```

If `a` or `b` is accessed in the global scope of `file1.vit`, as the comment points out the variables might not yet have been initialized and hence the behavior is undefined. When using the variables inside the main function, we know that all global sections of all imported files have been executed and the variables will therefore be initialized.

If the program above is compiled and run as follows:

```
1 vitaly -r -m file1.vit
2 ./a.out hello world
```

The program will output.

```
1 30
2 hello
3 world
```

Note that the first string of the array passed to the main function is the name of the executable itself, hence to access the command line arguments one must start from index 1 of the array.

Using imports can result in name collisions between symbols. Consider the following example:

| | | |
|---|--|---|
| <pre> file1.vit 1 import file2; 2 import file3; 3 4 a:A; # Conflicting type definitions 5 6 b:B; </pre> | <pre> file2.vit: 1 type A = bool; 2 3 type B = array of string; </pre> | <pre> file3.vit: 1 type A = int; 2 3 type B = array of string; </pre> |
|---|--|---|

It is clear that the type `A` cannot be resolved since the imported definitions are conflicting. In contrast to this type `B` is resolved just fine since there is no ambiguity in this regard.

Imports statements are not restricted to the top of a vitality source file, in fact imports can be placed anywhere a type definition can. Nested in functions, records, and even multiple times. This means that symbol collision can be avoided by nesting the import statements. For example:

| | | |
|--|--|--|
| <pre> file1.vit 1 func foo():void 2 write "File1"; 3 end foo 4 5 { 6 import file2; 7 foo(); 8 } 9 foo(); 10 { 11 import file3; 12 foo(); 13 } </pre> | <pre> file2.vit: 1 func foo():void 2 write "File2"; 3 end foo </pre> | <pre> file3.vit: 1 func foo():void 2 write "File3"; 3 end foo </pre> |
|--|--|--|

Will output:

```

1 File2
2 File1
3 File3

```

Note how all ambiguities have been avoided by nesting the imports while still allowing three different version of the function `foo`. This also applies to variables and type definitions.

When importing a type from one file which in turn is defined by an import from another file, the indirect import does not need to be specified in the first file. Consider the following example:

| | |
|---|---|
| <pre> 1 import file2; 2 import file4; 3 4 # b:B # requires import file3; 5 a:A; 6 allocate a; 7 8 a.i = 10; 9 a.b = true; 10 11 foo(a); 12 13 delete a; </pre> | <pre> file2.vit: import file3; type A = B; </pre> <hr/> <pre> file3.vit: type B = record of { i:int; b:bool; }; </pre> <hr/> <pre> file4.vit: import file3; func foo(b:B):void write b.i; write b.b; end foo </pre> |
|---|---|

Outputs:

| | |
|---|------|
| 1 | 10 |
| 2 | true |

Notice how `file1.vit` can access the fields of the record type `B` declared in `file3.vit`, although it is not directly imported. Even the function call on line 13 is valid since `a` is actually resolved to type `B`. As indicated on line 4, it is not possible to use the type `B` directly from `file1.vit` without directly importing `file3.vit`.

Nested imports can also override other imported definitions, imported in outer scopes:

2.1 Basic Vitaly Programming 2 THE VITALY PROGRAMMING LANGUAGE

| | |
|--|---|
| <div>file1.vit</div> <div><hr/><pre>1 func foo():void 2 write "File1"; 3 foo(); 4 5 func inner():int 6 import file3; 7 foo(); 8 { 9 import file2; 10 import file2; 11 foo(); 12 import file2; 13 } 14 end inner 15 16 inner(); 17 18 import file2; 19 end foo 20 21 foo();</pre><hr/></div> | <div>file2.vit:</div> <div><hr/><pre>1 func foo():void 2 write "File2"; 3 end foo</pre><hr/></div> <div>file3.vit:</div> <div><hr/><pre>1 func foo():void 2 write "File3"; 3 end foo</pre><hr/></div> |
|--|---|

Will output:

```
1 File1
2 File2
3 File3
4 File2
```

Again imports are scope specific and can be placed wherever in any given scope. Note how both of the function calls in line 3 and 11, in the example above, refer to the definition of `foo()` found in `file2.vit`.

2.1.13 Organizing a Vitaly Project

So far all examples considered assumed the files to be in the same directory. This is however not necessary using keyword `package`. By using `package` vitaly source files can be grouped into packages much like Java. See the following example:

| | |
|--|--|
| <pre> file1.vit 1 foo(); 2 foo(200); 3 foo(true); 4 5 import pack.file2; 6 7 func foo(b:bool):void 8 write 300; 9 end foo 10 11 import pack.file3; </pre> | <pre> pack/file2.vit: 1 package pack; 2 import file3; 3 4 func foo():void 5 foo(100); 6 end foo </pre> <hr/> <pre> pack/file3.vit: 1 package pack; 2 3 func foo(i:int):void 4 write i; 5 end foo </pre> |
|--|--|

Will output:

```

1  100
2  200
3  300

```

Notice how the package name of the imported files are included in the import statements. The package name can be omitted if the imported file is located in the same package like `pack/file2.vit` can import `pack/file3.vit` simply by "`import file3;`" since they both are located in the package `pack`. If files are imported with "`import pack.name;`" then the imported file must specify a package, namely `pack`, otherwise an error is reported. Also if a file specifies a package it must be placed in a similar directory structure, hence a file `file.vit` specifying "`package pack1.pack2.pack3;`" must be located in a directory `*/pack1/pack2/pack3/`

2.2 Advanced Vitaly Programming

2.2.1 Finalize

The `finalize` keyword have two different purposes in vitality. It can both be used to move code to a finalize subsection of the executable, and to define `finalize` functions which are vitality's answer to destructors. This will be discussed in the subsection on record destructors later.

For now let us consider the first use. The finalize subsection of the executable, is executed when all other code has been executed, i.e. after the code from the global scope and the optional `main` function:

```

1  write "global1";
2  finalize
3      write "finalize1";
4  write "global2";
5  finalize {
6      write "finalize2";
7      write "finalize3";
8  }
9  write "global3";

```

```

1  global1
2  global2

```

2.2 Advanced Vitaly Programming THE VITALY PROGRAMMING LANGUAGE

```
3 global3
4 finalize1
5 finalize2
6 finalize3
```

In the example above the code in the body of `finalize` is moved to the `finalize` subsection of the executable. This can be particularly useful when used for freeing memory, after execution. For example:

```
1 allocate rec;
2 finalize
3   delete rec;
4 ...
5 # use rec
6 ...
```

This makes sure that the record `rec` will eventually be deleted, but only after all the code of the global scope and possible main function has been executed.

2.2.2 Inheritance

The vitality programming language has good support for object oriented programming. For example:

```
1 type B = record of {
2   s:string;
3 };
4 type D = record of B { # record D extends/inherits record B
5   c:char;
6 };
7
8 func f(b:B):void
9   write b.s;
10 end f
11
12 d:D;
13 allocate d;
14 d.c = 'c';
15 d.s = "str";
16
17 f(d);
18
19 write d.c;
20
21 delete d;
```

which prints:

```
1 str
2 c
```

The record `d` with type `D` is automatically casted to it's base record type `B` when passed as argument to function `f`.

A record can inherit as many records we want:

```
1 type A = record of {
2   a:int;
3 };
4 type B = record of {
5   b:bool;
6 };
7 type C = record of {
8   c:char;
```

2.2 Advanced Vitaly Programming THE VITALY PROGRAMMING LANGUAGE

```
9  };
10 type ABC = record of A, B, C {};
11
12 abc:ABC;
13 allocate abc;
14 a:A = abc;
15 b:B = abc;
16 c:C = abc;
17
18 a.a = 10;
19 b.b = true;
20 c.c = 'c';
21
22 write abc.a;
23 write abc.b;
24 write abc.c;
25
26 delete abc;
```

Note that `abc` is automatically casted to the correct base records in the assignment statements. Output:

```
1  10
2  true
3  c
```

Alternatively:

```
1  type A = record of {
2    a:int;
3  };
4  type B = record of A {
5    b:bool;
6  };
7  type C = record of B {
8    c:char;
9  };
10 type ABC = record of C {};
11
12 abc:ABC;
13 allocate abc;
14 a:A = abc;
15 b:B = abc;
16 c:C = abc;
17
18 a.a = 10;
19 b.b = true;
20 c.c = 'c';
21
22 write abc.a;
23 write abc.b;
24 write abc.c;
25
26 delete abc;
```

Same output:

```
1  10
2  true
3  c
```

Multiple inheritance is often referred to as a good structuring tool. On the other hand multiple inheritance is criticized because it increases complexity of programs when misused.

In reality, how to best structure the inheritance hierarchy is a design issue and varies from program to program.

Consider the diamond problem:

2.2 Advanced Vitaly Programming THE VITALY PROGRAMMING LANGUAGE

```
1 type Base = record of {
2   baseVal:int;
3   # ...
4 };
5
6 type Left = record of Base {
7   leftName:string;
8   # ...
9 };
10
11 type Right = record of Base {
12   rightName:string;
13   # ...
14 };
15
16 type Derived = record of Left, Right {
17   # ...
18 };
19
20 d:Derived;
21 allocate d;
22 d.baseVal = 0; # Ambiguous field reference
23 delete d;
```

Both record `Left` and record `Right` inherit a field named `baseVal` from record `Base`, so `Derived` actually has two fields named `baseVal`.

When compiling the program there is a problem on line 22, the `vitaly` compiler does not know whether we want to reference the `baseVal` from record `Left` or the `baseVal` from record `Right`.

Thus `vitaly` is unable to compile the program and gives an error about an ambiguous field reference.

A solution to the problem is to explicitly tell the compiler which `baseVal` field we want to reference using type cast. E.g.:

```
1 (cast(Left) d).baseVal = 0; # Set baseVal inherited from Left
2 (cast(Right) d).baseVal = 1; # Set baseVal inherited from Right
3
4 # We don't need cast to reference the other fields:
5 d.leftName = "left"; # Fine.
6 d.rightName = "right"; # Fine.
```

Notice that the `.`-operator has higher precedence than `cast`, thus we use parenthesis to cast `d` before referencing the `baseVal` record field.

Although this surely solved the ambiguity problem, facing the diamond problem almost always indicates that there is a problem in the design, and we probably need to consider whether we are doing something wrong.

When records are implicitly casted to extended (base) records. We might think that it is also possible to implicitly cast from a base record to a derived record type:

```
1 type B = record of { (* ... *) };
2 type D = record of B { (* ... *) };
3
4 func useD(d:D):void
5   # ...
6 end useD
7
8 d:D;
9 allocate d;
10 b:B = d;
11
12 useD(b); # Error, incompatible record type
13 delete d;
```

2.2 Advanced Vitaly Programming THE VITALY PROGRAMMING LANGUAGE

When compiling this program `vitaly` gives us an error telling us that it cannot find a function `useD` with a parameter list that matches the arguments given on line 12.

What we need is an explicit `cast` :

```
1 useD(cast(D) b); # Fine, but our own responsibility.
```

Casting `b` back to record `d` solves the problem for us.

We should note that it is our own responsibility that the `cast` of record `b` to type `D` is valid. In the previous example it was fine, but in the following example it is not:

```
1 b2:B;
2 allocate b2;
3 useD(cast(D) b2); # Likely runtime error.
4 delete b2;
```

As we have already seen, casting from type `B` to type `D` is possible because `D` inherits `B`.

The problem here is that record `b2` is allocated as a record with type `B`, so casting it to type `D` is not correct.

Casting from a base type to a derived type can be disastrous, and for that reason `vitaly` does not do such type casts implicitly. Having to do an explicit `cast` makes it easier to spot the mistake when it occurs.

We cannot type cast between two unrelated types:

```
1 type A = record of {i:int};
2 type B = record of {b:bool};
3
4 a:A;
5 b:B;
6
7 allocate a;
8 b = cast(B) a; # Error, cast between unrelated record types.
9 delete a;
```

Casting from a record of type `A` to a record of type `B` is a compile time error.

2.2.3 Structural Equivalence

Vitaly uses structural equivalence when comparing records. E.g.:

```
1 type C1 = record of {
2   c:char;
3 };
4 type C2 = record of {
5   c:char;
6 };
7 c1:C1;
8 c2:C2;
9 c2 = cast(C2) c1; # Fine, cast to record of same type.
```

Casting from type `C1` to type `C2` is fine because they have equivalent structure. In fact we don't even need the `cast` to assign `c2` with `c1`, `c2 = c1` is also fine.

The structural equivalence in `vitaly` is different from what we normally think of as structural equivalence. In `vitaly` the record field names are significant when deciding whether records are equivalent. For example:

```
1 type A = record of {a:int};
2 type B = record of {b:int};
3 type T = record of {a:int};
4
```

2.2 Advanced Vitaly Programming THE VITALY PROGRAMMING LANGUAGE

```
5 a:A;
6 b:B;
7 t:T;
8 b = a; # Error assignment from incompatible type.
9 t = a; # Fine, same structure.
```

Line 8 is an error because the field name of the `int` inside record `a` and record `b` differ.

Line 9 is not an error because the `int` inside `t` has the same name as the `int` inside `a`, and thus they have equivalent structure.

A slightly more complex example:

```
1 type A1 = record of {
2   a:bool;
3   next:A2;
4 };
5 type A2 = record of {
6   a:bool;
7   next:A1;
8 };
9 a1:A1;
10 a2:A2 = a1; # Same structure.
11
12 type B1 = record of {
13   b:char;
14   other:B1;
15 };
16 type B2 = record of {
17   b:char;
18   other:B1;
19 };
20 b1:B1;
21 b2:B2 = b1; # Same structure.
```

The following is an error:

```
1 type A = record of {a:int};
2 type T = record of {a:char};
3 a:A;
4 t:T;
5 t = a; # Error.
```

The field names are equivalent but the field types differ.

Structural equivalence also applies with record inheritance:

```
1 type B1 = record of {
2   b:bool;
3 };
4 type A1 = record of B1 {
5   a:int;
6 };
7 type B2 = record of {
8   b:bool;
9 };
10 type A2 = record of B2 {
11   a:int;
12 };
13
14 func test (b2:B2):void
15 # ...
16 end test
17
18 a1:A1;
19 a2:A2;
20 a2 = a1; # Fine, same structure.
21
22 test(a1); # Implicit cast to record of {b:bool}
```

Structural equivalence does have side effects. For example:

```

1 type A = record of {
2   s:string;
3 };
4 type B = record of {
5   s:string;
6 };
7 type D = record of A, B {}; # Error, extending the same record twice

```

When compiling the above program `vitaly` outputs 2 errors telling us that `A` and `B` are inaccessible due to ambiguity.

Imagine that was allowed and then consider:

```

1 func foo(r:record of {b:bool}):void
2   # ...
3 end foo
4 d:D;
5 foo(d); # Cast to A or cast to B?

```

We might think it is possible to type cast our way out of the problem:

```

1 foo(cast(A) d); # Still ambiguous.

```

`A` has type `record of {b:bool}`, thus casting `d` to type `A` is equivalent with:

```

1 foo(cast(record of {b:bool}) d);

```

And we still have the ambiguity.

In practice this will rarely become an issue, the solution is to change a field name or add an extra field inside `A` or `B` such that their structure differs.

It is possible to indirectly extend the same record twice as we saw in the diamond problem:

```

1 type Base = record of {
2   baseVal:int;
3   # ...
4 };
5 type Left = record of Base {
6   leftName:string;
7   # ...
8 };
9 type Right = record of Base {
10    rightName:string;
11    # ...
12 };
13 type Derived = record of Left, Right {
14   # ...
15 };
16
17 func workWithBase(b:Base):void
18   # ...
19 end workWithBase
20
21 d:Derived;
22 allocate d;
23
24 (* Ambiguous, use Base inherited from Left or Right? *)
25 workWithBase(d);
26 (* Fine, use base inherited from Left. *)
27 workWithBase(cast(Left) d);
28 (* Fine, use base inherited from Right. *)
29 workWithBase(cast(Right) d);
30
31 delete d;

```

On line 27 and 29 we `cast` our way out of the ambiguity.

2.2.4 Functions In Records

We can put functions inside records:

```
1 type R = record of {
2   val:int;
3   func set(v:int):void
4     val = v;
5   end set
6
7   func get():int
8     return val;
9   end get
10 };
11 r:R;
12 allocate r;
13 r.set(1);
14 write r.get();
15 delete r;
```

Prints:

```
1 1
```

Note that the functions `set` and `get` can reference the record field `val`. We can explicitly state that we want to reference a field inside a record using the `record` keyword:

```
1 type R = record of {
2   val:int;
3   func set1(val:int):void
4     record.val = val;
5   end set1
6
7   func set2(val:int):void
8     val = val; # Statement with no effect.
9   end set2
10
11   func get():int
12     return val;
13   end get
14 };
15 r:R;
16 allocate r;
17
18 r.set1(70);
19 write r.get();
20
21 r.set2(90);
22 write r.get();
23
24 delete r;
```

Output:

```
1 70
2 70
```

In function `set1` we use the `record` keyword to explicitly say that we want to reference the `val` field from the record and assign it with the parameter `val`. In function `set2` we don't use the `record` keyword and we simply assign the parameter `val` with itself, which doesn't have any effect.

Overriding functions is also possible:

2.2 Advanced Vitaly Programming THE VITALY PROGRAMMING LANGUAGE

```
1 type B = record of {
2     func print():void
3         write "hello from B";
4     end print
5 };
6 type D = record of B {
7     func print():void
8         write "hello from D";
9     end print
10 };
11 func testPrint(b:B):void
12     b.print();
13 end testPrint
14
15 b:B;
16 allocate b;
17 testPrint(b);
18 delete b;
19
20 d:D;
21 allocate d;
22
23 b = d;
24 testPrint(b);
25 delete d;
```

Output:

```
1 hello from B
2 hello from D
```

Function overloading also applies to functions defined within records and functions inherited from parent records:

```
1 type B = record of {
2     func foo(i:int):void
3         write i;
4     end foo
5 };
6
7 type A = record of B {
8     func foo(c:char):void
9         write c;
10    end foo
11    func foo(b:bool):void
12        write b;
13    end foo
14 };
15
16 a:A;
17 allocate a;
18
19 a.foo(1);
20 a.foo('a');
21 a.foo(true);
22
23 delete a;
```

Outputs:

```
1 1
2 a
3 true
```

If we really want to override function `foo` from `B` we must make sure to define the function `foo` inside `A` with the exact same signature. That is the same parameter types and same return type, the parameter names are not significant. For example:

```
1 type B = record of {
2     func f(t:string):void
3     end f
4 };
5 type D = record of B {
6     # Does not override f from B (different parameter type)
7     func f(t:char):void
8     end f
9
10    # Does not override f from B (different return type)
11    func f(t:string):char
12    end f
13
14    # This one does override f from B
15    func f(p:string):void
16    end f
17 };
```

If we want to make sure we don't call a possibly overridden version of a record function, we can directly call a specific function from inside a record:

```
1 type P = record of {
2     func f():void
3         h(); # Record Q overrides h(), so we call Q.h()
4         write "func P.f() ";
5         record[.].h(); # Directly call P.h()
6     end f
7
8     func h():void
9         write "func P.h() ";
10    end h
11 };
12 type Q = record of P {
13     func f():void
14         record[P].f(); # Directly call P.f()
15         write "func Q.f() ";
16     end f
17
18     func h():void
19         write "func Q.h() ";
20    end h
21 };
22 q:Q;
23 allocate q;
24 q.f(); # Calls Q.f()
25 delete q;
```

Outputs:

```
1 func Q.h()
2 func P.f()
3 func P.h()
4 func Q.f()
```

2.2.5 Record Constructors

We use constructors to initialize a record such that invariants regarding the state of the record are respected from the point the record is allocated.

Vitaly supports record constructors using the `record` keyword, constructors must have `void` return type:

```
1 type R = record of {
2     func record(s:string):void
3         data = s;
4     end record
5 }
```

2.2 Advanced Vitaly Programming THE VITALY PROGRAMMING LANGUAGE

```
6     data:string;
7 };
8 r:R;
9 allocate r of record("R data"); # Allocate r and invoke constructor.
10 write r.data;
11 delete r;
```

Output:

```
1 R data
```

In the example we allocate `r` and afterwards invoke its constructor.

If we try to allocate `r` with the `allocate` statement we have been using this far when allocating records (`allocate r;`), `vitaly` gives an error saying that `R` doesn't have a default constructor.

The default constructor is a constructor with an empty parameter list. We can overload the constructor and supply the default constructor if wanted:

```
1 type R = record of {
2     func record():void
3         data = "data 100";
4         val = 100;
5     end record
6
7     func record(s:string):void
8         data = s;
9         val = 200;
10    end record
11
12    func record(s:string, i:int):void
13        data = s;
14        val = i;
15    end record
16
17    func put():void
18        write data;
19        write val;
20    end put
21
22    data:string;
23    val:int;
24 };
25 r11:R;
26 r12:R;
27 r2:R;
28 r3:R;
29
30 # Implicitly invoke default constructor:
31 allocate r11;
32 # Explicitly invoke default constructor:
33 allocate r12 of record();
34 # Invoke constructor with string parameter:
35 allocate r2 of record("data 200");
36 # Invoke constructor with string and int parameter:
37 allocate r3 of record("data 300", 300);
38
39 r11.put();
40 r12.put();
41 r2.put();
42 r3.put();
43
44 delete r11;
45 delete r12;
46 delete r2;
47 delete r3;
```

Outputs:

2.2 Advanced Vitaly Programming THE VITALY PROGRAMMING LANGUAGE

```
1 data 100
2 100
3 data 100
4 100
5 data 200
6 200
7 data 300
8 300
```

Note that the default constructor with empty parameter list is implicitly called when allocating `r11`.

Default constructors of base records are automatically called:

```
1 type A = record of {
2     func record():void
3     write "Hello from A";
4 end record
5 a:int;
6 };
7 type B1 = record of {
8     func record():void
9     write "Hello from B1";
10 end record
11 b:int;
12 };
13 type B2 = record of B1 {
14     func record():void
15     write "Hello from B2";
16 end record
17 };
18 type C = record of A, B2 {};
19
20 c:C;
21 allocate c;
22 delete c;
```

Results in output:

```
1 Hello from A
2 Hello from B1
3 Hello from B2
```

It is an error if the base record does not have a default constructor and we do not invoke it's constructor. For example:

```
1 type B = record of {
2     func record(s:string):void
3     write s;
4 end record
5 };
6
7 # Compile time error, missing call to B's constructor
8 type D1 = record of B {};
9
10 # Compile time error, still missing call to B's constructor
11 type D2 = record of B {
12     func record():void
13     write "hello";
14 end record
15 };
```

We call base record constructors explicitly as follows:

```
1 type A = record of {
2     func record():void
3     a = 333;
4     write "hello from default A constructor";
5 end record
```



```
6      func record(i:int):void
7          a = i;
8      end record
9      a:int;
10 };
11
12 type B = record of {
13     func record(s:string):void
14         b = s;
15     end record
16     b:string;
17 };
18 type AB = record of A, B {
19     func record():void
20         record[A](42);
21         record[B]("yes");
22     end record
23 };
24 ab:AB;
25 allocate ab;
26 write ab.a;
27 write ab.b;
28 delete ab;
```

Output:

```
1 42
2 yes
```

Note that the default constructor of type `A` is not invoked when we're explicitly invoking its constructor. We are also allowed to explicitly invoke the default constructor if we want, but we may not invoke a constructor of the same base record twice. Consider:

```
1 type B = record of {
2     func record():void
3         write "B default constructor";
4     end record
5
6     func record(b:bool):void
7         write b;
8     end record
9 };
10 type D = record of B {
11     func record():void
12         record[B](); # Fine, invoke default constructor
13     end record
14
15     func record(b:bool):void
16         record[B](true);
17         record[B](); # Error, second invocation of B constructor
18     end record
19
20     func record(i:int):void
21         write i;
22         record[B](); # Error, constructor invocations must be first
23     end record
24
25     # Error, we may not invoke a constructor from this function:
26     func foo(i:int):void
27         record[B]();
28     end foo
29 };
```

Note that calls to base constructors must be the first thing we do inside constructors. Also note that we are not allowed to invoke constructors unless we are inside a constructor.

Besides calling constructors of base records, records can call their own constructor:

```

1  type A = record of {
2      func record(i:int):void
3          a = i;
4      end record
5      a:int;
6  };
7  type B = record of {
8      func record(s:string):void
9          b = s;
10     end record
11     b:string;
12 };
13 type AB = record of A, B {
14     func record():void
15         record[] (1, "yes");
16     end record
17
18     func record(val:int, str:string):void
19         record[A] (val);
20         record[B] (str);
21     end record
22 };
23 ab:AB;
24
25 allocate ab;
26 write ab.a;
27 write ab.b;
28 delete ab;
29
30 allocate ab of record(0, "no");
31 write ab.a;
32 write ab.b;
33 delete ab;

```

Prints:

```

1  1
2  yes
3  0
4  no

```

Calls to own constructor must be the first statement. The record is not allowed to invoke base record constructors after invocation of its own constructor.

2.2.6 Records Destructors

The opposite of the constructor is the destructor. The destructor gets called when we delete a record.

We define the destructor using the `finalize` keyword:

```

1  type R = record of {
2      data:array of char;
3      func record(len:int):void
4          write "construct R";
5          allocate data of length len;
6      end record
7
8      func getS():string
9          return "R is constructed";
10     end getS
11
12     func finalize():void
13         write "finalize R, length of data is:";
14         write |data|;
15         delete data;
16     end finalize
17 };
18 r:R;
19 allocate r of record(3);

```

2.2 Advanced Vitaly Programming THE VITALY PROGRAMMING LANGUAGE

```
20 write r.getS();
21 delete r;
```

Outputs:

```
1 construct R
2 R is constructed
3 finalize R, length of data is:
4 3
```

The `finalize` function (destructor) must have empty parameter list and must have `void` return type.

We cannot define more than one destructor per record.

As shown in the following example, destructors of base records are implicitly called when deleting a record:

```
1 type A = record of {
2     func record():void
3     write "construct A";
4     end record
5
6     func finalize():void
7     write "finalize A";
8     end finalize
9 };
10 type B = record of {
11     func record():void
12     write "construct B";
13     end record
14
15     func finalize():void
16     write "finalize B";
17     end finalize
18 };
19 type C = record of A, B {};
20 c:C;
21 allocate c;
22 delete c;
```

Produces:

```
1 construct A
2 construct B
3 finalize A
4 finalize B
```

2.2.7 Interfacing with C

A vitality program can call any C function defined in the glibc library. For example if we would like to find the length of a string we can do as follows:

```
1 extern(C) func strlen(s:string):int;
2 str:string = "hello";
3 write strlen(str);
```

Output:

```
1 5
```

Or if we want the value of an environment variable:

```
1 extern(C) func getenv(name:string):string;
```

```
2 extern(C) func puts(s:string):int;  
3 home:string = getenv("HOME");  
4 if home != null then  
5     puts(home);
```

Which outputs the value of the `HOME` environment variable if it exists.

2.3 Vitaly Standard Library

A small standard library, referred to as `libvitaly`, is included with the `vitaly` compiler. The library consists of the following:

```
std.string  
std.array  
std.comparator  
std.object  
std.math  
std.sort  
std.stdio  
std.vector  
std.indexable  
std.errno  
std.thread  
std.lock  
std.c.string  
std.c.ctype  
std.c.cstdio  
std.c.stdio
```

2.3.1 std.c

This package contains interfaces to C functions. Such as a wrapper for the `strcmp` function of C.

2.3.2 std.object

The `Object` type made available by importing `std.object` contains no functions, and is merely used as a general interface for records in the standard library. This type should only be extended and used as a common interface. Uses of this type becomes clear in the following sections.

2.3.3 std.string

By importing `"import std.string;"`, the following functionality is available: functions `intToString(int):String`, `charToString(char):String` and type `String`.

`String` has constructors which are overloaded to take `int`, `char`, `bool`, `string`, array of `char` or `String` which will be converted to a string and the `String` variable will be initialize to this string

The `String` type has among others the following interesting functions:

`String.append()` and `String.assign()` which either appends to or assigns a variable of type `String`. These have both been overloaded just as thee constructors. All overload returns a self pointer to the variable, such that calls to these functions can be chained:

```
.assign("Hello").append(" ").append("World");
```

`String.compare(oth:String)` returns an int indicating whether the string is smaller greater or equal to another `String` variable.

Note that the `String` type will automatically expand as more memory is needed. Note also that the `String` type is extending the `Object` type. See example of the use of `String` below:

```
1 import std.string;
2
3 s:String;
4 allocate s of record("Hello");
5 s.append(" World!");
6 write s.str(); # to simple type string
7 write s.getLast(); # get last character
8 write s.len(); # get length of the string
9
10 s2:String = s.copy(); # allocate a copy of the string
11 write s.compare(s2); # returns 0 if equal -1 or 1 otherwise
12 s2.assign("Goodbye");
13 write s2.str();
14 write s.compare(s2);
15
16 delete s;
17 delete s2;
```

Will output:

```
1 Hello World!
2 !
3 12
4 0
5 Goodbye
6 1
```

2.3.4 std.errno

By importing `std.errno` we have access to an `errno` variable, which is an interface to the `errno` provided by C. The `Errno` type has the following functions:

`get()` which returns the current value of `errno`.

`set(int)` which sets the value of `errno` and returning the old value of `errno`.

`clear()` which sets the value of `errno` to 0.

`assign(String)` and `append(String)` which respectively appends or assigns the error message corresponding to the current `errno` value, to the string given as argument.

Besides the variable `errno` we also have access to functions that return values of different error types such as `EINVAL()` and `ENOMEM()` which returns the values of the C constants `EINVAL` and `ENOMEM`. See the following example for use of `std.errno`:

```
1 import std.string;
2 import std.stdio;
3 import std.errno;
4
5 msg:String;
6 allocate msg;
7 errno.set(0);
8 errno.assignTo(msg);
9 stdo.putln(msg);
10
11 errno.set(EINVAL);
```

```
12 errno.assignTo(msg);
13 stdo.putln(msg);
14
15 delete msg;
```

Output:

```
1 Success
2 Invalid argument
```

2.3.5 std.stdio

By importing " std.stdio " two global variables `stdo` and `stdi` are made available. `stdo` is of type `StdOstream` and `stdi` is of type `StdIstream`. The output stream `stdo` provides an interface to write to standard out and `stdi` an interface to read from standard in.

2.3.6 Standard out - stdo

Various functions are available to write to the standard output stream `stdo`.

The functions `put()` and `putln()` will write to standard out and when calling `putln()` a line break will follow. Both functions have been overloaded to take `int`, `bool`, `char`, `string`, `String` and `Errno` as parameters. A third function available is `ln()` which simply prints a line break to standard out.

Common to all three functions is that they return a self pointer to the `stdo` variable, which enables chaining of calls to `put()`, `putln()` and `ln()`. See the following example on the use of `stdo`:

```
1 import std.string;
2 import std.stdio;
3
4 s:String;
5 allocate s of record("String");
6
7 a:array of char;
8 allocate a of length 3;
9 a[0] = 'a';
10 a[1] = 'r';
11 a[2] = 'y';
12
13 stdo.putln("raw string")
14     .putln(s)
15     .putln(a)
16     .putln(true)
17     .ln()
18     .putln('c')
19     .put(0 - 12345)
20     .put(" ")
21     .putln(54321);
22
23 delete s;
24 delete a;
```

Outputs:

```
1 raw string
2 String
3 ary
4 true
5
```

```
6 c
7 -12345 54321
```

2.3.7 Standard in - `stdi`

The standard input stream `stdi` have two interesting functions: `get(String)` and `getln(String)`. Both functions takes as parameter a `String` variable to which the read input will be appended. They also both return a self pointer to `stdi`, to allow chained reads from the input stream.

`get()` is a blocking read of the next word ended by whitespace on the input stream. White spaces are stripped before the word. `getln()` is also a blocking read, but it reads a until line break or end of file is reached. See the example below:

```
1 import std.string;
2 import std.stdio;
3
4 s:String;
5 q:String;
6 allocate q of record("q");
7 allocate s;
8
9 while s.compare(q) != 0 do {
10     s.assign("");
11     stdi.getln(s);
12 }
13 delete q;
14 delete s;
```

The example above will keep reading words of the input stream until the word consisting of a single "q" is read.

2.3.8 `std.math`

Currently the only feature available by importing `std.math` is a single function `mod(int, int)` which returns the first parameter modulo the second. Hence

```
1 import std.math
2 write mod(23, 10);
3 write mod(4, 4);
```

will output:

```
1 3
2 0
```

2.3.9 `std.indexable`

The type `Indexable` is meant as an interface to data structures which provides random access to variables by index. The interface made available by this type consists of basic functions like `get(int):Object`, `set(int, Object):void` and `size():int` which must be overridden in types that extends `Indexable`. See `std.array` and `std.vector`.

Here the use of the `Object` type is clear, as it allows the `Indexable` type to be more general.

2.3.10 std.array

The `Array` is used as a fixed length array of objects. The `Array` type extends `Indexable` and overrides the functions of `Indexable`.

```

1 import std.object;
2 import std.array;
3
4 type E = record of Object {
5     s:string;
6     func record(s:string):void
7         record.s = s;
8     end record
9
10    func str():string
11        return s;
12    end str
13
14    func finalize():void
15        delete s;
16    end finalize
17 };
18
19 a:Array;
20 allocate a of record(3);
21 e1:E;
22 e2:E;
23 e3:E;
24 allocate e1 of record("e1");
25 allocate e2 of record("e2");
26 allocate e3 of record("e3");
27
28 a.set(0, e1);
29 a.set(2, e3);
30 a.set(1, e2);
31
32 i:int = 0;
33 while (i < a.size()) do {
34     write (cast(E)a.get(i)).str();
35     i = i + 1;
36 }
37
38 a.destroy();

```

Output:

```

1 e1
2 e2
3 e3

```

2.3.11 std.vector

The `Vector` type is extending `Array` and implements a vector of objects. The `Vector` overrides the `size()` function of `Array` and add a new function `append()` to append `Object`'s to the vector, if more space is needed to append another object, the vector will dynamically resize itself. The use is much similar to that of `Array` discussed in the previous subsection.

See later subsection `std.sort` for an example of use.

2.3.12 std.comparator

The `Comparator` type has a single function `compare(Object, Object):int` and is used as an interface that compares two `Object` type variables. The `Comparator`

should be extended and `Comparator.compare()` overridden such that the object can be cast to the desired type and correctly be compared. See the example below:

```

1  import std.object;
2  import std.comparator;
3
4  type MyType = record of Object {
5      val:int;
6      func record(i:int):void
7          val = i;
8      end record
9  };
10
11 type MyTypeComparator = record of Comparator {
12     func compare(o1:Object, o2:Object):int
13         v:MyType = cast(MyType)o1, v2:MyType = cast(MyType)o2;
14         if(v.val < v2.val) then
15             return 0 - 1;
16         if(v.val > v2.val) then
17             return 1;
18         return 0;
19     end compare
20 };
21
22 v:MyType, v2:MyType;
23 comp:MyTypeComparator;
24 allocate v of record(1);
25 allocate v2 of record(2);
26 allocate comp;
27
28 write comp.compare(v, v2);
29 write comp.compare(v2, v);
30
31 delete comp;
32 delete v2;
33 delete v;

```

Outputs:

```

1  -1
2  1

```

2.3.13 std.sort

By importing "std.sort" a function `sort(Indexable, Comparator):void` is available. This function is sorting the indexable, by quick sort using the `Comparator` to compare two objects in the indexable. Since both `Array` and `Vector` are extending `Indexable` they can be sorted by this sort function.

```

1  import std.sort;
2  import std.comparator;
3  import std.vector;
4  import std.object;
5  import std.stdio;
6
7  type CharVector = record of Vector {
8      func print():void
9          i:int = 0;
10         stdo.put("{ ");
11         while (i < size()-2) do {
12             stdo.put((cast(MyChar)get(i))._c).put(", ");
13             i = i + 1;
14         }
15         i = i + 1;
16         stdo.put((cast(MyChar)get(i))._c).put("}").ln();
17     end print
18 }

```

```

19     func println():void
20         print();
21         stdo.ln();
22     end println
23 };
24
25 type MyChar = record of Object {
26     _c:char;
27     func record(c:char):void
28         _c = c;
29     end record
30 };
31
32 type CharComparator = record of Comparator {
33     func compare(o1:Object, o2:Object):int
34         if ((cast(MyChar)o1)._c < (cast(MyChar)o2)._c) then
35             return 0 - 1;
36         if ((cast(MyChar)o1)._c > (cast(MyChar)o2)._c) then
37             return 1;
38         return 0;
39     end compare
40 };
41
42 vec:CharVector;
43 allocate vec;
44
45 c:char = 'z';
46 while (c >= 'a') do {
47     tmp:MyChar;
48     allocate tmp of record(c);
49     vec.append(tmp);
50     c = c - 1;
51 }
52
53 vec.print();
54
55 comp:CharComparator;
56 allocate comp;
57 sort(vec, comp);
58 delete comp;
59
60 vec.print();
61
62 vec.destroy();

```

Output:

```

1 {z, y, x, w, v, u, t, s, r, q, p, o, n, m, l, k, j, i, h, g, f, e, d, c, a}
2 {a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, z}

```

2.3.14 std.thread

This is the package containing the `Thread` record which we extend when we need an extra thread of execution. We must make sure to compile with the `-thread` option enabled if we want to import the `Thread`.

To create a new thread we extend the `Thread` record and override the `run()` function. This function is called once the thread is started.

The `Thread` record defines a function called `start()`. We use the `start()` function to start the execution of the thread.

After we have started a thread we may call the `join()` function to wait for the thread to finish execution.

The functions inside the `Thread` record return `true` if they succeed and `false` if they fail. Upon failure `errno` is set accordingly.

An example of using the `Thread` record follows:

```
1 import std.thread;
2 import std.stdio;
3 import std.errno;
4 import std.stdlib;
5
6 type Resource = record of {
7     val:int;
8
9     func record():void
10         val = 0;
11     end record
12
13     func inc():void
14         val = val + 1;
15     end inc
16
17     func put():void
18         stdo.putln(val);
19     end put
20 };
21
22 type T = record of Thread {
23     res:Resource;
24     func record(r:Resource):void
25         res = r;
26     end record
27
28     func run():void
29         i:int = 0;
30         while i < 1000000 do {
31             res.inc();
32             i = i + 1;
33         }
34         res.put();
35     end run
36 };
37
38 res:Resource;
39 allocate res;
40
41 t1:T;
42 t2:T;
43 t3:T;
44 allocate t1 of record(res);
45 allocate t2 of record(res);
46 allocate t3 of record(res);
47
48 if t1.error() || t2.error() || t3.error() then {
49     stdo.putln(errno);
50     exit(2);
51 }
52
53 if !t1.start() || !t2.start() || !t3.start() then {
54     stdo.putln(errno);
55     exit(3);
56 }
57
58 if !t1.join() || !t2.join() || !t3.join() then {
59     stdo.putln(errno);
60     exit(4);
61 }
62
63 delete t1;
64 delete t2;
65 delete t3;
66 delete res;
```

Saving this to a file called `nondet_thread.vit` and compiling with

```
1 vitality nondet_thread.vit --thread
```

will output the executable `a.out`.

Since three threads `t1`, `t2` and `t3` are simultaneously accessing the same resource `res` without use of synchronization, the output of this program is nondeterministic.

Whenever we run this program we will often experience different outputs. An example output is:

```
1 1035767
2 1794331
3 2333902
```

2.3.15 std.lock

We import `std.lock` whenever we need to synchronize the execution of threads, `std.lock` exposes the `Lock` record.

The `Lock` record has the functions `lock()` and `unlock()` which are used when we need to acquire and release a resource respectively.

When one thread have a `lock()` on a `Lock` record other threads calling the `lock()` function are put to sleep until the first thread releases the lock by calling `unlock()`.

That is, only one thread is allowed to have a lock at any given time. A modification of the example from the previous subsection using `Lock` is shown below:

```
1 import std.thread;
2 import std.lock;
3 import std.stdio;
4 import std.errno;
5 import std.stdlib;
6
7 type Resource = record of {
8     val:int;
9     lock:Lock;
10
11     func record():void
12         val = 0;
13         allocate lock;
14     end record
15
16     func error():bool
17         return lock.error();
18     end error
19
20     func acquire():bool
21         return lock.lock();
22     end acquire
23
24     func release():bool
25         return lock.unlock();
26     end release
27
28     func inc():void
29         val = val + 1;
30     end inc
31
32     func put():void
33         stdo.putln(val);
34     end put
35
36     func finalize():void
37         delete lock;
38     end finalize
39 };
40
41 type T = record of Thread {
```

```
42     res:Resource;
43     func record(r:Resource):void
44         res = r;
45     end record
46
47     func run():void
48         res.acquire();
49         i:int = 0;
50         while i < 1000000 do {
51             res.inc();
52             i = i + 1;
53         }
54         res.put();
55         res.release();
56     end run
57 };
58
59 res:Resource;
60 allocate res;
61 if res.error() then {
62     stdo.putln(errno);
63     exit(1);
64 }
65
66 t1:T;
67 t2:T;
68 t3:T;
69 allocate t1 of record(res);
70 allocate t2 of record(res);
71 allocate t3 of record(res);
72
73 if t1.error() || t2.error() || t3.error() then {
74     stdo.putln(errno);
75     exit(2);
76 }
77
78 if !t1.start() || !t2.start() || !t3.start() then {
79     stdo.putln(errno);
80     exit(3);
81 }
82
83 if !t1.join() || !t2.join() || !t3.join() then {
84     stdo.putln(errno);
85     exit(4);
86 }
87
88 delete t1;
89 delete t2;
90 delete t3;
91 delete res;
```

Saving that in a file called `det_thread.vit` and compiling with:

```
1 vitally det_thread.vit --thread
```

will give us the executable `a.out`. In contrast to the example from the previous subsection we use the `Lock record` to synchronize the threads. Thus, assuming no errors, the output from this program is deterministic:

```
1 1000000
2 2000000
3 3000000
```

3 Compiler flags

In the following sections the different flags of the compiler will shortly be introduced.

3.1 General flags

General purpose flags:

```

--help (-h)
    display this message
--help=optimize
    display information about optimization options
--help=warning
    display information about warning options
--help=dump
    display information about options for dumping internal
    data structures

--output=OUTFILE (-o OUTFILE)
    write output to OUTFILE
--main (-m)
    insert code that calls function main when linking program
--ign-main
    ignore the --main (-m) option
--lib-init
    put code in the global scope into an initialization section
    executed before the normal initialization section
--ign-lib-init
    ignore the --lib-init option
--thread
    include support for libvitaly threads
--ign-thread
    ignore the --thread option
--no-libvit (-x)
    do not link with libvitaly (standard library becomes inaccessible)
--ign-no-libvit
    ignore the --no-libvit (-x) option
--lib-path=PATH (-L PATH)
    add PATH to the list of paths with library files
--lib=LIB (-l LIB)
    link against library LIB accessible though one of the library
    search paths
--import-path=PATH (-i PATH)
    add PATH to the list of paths searched when looking for import files
--max-msg=n
    do not produce more than n error and warning messages
--stubborn
    keep compiling new input source files, even when errors are detected
--ign-stubborn
    ignore the --stubborn option
--compile-only (-c)
    assemble, compile and keep object files, but do not link
--ign-compile-only
    ignore the --compile-only (-c) option
--asm-only (-s)
    produce assembly output, do not compile or link
--ign-asm-only
    ignore the --asm-only (-s) option
--keep-obj (-k)
    keep generated object files
--ign-keep-obj
    ignore the --keep-obj (-k) option
--verbose (-v)
    print verbose output to standard output
--ign-verbose
    ignore the --verbose (-v) option
--recursive (-r)
    recursive compilation, automatically compile imported files
--ign-recirsive
    ignore the --recursive (-r) option
--gen-viti (-I)
    generate a .viti file used as import file instead of the
    corresponding .vit file
--ign-gen-viti
    ignore the --gen-viti (-I) option

```

3.2 Optimize flags

Options for enabling and disabling optimizations

```
--optimize=OPTION[,OPTION]... (-O OPTION[,OPTION]...)
OPTION
  all [enabled by default]
    enable all optimizations
  ign-all
    disable all optimization options

  const-prop [enabled by default]
    do constant propagation optimization
  ign-const-prop
    ignore the const-prop optimization option
  instr-elim [enabled by default]
    do instruction elimination optimization (dead code elimination)
  ign-instr-elim
    ignore the instr-elim optimization option
  unused-mov [enabled by default]
    try to minimize the number register to register mov instructions
  ign-unused-mov
    ignore the unused-mov optimization option
  unused-set [enabled by default]
    try to eliminate useless set instructions
  ign-unused-set
    ignore the unused-set optimization option
  reg-vars [enabled by default]
    try to keep variables in registers when possible
  ign-reg-vars
    ignore the reg-vars optimization option
  def-to-use [enabled by default]
    move definitions of variables closer to where they are used
  ign-def-to-use
    ignore the def-to-use optimization option
  func-access [enabled by default]
    detect information about which variables functions are using
  ign-func-access
    ignore the func-access optimization option
```

3.3 Warning flags

Options for enabling and disabling warnings

```
--warning=OPTION[,OPTION]... (-w OPTION[,OPTION]...)
OPTION
  all
    enable all warning options
  ign-all
    disable all warning options

  is-error
    treat warnings as errors
  ign-is-error
    ignore the is-error warning option

  no-finalize
    show warning when extended records doesn't have a finalize function
  ign-no-finalize
    ignore the no-finalize warning option
  implicit-cast [enabled by default]
    give warning about implicit type casts
  ign-implicit-cast
    ignore the implicit-cast warning option
  ref-compare [enabled by default]
    give warning when comparing reference types
  ign-ref-compare
    ignore the ref-compare warning option
  overflow [enabled by default]
    give warning when overflow is detected
```

```

ign-overflow
    ignore the overflow warning option
div-zero [enabled by default]
    give warning when division by zero is detected
ign-div-zero
    ignore the div-zero warning option
uninitialized [enabled by default]
    give warning when variables might be uninitialized before use
ign-uninitialized
    ignore the uninitialized warning option

```

3.4 Dump flags

Options for dumping internal data structures:

```

--dump=OPTION[,OPTION]...
OPTION
    parse-tree
        dump XML parse tree
    parse-tree-graph
        dump PDF parse tree
    symbol-table
        dump ACSII symbol table
    symbol-table-graph
        dump PDF symbol table
    init-ic
        dump initial intermediate code representation
    norm-addr-ic
        dump intermediate code representation right after initial
        normalization of addressing instructions
    const-prop-ic
        dump intermediate code after each constant propagation pass
    instr-elim-ic
        dump intermediate code after dead instruction elimination passes
    def-to-use-ic
        dump intermediate code after def-to-use optimization pass
    norm-x86-32-ic
        dump intermediate code after instructions have been converted to
        x86-32 compatible instructions
    unused-mov-ic
        dump intermediate code after elimination of unused mov instructions
    reg-alloc-x86-32-ic
        dump intermediate code right after register allocation
    reg-vars-liveness-ic
        dump liveness analysis before deciding which variables should
        stay in registers
    reg-vars-ic
        dump liveness analysis after register variables have been chosen
    init-liveness-x86-32-ic
        dump liveness analysis before register allocation
    liveness-x86-32-ic
        dump liveness analysis right before assigning pseudo registers to
        x86-32 registers
    reg-alloc-x86-32-ic
        dump intermediate code right after register allocation
    warn-uninit-liveness-ic
        dump liveness analysis before locating variables which might be
        uninitialized before use
    final-x86-32-ic
        dump final intermediate code representation before emitting
        machine code
    c-header
        dump C header file with vitaly record declarations as C structs
    asm
        dump assembly source file before assembling

```

4 Compiler Library

The C programming language has been chosen as implementation language for the vital compiler. The most noticeable advantages of C are efficiency and portability, especially because of the gcc C compiler which we are using in the project.

However, as a general purpose programming language, C does have disadvantages. Noticeable is the lack of commonly used data structures like linked lists, hash maps and search trees in the standard library, and the limited support for object oriented features like inheritance and polymorphism.

To cope with the missing data structures in the C standard library we have implemented a little, but efficient, library offering the most common data structures needed. The report contains a section which shortly describes this library.

The compiler library implements:

1. String data structure.
2. Single linked list.
3. Double linked list.
4. Vector.
5. Dynamic hash map.
6. Red black tree.
7. Debugging facilities.

4.1 String

The `String` data structure is heavily used in the project, thus we are going to take a through discussion of this data structure and how to use it.

The `String` data structure is preferred over the raw `char *` for the following reasons:

- The `String` data type is dynamic. With a single line of code reassign the string without thinking about freeing and allocating memory.
- The `String` data structure is getting stronger and stronger. Every time we need new functionality from the `String` data type it is implemented.
- Our `String` is as easy to initialize as a `char *`.
- The `String` data structure is almost as efficient as `char *`.

How do we use the `String` data structure?

```
1 String s = STRING_INIT(s, "I'm a string");
2 Const_String cs = S("I'm a constant String");
```

In the example above we are initializing a new `String` named `s` with the value `"I'm a string"`. If this happens inside a function the string is allocated on the stack and no string copying is done. The string is internally assigned the `"I'm a string" const char *`. The `S()` macro is defined to allow convenient initialization of constant strings (`Const_String`) as in the example. A decent amount of functions in the project take strings as arguments.

The following code example will show some of the benefits of the `String` data structure:

```

1 // Declare and initialize str with "He"
2 STRING(str, "He");
3 // Internally str is pointing to the const char * "He"
4
5 // Append "llo" to str
6 string_append(str, S("llo"));
7 // Now str is assigned a dynamically allocated char * "Hello"
8
9 // Get a copy of str
10 String cpy = string_duplicate(str);
11
12 // Print: Hello
13 print_message(cpy);
14
15 // Allocate a new string from the format "%S %S",
16 // where %S is to String what %s is to char *
17 String fmt = string_from_format(S("%S %S"), cpy, str);
18
19 // Print: Hello Hello
20 print_message(fmt);
21
22 // Print: Hello Hello Hello
23 print_message(S("%S %S"), str, fmt);
24
25 // Clean up
26 string_clear(str);
27 string_destroy(cpy);
28 string_destroy(fmt);

```

Note that `str` was allocated on the stack and we use `string_clear` to deallocate memory used by that `String`. The strings `cpy` and `fmt` was dynamically allocated and `string_destroy()` was used to deallocate them.

Want more information on strings? Take a look at the `src/str.h` and `src/str.c` appendices.

4.2 Linked Lists

Our library also implements two linked list data structures, single linked list and double linked list. Some of the data structures in the library make use of the object oriented concept "inheritance" the linked lists are among those data structures. E.g. if we want to make use of the double linked list (`Double_List`) we must create a struct which inherits/contains the `Double_List_Node` struct. An example follows:

```

1 struct Db_Entry {
2     Double_List_Node dbnode;
3     String data;
4 };

```

Initializing a new double list and appending a `struct Db_Entry our_entry` to the list works as follows:

```

1 DOUBLE_LIST(dblist);
2 double_list_append(&dblist, &our_entry.dbnode);

```

Think of `&our_entry.dbnode` as casting the `struct Db_Entry` to a `Double_List_Node`.

If we want to iterate through all the nodes in the double linked list `dblist` from the previous example. We do as follows:

```

1 struct Db_Entry *entry;
2 Double_List_Node *n;
3 DOUBLE_LIST_FOREACH(&dblist, n) {
4     entry = DOUBLE_LIST_ENTRY(n, struct Db_Entry, dbnode);
5     process_data(entry->data); // Or whatever
6 }

```

Essentially what we are doing is: for each `Double_List_Node *n` in the double linked list we are casting it back into the inheriting/containing `struct Db_Entry *`.

The `DOUBLE_LIST_ENTRY()` macro makes use of gcc extensions to the C programming language.

The single linked list (`Single_List`) works in much the same way.

For more info on the linked list data structures see the `src/double_list.h`, `src/double_list.c`, `src/single_list.h` and `src/single_list.c` appendices.

4.3 Vector

Essentially the same data structure as the C++ `vector` or Java's `ArrayList`, a dynamic array.

The vector is initialized with some size `n`. When `n` elements are inserted in the vector it automatically expands to size `n := 2*n`. When removing elements from the vector it decreases its size to `n := n/2` when the number of elements is less than `n/4`.

The `Vector` data structure is very efficient, and we don't inherit anything to use it. Calling `vector_append(v, data)` will append data to the end of the `Vector *v`.

For more information on `Vector` check out the `src/vector.h` and `src/vector.c` appendices.

4.4 Dynamic Hash Map

Our dynamic hash map (`Hash_Map`) is using the `Single_List` data structure for chaining. It automatically increases the number of hash slots when the load factor exceeds `0.75` and decreases the number of slots when the load factor drops below `0.1875`.

When initializing a new hash map we must pass it a `Hash_Map_Comparator` function used when searching the hash map for data. An example of inserting data in a `Hash_Map` might be:

```

1 struct Hash_Entry {
2     Hash_Map_Slot hslot;
3     String data;
4 };
5
6 bool comparator(String search_str, Hash_Map_Slot *map_slot) {
7     struct Hash_Entry *e = HASH_MAP_ENTRY(map_slot, struct Hash_Entry, hslot)
8     return string_compare(search_str, e->data) == 0;
9 }
10
11 HASH_MAP(hmap, comparator);
12
13 void insert(String s) {
14     struct Hash_Entry *e = malloc(sizeof(struct Hash_Entry));
15     e->data = string_duplicate(s);

```

```

16 hash_map_insert(&hmap, &e->hslot, string_hash_code(s));
17 }

```

Again, we are making use of inheritance to cast between `struct Hash_Entry *` and `Hash_Map_Slot *`.

The dynamic hash map also has a for-each loop. To emphasize how far we go to provide such a feature, just take a look at the following implementation of the `Hash_Map` for-each loop:

```

1 // XXX - Use goto to break this loop.
2 #define HASH_MAP_FOR_EACH(map, slot) \
3     if ((map)->slots) \
4         for (Single_List_Node *__n, *__i = 0; \
5              PTR_TO_UINT(__i) < (map)->num_slots; \
6              __i = INT_TO_PTR(PTR_TO_UINT(__i) + 1)) \
7             SINGLE_LIST_FOR_EACH(&(map)->slots[PTR_TO_UINT(__i)], __n) \
8             if ((slot = HASH_MAP_SLOT_OF(__n)) || !slot)

```

The `Hash_Map` data structure is used in the symbol table implementation. For more information on `Hash_Map` take a look at the appendices `src/hash_map.h` and `src/hash_map.c`.

4.5 Red Black Tree

The red black tree also makes use of inheritance and is implemented much like the red black tree inside of the Linux kernel and is described in Thomas H. Cormen ... [et al.] - Introduction To Algorithms.

The red black tree implementation is very much optimized compared to the one described in Thomas H. Cormen ... [et al.]. If you are interested in the red black tree implementation check out the `src/rb_tree.h` and `src/rb_tree.c` appendices.

4.6 Debugging Facilities

The last feature from the compiler library we are going to discuss is the debugging facilities. Two macros `DEBUG()` and `DLOG()` are defined. We feed the `DEBUG()` macro with source code for debugging purposes. For example:

```

1 #undef DEBUG_TYPE
2 #define DEBUG_TYPE my-debug-type
3 DEBUG(
4     file_print_message(stderr, S("Log this to stderr\n"));
5 );

```

Note that the `DEBUG_TYPE` macro is redefined. This is done to specify how to enable the statements inside the `DEBUG()` macro. Whenever the `DEBUG` environment variable matches the regular expression `(.*)?my-debug-type(.*?)?` and the `NDEBUG` preprocessor macro is undefined and execution reaches line 4 in the example, the `file_print_message()` function is called.

So if we want to enable multiple debug types we can export the `DEBUG` environment variable such that `DEBUG=symbol-table:parser-parse:parser-recover`.

The `DLOG()` macro is defined as follows

```

1 #define DLOG(fmt, ...) \
2     DEBUG(file_print_message(stderr, S(fmt), ## __VA_ARGS__);)

```

The `__VA_ARGS__` is a `gcc` extension and essentially allows us to pass a variable number of arguments to macros.

There is more information about the debugging facilities in the appendices `src/debug.h` and `src/debug.c`.

5 The Vitaly Grammar

Our parser extends the standard Vitaly grammar to offer some more advanced features. A simplified version of the extended Vitaly grammar is described below. Terminals are written in **bold** font and non-terminals are surrounded with angle brackets.

| | |
|-------------------------------------|---|
| $\langle start \rangle$ | $::= \langle opt-package \rangle \langle start-body \rangle$ |
| $\langle start-body \rangle$ | $::= \langle decl-stmt-list \rangle$ ϵ |
| $\langle opt-package \rangle$ | $::= \langle package \rangle$ ϵ |
| $\langle package \rangle$ | $::= \textbf{package module-const ;}$ |
| $\langle function \rangle$ | $::= \langle func-head \rangle \langle opt-decl-stmt-list \rangle \langle func-tail \rangle$ |
| $\langle func-head \rangle$ | $::= \textbf{func} \langle func-id \rangle (\langle opt-par-decl-list \rangle) : \langle void-type \rangle$ |
| $\langle func-tail \rangle$ | $::= \textbf{end} \langle func-id \rangle$ |
| $\langle func-id \rangle$ | $::= \textbf{id}$ finalize record |
| $\langle extern-function \rangle$ | $::= \textbf{extern (id) func id (} \langle opt-par-decl-list \rangle \textbf{) : } \langle void-type \rangle$ |
| $\langle opt-par-decl-list \rangle$ | $::= \langle par-decl-list \rangle$ ϵ |
| $\langle par-decl-list \rangle$ | $::= \langle par-decl-list \rangle , \langle var-type \rangle$ $\langle var-type \rangle$ |
| $\langle void-type \rangle$ | $::= \langle type \rangle$ void |
| $\langle type \rangle$ | $::= \textbf{id}$ int bool array of $\langle type \rangle$ char string $\langle record-decl \rangle$ |

| | |
|---|---|
| $\langle \text{record-decl} \rangle$ | $::= \langle \text{record-head} \rangle \{ \langle \text{opt-record-decl-list} \rangle \}$ |
| $\langle \text{record-head} \rangle$ | $::= \textbf{record of} \langle \text{opt-extend-list} \rangle$ |
| $\langle \text{opt-extend-list} \rangle$ | $::= \langle \text{extend-list} \rangle$ ϵ |
| $\langle \text{extend-list} \rangle$ | $::= \langle \text{extend-list} \rangle, \textbf{id}$ \textbf{id} |
| $\langle \text{opt-record-decl-list} \rangle$ | $::= \langle \text{record-decl-list} \rangle$ ϵ |
| $\langle \text{record-decl-list} \rangle$ | $::= \langle \text{record-decl-list} \rangle \langle \text{record-member} \rangle$ $\langle \text{record-member} \rangle$ |
| $\langle \text{record-member} \rangle$ | $::= \langle \text{function} \rangle$ $\langle \text{var-type} \rangle \langle \text{record-seperator} \rangle$ $\textbf{var} \langle \text{var-type} \rangle \langle \text{record-seperator} \rangle$ $\langle \text{type-def} \rangle \langle \text{record-seperator} \rangle$ $\langle \text{import-start} \rangle \langle \text{record-seperator} \rangle$ $\langle \text{record-seperator} \rangle$ |
| $\langle \text{record-seperator} \rangle$ | $::= ,$ $;$ |
| $\langle \text{var-decl-list} \rangle$ | $::= \langle \text{var-decl-list} \rangle, \langle \text{var-decl-type} \rangle$ $\langle \text{var-decl-type} \rangle$ |
| $\langle \text{var-decl-type} \rangle$ | $::= \langle \text{var-type} \rangle$ $\langle \text{var-type} \rangle = \langle \text{expression} \rangle$ |
| $\langle \text{var-type} \rangle$ | $::= \textbf{id} : \langle \text{type} \rangle$ |
| $\langle \text{opt-decl-stmt-list} \rangle$ | $::= \langle \text{decl-stmt-list} \rangle$ ϵ |
| $\langle \text{decl-stmt-list} \rangle$ | $::= \langle \text{decl-stmt-list} \rangle \langle \text{decl-stmt} \rangle$ $\langle \text{decl-stmt} \rangle$ |
| $\langle \text{decl-stmt} \rangle$ | $::= \langle \text{declaration} \rangle$ $\langle \text{statement} \rangle$ |
| $\langle \text{declaration} \rangle$ | $::= \langle \text{function} \rangle$ $\langle \text{extern-function} \rangle ;$ $\langle \text{type-def} \rangle ;$ $\textbf{var} \langle \text{var-decl-list} \rangle ;$ $\langle \text{var-decl-list} \rangle ;$ |
| $\langle \text{type-def} \rangle$ | $::= \textbf{type id} = \langle \text{type} \rangle$ |

| | |
|---------------------------------------|--|
| $\langle \text{statement} \rangle$ | $::=$; $\langle \text{import-start} \rangle$; allocate $\langle \text{variable} \rangle$; allocate $\langle \text{variable} \rangle$ of record (opt-exp-list) ; allocate $\langle \text{variable} \rangle$ of length $\langle \text{expression} \rangle$; delete $\langle \text{variable} \rangle$; return ; return $\langle \text{expression} \rangle$; write $\langle \text{expression} \rangle$; $\langle \text{variable} \rangle = \langle \text{expression} \rangle$; $\langle \text{expression} \rangle$; if $\langle \text{expression} \rangle$ then $\langle \text{decl-stmt} \rangle$ $\langle \text{opt-else} \rangle$ while $\langle \text{expression} \rangle$ do $\langle \text{decl-stmt} \rangle$ { $\langle \text{opt-decl-stmt-list} \rangle$ } finalize $\langle \text{decl-stmt} \rangle$ |
| $\langle \text{opt-else} \rangle$ | $::=$ else $\langle \text{decl-stmt} \rangle$ ϵ |
| $\langle \text{import-start} \rangle$ | $::=$ import module-const |
| $\langle \text{expression} \rangle$ | $::=$ $\langle \text{expression} \rangle$ $\langle \text{bin-op} \rangle$ $\langle \text{expression} \rangle$ $\langle \text{term} \rangle$ |
| $\langle \text{term} \rangle$ | $::=$ $\langle \text{variable} \rangle$ ($\langle \text{expression} \rangle$) $\langle \text{expression} \rangle$! $\langle \text{term} \rangle$ $\langle \text{type-cast} \rangle$ $\langle \text{term} \rangle$ int-const char-const string-const true false null |
| $\langle \text{variable} \rangle$ | $::=$ $\langle \text{variable} \rangle$. $\langle \text{variable-id} \rangle$ $\langle \text{variable} \rangle$ [$\langle \text{expression} \rangle$] ($\langle \text{variable} \rangle$) $\langle \text{type-cast} \rangle$ $\langle \text{variable} \rangle$ $\langle \text{record-id} \rangle$ $\langle \text{direct-record-ref} \rangle$. $\langle \text{variable-id} \rangle$ $\langle \text{direct-record-ref} \rangle$ ($\langle \text{opt-exp-list} \rangle$) |
| $\langle \text{record-id} \rangle$ | $::=$ $\langle \text{variable-id} \rangle$ record |
| $\langle \text{variable-id} \rangle$ | $::=$ id id ($\langle \text{opt-exp-list} \rangle$) |

| | |
|--|---|
| $\langle \text{direct-record-ref} \rangle$ | $::=$ record [id] $\quad \mid$ record [] |
| $\langle \text{type-cast} \rangle$ | $::=$ cast ($\langle \text{type} \rangle$) |
| $\langle \text{opt-exp-list} \rangle$ | $::=$ $\langle \text{exp-list} \rangle$ $\quad \mid$ ϵ |
| $\langle \text{exp-list} \rangle$ | $::=$ $\langle \text{exp-list} \rangle$, $\langle \text{expression} \rangle$ $\quad \mid$ $\langle \text{expression} \rangle$ |
| $\langle \text{bin-op} \rangle$ | $::=$ $\quad \mid$ && $\quad \mid$ == $\quad \mid$!= $\quad \mid$ < $\quad \mid$ > $\quad \mid$ <= $\quad \mid$ >= $\quad \mid$ + $\quad \mid$ - $\quad \mid$ * $\quad \mid$ / |

Most of the terminal symbols (the symbols written in **bold**) are literally as they appear in the grammar, i.e. they are either keywords or special symbols. However there are 5 terminals **id**, **int-const**, **char-const**, **string-const** and **module-const** which differ from this rule.

- The terminal **id** is any identifier not a keyword and recognized by the regular expression $[a-zA-Z_][a-zA-Z_0-9]^*$.
- The **int-const** is any non negative integer.
- **char-const** is a character surrounded by single quotes or an escape character, much like we know escape characters like `'\n'` from the C programming language.
- **string-const** is a string literal surrounded by double quotes. Like C strings our strings also support escape characters inside of them. Thus an example of a string literal is `"Hello\n"`.
- The **module-const** terminal symbol is one or more identifiers optionally separated by `.,` that is the regular expression:
 $([a-zA-Z_][a-zA-Z_0-9]^* \backslash . ?)^* [a-zA-Z_][a-zA-Z_0-9]^*$

Our Vitaly grammar extends the original Vitaly grammar in the following ways:

1. New primitive types: `string`, `char`.
2. Functions and variables may be declared anywhere.
3. Variables can be initialized, where they are declared (`var i:int = 10;`).

4. Support for multiple dot references (`rec.ary[0].fun(1, 2).field`).
5. Added `delete` keyword to free memory allocated by `allocate`.
6. Extensions to functions:
 - Added return type `void`.
 - Allow empty function bodies.
7. Extensions to records:
 - Record fields can also be separated by `;`.
 - Allow unlimited number of separators between record fields.
 - Allow `var` keyword before record field declarations.
 - Allow type definitions inside records.
 - Functions can be defined inside records.
 - Records can extend multiple other records (multiple inheritance).
 - Records can have empty bodies.
8. Added type cast with `cast(T) v`, where `T` is a type and `v` is a variable which is casted to type `T`.
9. Added `finalize` statement to put code into a special `finalize` section.
10. Added `import` keyword to import other vitaly source files.
11. Added `package` keyword to organize imports.
12. Added extern function declarations, ability to call C functions from `vitaly`.
13. Record function can directly reference its base record, or its own, functions using the `record[id].fun()` syntax.

The extensions in the list are discussed more thoroughly in other sections.

6 Parsing

Our parser is using `flex` and `bison` for scanning and parsing respectively. The `flex` tool reads/scans files and converts them into token streams which the `bison` tool can use for parsing.

The tools `flex` and `bison` introduce some limitations when it comes to error recovery, however with proper hacking we can tweak the tools and implement decent error recovery and error reporting mechanisms.

6.1 Scanning

The `flex` file is attached as appendix `src/parser/scanner.l`, and basically gives the parser the next token in the input whenever asked for it. The `flex` file is so simple we will refer to the appendix `src/parser/scanner.l` for more info on the scanner implementation.

6.2 Parsing

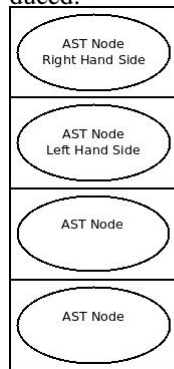
The `bison` file is in the appendix `src/parser/parser.y`.

Our goal of parsing is to generate an abstract syntax tree (AST). Nodes in the AST can have an arbitrary number of children and the AST nodes are often generated when grammar productions are reduced.

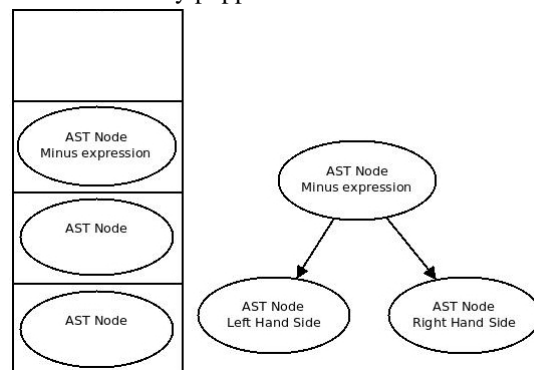
Whenever a production like $\langle expression \rangle$ is reduced an AST node is pushed on a stack.

As an example we will consider what happens when a binary expression like $\langle expression \rangle - \langle expression \rangle$ is reduced to $\langle expression \rangle$.

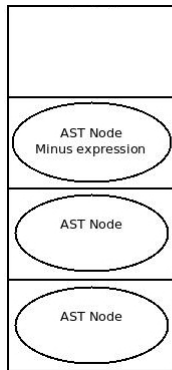
We know there are two AST nodes on top of the stack. The first AST node was pushed on the stack when the expression on the left hand side was reduced and the other AST node was pushed on the stack when the right hand side expression was reduced:



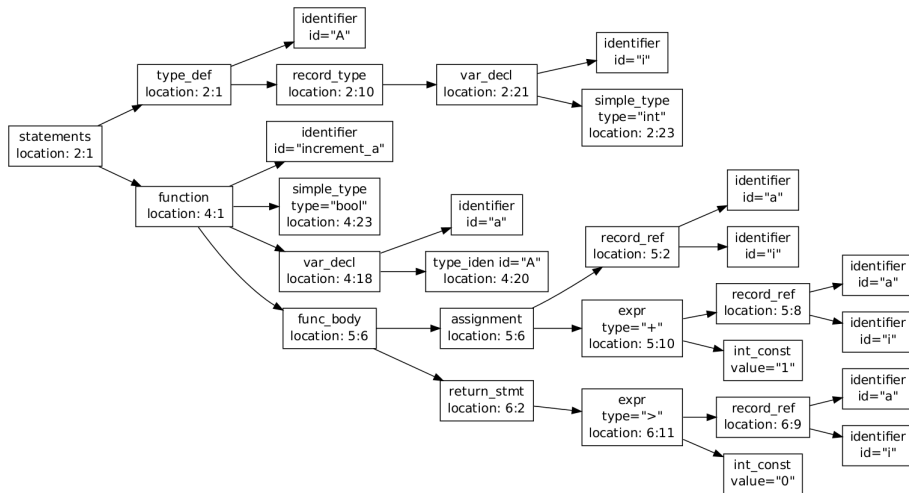
Thus we can pop these two AST nodes from the stack and generate a new AST node with the recently popped AST nodes as children:



At last we push the minus expression node back onto the stack:



After parsing a simple program the AST might look something like:



Which is an AST generated by the compiler with the `-dump=parse-tree-graph` option enabled. As we can see the AST nodes contain information about the parsed program. For instance the function AST node contains a function identifier, a return type, a list of parameters, and a function body.

One thing common to all AST nodes is that they have a location, that is a line number and a column number. We use `bison`'s semantic values to keep track of token locations and string representations. In most cases we are relying on `bison`'s default action for setting the semantic value of a symbol which is:

```
1 $$ = $1 // Save semantic value of first symbol in resulting symbol $$
```

We are using a number of `Vector` data structures as AST node stacks. To understand how it's working we will consider what happens when recognizing an if-statement. The slightly simplified `bison` code looks as follows:

```
1 statement
2   : TKN_IF_KEY expression TKN_THEN_KEY {
3     push_vector_stack(&decl_stmt_stack);
4   } decl_stmt {
5     Vector *statements = pop_vector_stack(&decl_stmt_stack);
6     Ast_Node *s = GET_STMT_LIST(AST_STMT_LIST, statements);
7     push_ast_node(&decl_stmt_stack, s);
8     PARSE_BINARY_NODE(AST_IF_STMT, $1.lineno, $1.startcol);
```

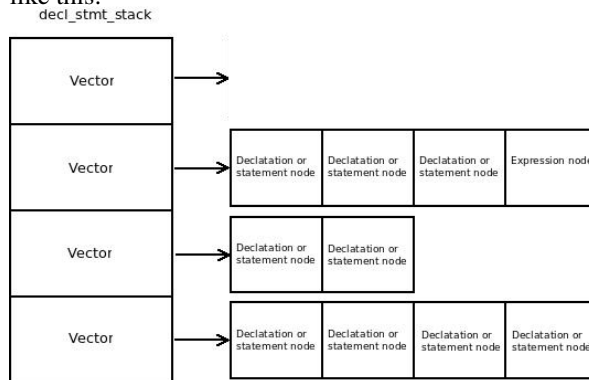
```

9      }
10     // Other production rules

```

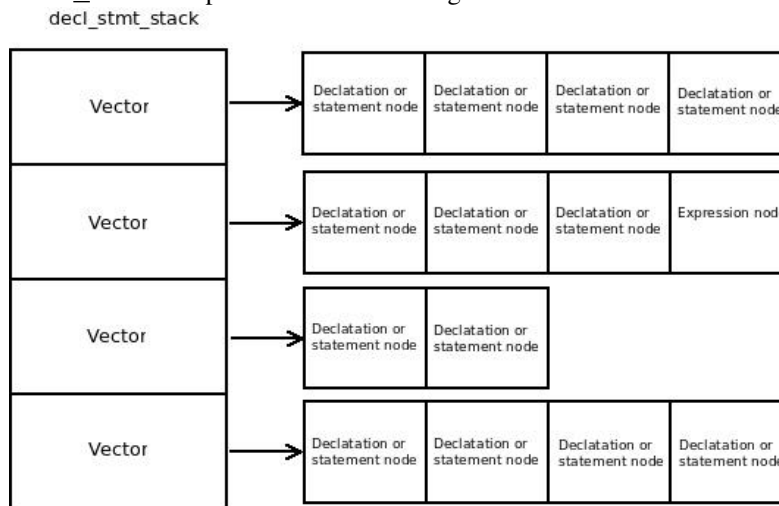
The `decl_stmt_stack` is a stack of vectors and when we have recognized the **if** keyword, an *expression* and the **then** keyword we will (in the mid-rule action) push a new `Vector` for declaration and statement AST nodes on top of the `decl_stmt_stack`. Whenever we recognize a declaration or statement we allocate an AST node for it and append that AST node to the end of the `Vector` on top of the `decl_stmt_stack`.

So right before starting to recognize `decl_stmt` the picture looks something like this:



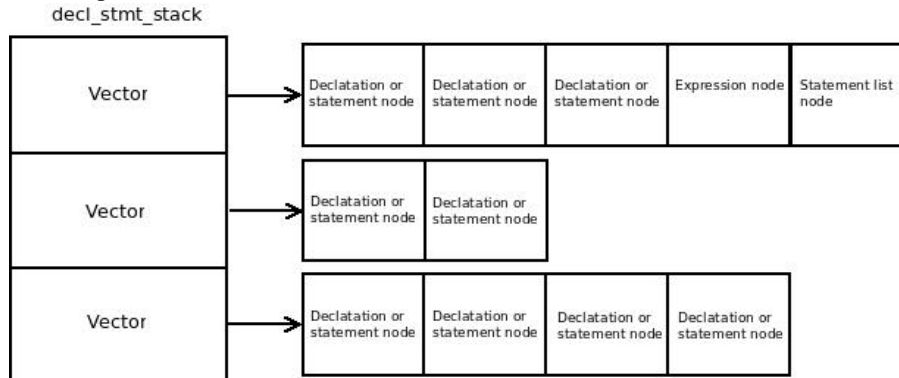
Where the `Vector` on the top is also the `Vector` on top of the `decl_stmt_stack`.

As we know one of the `decl_stmt` productions is involving a list of declarations and statements, so we don't know how many statements or declarations we will append to the `Vector` on top of the `decl_stmt_stack` before reducing `decl_stmt`. Anyway, right after reducing `decl_stmt` the picture looks something like this:

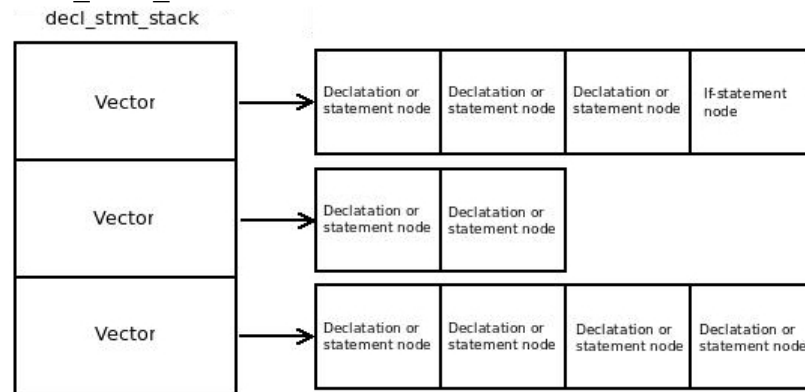


Now we (in the end-of-rule action) pop the `Vector` on top of the `decl_stmt_stack` and allocate a new statement list AST node which we append to the `Vector` which is now on top of the `decl_stmt_stack`. Note that the `Vector` which was popped from the `decl_stmt_stack` is passed to the statement list AST node.

The picture now looks as follows:



The last thing we do (with the `PARSER_BINARY_NODE()` macro) is to pop the statement list node and expression node from the `Vector` on top of the `decl_stmt_stack` and append an if-statement node instead. After the `<statement>` has been reduced the `decl_stmt_stack` is as shown below:



This example should serve as a good introduction to how the parser algorithm is implemented.

The tool `bison` is great as long as we don't care about recovering from syntax errors or supplying the user with good error messages. We certainly do care about this, thus we had to find some reasonable way to recover from errors.

We are relying on `bison`'s error recovery mechanism in a pretty limited fashion in our implementation. An example of its use is:

```

1 decl_stmt
2   : declaration {
3     // ...
4   }
5   | statement {
6     // ...
7   }
8   | errors %prec decl_stmt_prec {
9     RECOVER_EXPECT(decl_stmt_recover, $errors.lineno,
10      $errors.startcol);
11   }
12 ;

```

When an unexpected token comes in the middle of a statement or declaration we catch the error with the `errors` non-terminal. But after catching the error we take care of recovering ourselves.

The `RECOVER_EXPECT()` macro is responsible for reporting the syntax error and calling the `decl_stmt_recover()` function. The `decl_stmt_recover()` function eats up tokens from `flex` until it discovers a consistent state such that it can give back control to `bison` and let `bison` do what it's good at, namely parsing syntactically correct code.

By defining

```
1 %define parse.lac full
2 %error-verbose
```

we tell `bison` to supply more verbose error messages. However we don't use `bison`'s error messages. We process the messages to extract the information we need and supply our own error messages. Although it isn't perfect, it does result in better error messages to the user.

6.3 Abstract Syntax Tree

Once we have generated the abstract syntax tree (AST) from the `bison` file `src/parser/parser.y` we want to be able to traverse the tree, e.g. at some point we want to generate a symbol table and do type checking given the AST. In this section we will discuss how to implement such an AST traversal, which is called an AST pass in this project, in particular we will talk about how the `src/ast/ast_visitor_print.c` file implements a pretty printer for the AST using an AST pass.

The file `src/ast/ast_visitor.h` contains the `Ast_Visitor` struct which we can think of as an abstract super class supplying the interface that any AST visitor base struct must implement.

Implementing an AST visitor does take some time, the `Ast_Visitor` is declaring 55 abstract methods which must be implemented by any AST visitor base struct.

To implement an AST pass we must first declare a struct which inherits the `Ast_Visitor`. This is done using the `AST_VISITOR_STRUCT_BEGIN()` and `AST_VISITOR_STRUCT_END()` macro pair. The AST pretty printer struct is called `Ast_Visitor_Print` and declared as follows:

```
1 AST_VISITOR_STRUCT_BEGIN(Ast_Visitor_Print)
2     Uns indentation;
3     FILE *output_file;
4 AST_VISITOR_STRUCT_END(Ast_Visitor_Print)
```

Now we must implement the interface specified by the `Ast_Visitor`, that means we must implement 55 methods. If we don't do so the code will not even compile. The benefit of this is that we make sure not to forget to implement some method which is expected to exist. The `src/ast/ast_visitor.h` file defines the macro pair `ASTVF_BEGIN()` and `ASTVF_END` we need in order to implement the "abstract" methods declared in the `Ast_Visitor` struct. An example of such a method implementation follows:

```
1 ASTVF_BEGIN(AST_EXPR_FUNC_CALL, Ast_Visitor_Print, v, Ast_Expr_Func_Call, n)
2     PRINT_OPEN(v->output_file, "func_call line=\"%U\" column=\"%U\"",
3         v->indentation,
4         n->ast_node.location.line,
5         n->ast_node.location.column);
6
7     v->indentation += INDENT_COUNT;
8
```

```

9     n->identifier->accept_visitor(n->identifier, AST_VISITOR_OF(v));
10
11     Ast_Node *arg;
12     Vector *vargs = n->arguments;
13     VECTOR_FOR_EACH_ENTRY(vargs, arg)
14         arg->accept_visitor(arg, AST_VISITOR_OF(v));
15
16     v->indentation -= INDENT_COUNT;
17
18     PRINT_CLOSE(v->output_file, "func_call", v->indentation);
19 ASTVF_END

```

In this example we are defining the `Ast_Visitor_Print` method which implements the functionality for pretty printing function call AST nodes (`Ast_Expr_Func_Call`).

The first thing the method does when called is to print an opening `func_call` tag with attributes `line` and `column`.

Later the method is responsible for notifying its AST node children that `Ast_Visitor_Print` wants to visit them. This is done by calling the children node's `accept_visitor()` method. In the above example the `accept_visitor()` methods of the function identifier and function argument nodes are called.

The last thing the method in the example does is to print a closing `func_call` tag.

The AST node's `accept_visitor()` method is responsible for calling the correct `Ast_Visitor` method based on the AST node's type. For example the function call node's `accept_visitor()` method is responsible for calling the function call node `Ast_Visitor` method. The function call AST node `accept_visitor()` method is cryptically defined as follows:

```

1 AST_NODE_ACCEPT_VISITOR_DEF(AST_EXPR_FUNC_CALL);

```

If you are interested in the details how the things in this section is even possible, you should note that all AST node types are given a unique name which is used by macros to generate the correct function and method names. You probably want to take a look at the `src/ast/ast.h`, `src/ast/ast.c`, `src/ast/ast_visitor.h` and `src/ast/ast_visitor_print.c` files to fully understand the magic behind the `Ast` and `Ast_Visitor` implementation.

Given the `inc.vit` file:

```

1 type A = record of {i:int};
2 func increment_a(a:A):bool
3     a.i = a.i + 1;
4     return a.i > 0;
5 end increment_a

```

we can compile the file and get the `Ast_Visitor_Print` output when we specify the `-dump=parse-tree` option:

```

1 vitality --dump=parse-tree inc.vit

```

This generates a file named `inc.vit.vitaly.parse-tree`, containing an xml representation of the parse tree for `inc.vit`. The xml representation looks as follows:

```

1 <statements line="1" column="1">
2   <type_def line="1" column="1">
3     <identifier id="A" type="record" line="1" column="6" />

```

```

4      <record_type line="1" column="10">
5          <var_decl line="1" column="21">
6              <identifier id="i" type="int" line="1" column="21" />
7              <simple_type type="int" line="1" column="23" />
8          </var_decl>
9      </record_type>
10  </type_def>
11  <function line="2" column="1">
12      <identifier id="increment_a" type="bool" line="2" column="6" />
13      <simple_type type="bool" line="2" column="23" />
14      <var_decl line="2" column="18">
15          <identifier id="a" type="record" line="2" column="18" />
16          <type_id id="A" line="2" column="20" />
17      </var_decl>
18      <func_body line="3" column="6">
19          <assignment line="3" column="6">
20              <record_ref type="int" line="3" column="2">
21                  <identifier id="a" type="record" line="3" column="2" />
22                  <identifier id="i" type="int" line="3" column="4" />
23              </record_ref>
24              <expr_plus type="int" line="3" column="10">
25                  <record_ref type="int" line="3" column="8">
26                      <identifier id="a" type="record" line="3" column="8" />
27                      <identifier id="i" type="int" line="3" column="10" />
28                  </record_ref>
29                  <int_const value="1" line="3" column="14" />
30              </expr_plus>
31          </assignment>
32          <return_stmt line="4" column="2">
33              <expr_gt type="bool" line="4" column="11">
34                  <record_ref type="int" line="4" column="9">
35                      <identifier id="a" type="record" line="4" column="9" />
36                      <identifier id="i" type="int" line="4" column="11" />
37                  </record_ref>
38                  <int_const value="0" line="4" column="15" />
39              </expr_gt>
40          </return_stmt>
41      </func_body>
42  </function>
43 </statements>

```

6.4 Error Recovery

Most of the testing of the parser has been to verify that we were outputting an acceptable number of error messages, with correct line and column numbers. The parser was tested with Vitaly source files containing errors much more frequently than it was tested with correct Vitaly source files. The test file `test_programs/errors.vit` is shown below:

```

1  # Error recovery test
2  # Working example
3  type A = record of record of {i:int}, B {s:string};
4  type B = record of {c:char};
5  func equals(a:A, b:B):bool
6      var ret:bool = true;
7      if a != cast(A) b then
8          ret = false;
9      return ret;
10 end equals
11
12 type T1 = record of A B {};
13 type T2 = record of {
14     a a;
15     func foo()
16     end foo;
17 };
18 type T3 = record of {};
19 func foo(r:record of):int end foo
20 func foo(i i, b b):bool end foo

```



```
21 func foo(@):bool end foo
22 func foo():int
23     func foo():record of A {var a:int, bbool};
24         return 0 0
25     end foo
26     type R = record f {};
27 end foo
```

Feeding our compiler with that file gives the following output:

```
1 errors.vit:3:20: (error) unexpected 'record', expected '{' or identifier
2 errors.vit:5:15: (error) unable to resolve type 'A'
3 errors.vit:7:15: (error) unable to resolve type 'A'
4 errors.vit:12:23: (error) unexpected 'B', expected ',' or '{'
5 errors.vit:14:4: (error) unexpected 'a'
6 errors.vit:16:2: (error) unexpected 'end', expected ':'
7 errors.vit:18:21: (error) unexpected '}', expected '{' or identifier
8 errors.vit:19:21: (error) unexpected ')'
9 errors.vit:20:12: (error) unexpected 'i'
10 errors.vit:21:10: (error) unexpected '@'
11 errors.vit:23:23: (error) unable to resolve type 'A'
12 errors.vit:23:42: (error) unexpected '}'
13 errors.vit:24:12: (error) unexpected '0'
14 errors.vit:26:18: (error) unexpected 'f', expected 'of'
15 vitality: 14 errors
```

As we see, in this example program, the error recovery is alright. The error messages could have been better, however they are precise about the error locations.

7 Symbol Table

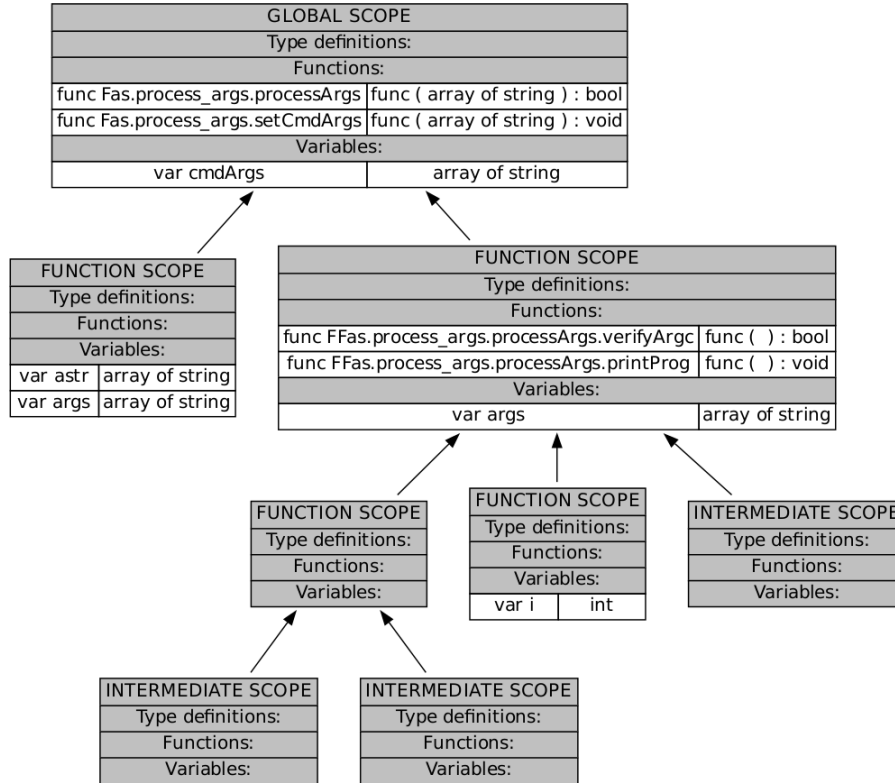
Given a function name, a type name or a variable name we use the symbol table to lookup its type.

One AST pass, implemented in `src/ast/ast_visitor_symbol_table.c`, is dedicated to generating the symbol table.

Invoking vitality with the `-dump=symbol-table-graph` option and the `process_args.vit` input file:

```
1 cmdArgs:array of string;
2 func setCmdArgs(args:array of string):void
3     astr:array of string = args;
4     cmdArgs = astr;
5 end setCmdArgs
6
7 func processArgs(args:array of string):bool
8     func verifyArgc():bool
9         if |args| > 1 then
10             return true;
11         else
12             return false;
13         end verifyArgc
14
15     func printProg():void
16         i:int = 0;
17         write args[i];
18     end printProg
19
20     if !verifyArgc() then
21         return false;
22
23     printProg();
24     setCmdArgs(args);
25     # Do other stuff ...
26     return true;
27 end processArgs
```

produces the PDF file `process_args.vit.vitaly.symbol-table.pdf` containing a graphical representation of the symbol table:



This graph is a good illustration of how the symbol table works.

The nodes in the graph are referred to as symbol table nodes. Except from the root node, each symbol table node contains a reference to its parent symbol table node.

Entering the body of a function or an if-statement opens a new scope and a new symbol table node is inserted in the graph.

With this symbol table structure we make sure the `processArgs()` function cannot access variables, type definitions or functions declared in the `setCmdArgs()` function, and vice versa. The same thing applies to the intermediate scopes from the if-else-statement in function `verifyArgc()`, they cannot access identifiers declared in the `printProg()` or `setCmdArgs()` functions, however they can access identifiers declared in the global scope or the `processArgs()` function for example.

You may have noticed some strange identifiers in the graph, e.g. the `processArgs()` function is inserted in the symbol table node as `Fas.process_args.processArgs`.

This is so because we are allowing function overloading. It is called name mangling, and is discussed in the next section.

Once the declared identifiers are inserted in the symbol table the whole symbol table is resolved. This is done in order to determine the type of variables, functions and type definitions.

After the identifiers in the symbol table are resolved the record types in the symbol table are finalized. For example, records are allowed to inherit multiple other records

thus the symbol table is responsible for inserting record fields from base records into the derived record's symbol table node.

7.1 Name Mangling

Functions, variables and records are given unique names based on where they are declared, e.g. such that the user can declare a record with the same name in different vitaly source files and different scopes.

Besides location of declaration, the unique name given to a function is also determined by parameter types.

Consider the vitaly source file `module.vit`:

```

1 package path;
2
3 type Sum = record of {
4     func add(lhs:int, rhs:int):int
5         return lhs + rhs;
6     end add
7 };
8
9 func add(s:Sum, lhs:int, rhs:int):int
10     return s.add(lhs, rhs);
11 end add
12
13 func add(s:Sum, lhs:int, rhs:bool):int
14     return add(s, lhs, cast(int) rhs);
15 end add
16
17 globalStr:string;
```

Initially the unique name `.path.module` is generated based on package name and file name.

Entering the type definition of `Sum` a `T`, for (t)ype definition, is prepended, an `R`, for (r)ecord, is prepended and the record name `Sum` is appended to the unique name: `RT.path.module.Sum`

This unique name is used when generating the virtual method table for records declared with type `Sum`, we will discuss virtual method tables in a later section.

Next entering the `add` function inside the record `Fii` is prepended to the unique name, `Fii` stands for (f)unction taking two (i)nt arguments. And the function name is appended to the unique name: `FiiRT.path.module.Sum.add`, which is the unique name given to the function.

Once we have left the `Sum` type definition the current unique name is reset to what it was before entering the type definition: `.path.module`

Entering the first `add` function in the global scope prepends `FSumii` to the unique name, standing for (f)unction taking type `Sum` (`Sum`) and two (i)nt arguments. And the function name `add` is appended: `FSumii.path.module.add`, which is the unique name for that function.

The last `add` function gets the name: `FSumib.path.module.add`

The only difference between this and the previous name is a `b` instead of an `i` because this last definition of the `add` function takes a (b)ool instead of an int as the third argument.

The last line defines a global variable which gets the unique name:

`V.path.module.globalStr`, the `V` is prepended because it is a (v)ariable.

Variables and functions defined in files with other names or other package names are always given different unique names because the initial unique name will be different.

Thus the user can define variables and functions in one file without having to consider whether the names will clash with another variable or function in another file.

As we saw, the parameter types was also used when generating unique names to functions to allow function overloading. The type checker is responsible for determining which overloaded function is best suited to function call arguments. We discuss the type checker in another section.

On line 14 in the example, the type checker will decide that the `add` function taking type `Sum`, `int` and `int` arguments is called (`FSumii.path.module.add`), since that function matches the given arguments better than the version taking `Sum`, `int` and `bool` arguments (`FSumib.path.module.add`).

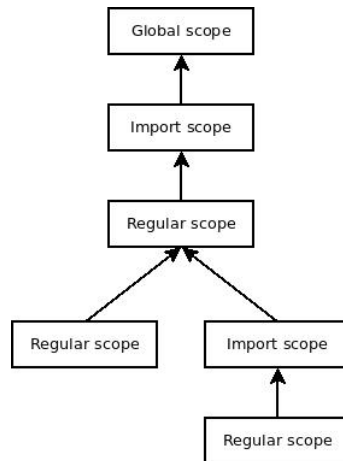
8 Imports

To extend the usability of the vitality programming language a feature to import symbols from other sources have been implemented. This enables one source file to access variables, types and functions defined in the global scope of other modules.

Throughout this section modules refer to objects which can be imported such as other source files, and viti files. The viti file type will be discussed in a later section. As an example one could import the module `mod` by `"import mod; "`.

As mentioned in the introduction to the vitality programming language, packages can be used to organize source files just like packages in Java. When a source file specifies a package, the file must be located within a corresponding directory structure. Hence a file specifying the package `"package subpack.pack; "` must be located in a directory `*/subpack/pack.`

The actual importing of symbols happens by introducing a new symbol scope in which the symbol are inserted. This import scope will be placed in the scope hierarchy as indicated below:



In the figure above `Regular scope` means any scope except scopes introduced by imports. Putting the imported symbols in a parent scope to the importing scope makes sure that local symbols will have higher priority. This means that when looking up a symbol, a symbol declared in the local scope will be found before any symbol with the

same name that might be imported, since these are placed in a parent scope.

If a scope imports multiple modules, all imported symbols will be placed in a single parent import scope. Thus all imported symbols will have equal priority, regardless of the order of the import statements.

When importing a module the types of the import should be fully resolved, even if the types are defined by modules not directly imported. See the example in the introduction to the *vitaly* programming language on accessing indirect type fields. To understand what indirect use of a type means, consider a module `car` which defines a type `Car`. This type extends another type `Vehicle` which in turn is a type imported from the module `vehicle`. Now assume that `Vehicle` has a function `drive()`. Then by importing the module `car` one is able to call `c.drive()` on a variable `c` of type `Car`, even though this function is only defined in the module `vehicle` which is not imported. Since `vehicle` is not directly imported it is not possible to use the type `Vehicle` directly (such as in casts or variable declarations), unless the module `vehicle` is explicitly imported.

This means we can access fields and functions of a type not directly imported, but only through a type that is indeed directly imported. Notice how this does not make symbols directly usable if they are not directly imported.

Handling imports consists of two parallel actions. One to handle statements, expressions and references throughout the AST and another to handle the import statements themselves.

The first part will run through the AST and translate variable names, functions identifiers and types to their real names. For example if a variable `a` is used somewhere in the AST, we must identify whether the `a` refers to a local variable or `a` refers to a variable imported from another module. If it is an imported variable `a` should be replaced by the unique name of the imported variable. The same applies to any type definition which might contain type identifiers of types not locally defined. This translation is done with a special lookup, which first tries to look up the symbol in the local current local scope. If the symbol is found locally nothing will be done. Otherwise the set of modules imported in the local symbol table node will be iterated to lookup the symbol in each module. If the symbol is still not found this process iterates towards the root of the symbol table.

Whenever an imported module is needed, by means of accessing its symbols, a hash map of scanned and parsed modules will be searched for the module. If the module is not yet scanned and parsed this will be done and the module will be added to the hash map. Thus any module which might be needed for the compilation will only be scanned and parsed once.

When searching the file system for an imported module, the compiler will first search the current package for the module that is the directory corresponding to the package of the file specifying the import statement. If the module is not found the search will continue from the project root and finally the include paths specified by the `-I` command line argument. This means that when working within a package, one does not have to prepend the current package to the import statements, if the modules are located within this package.

Note that the project root is determined by the package name of the file which the compilation was started. Thus if a file say `*/dir/pack1/pack2/file1.vit`

specifies a package such as:

```
1 package pack1.pack2;
2
3 ...
```

then the project root will be `*/dir`.

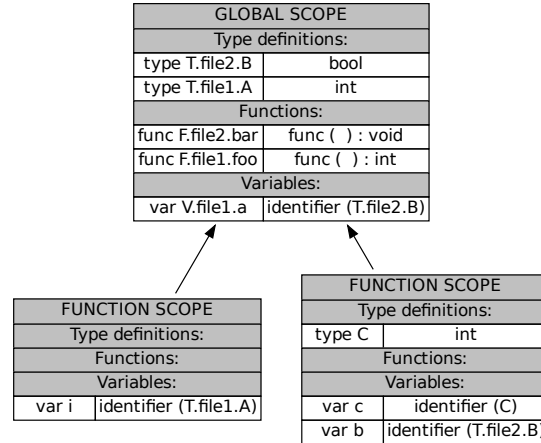
8.1 The Merge Table

The scanned and parsed modules will be contained in a struct containing a copy of the symbol table generated from the AST of the module. This table is used to create a special merge table, which is a single symbol table containing all direct and indirect imported modules. The symbol types of the copied module symbol table will be updated prior to being merged into the merge table. The symbol type update changes type identifiers, referring to types defined in the global scope or in other modules to the unique name of the symbols. When the symbol types have been updated, the symbols will be copied into the merge table, using only their unique names as identifiers. This ensures the symbols inserted are all still valid in terms of their types and no name collisions between symbols can occur due to the uniqueness of the symbol names.

After creating a combined merge table with all modules needed to resolve all imported types, the merge table is resolved just like a normal symbol table would be resolved. Then the symbols of the directly imported modules can be copied into the relevant import scopes of the file currently being compiled. Notice also that it can be necessary to add to and resolve this table multiple times during compilation. Consider the following simple example:

| file1.vit | file2.vit: | |
|---------------------------------|--------------------------------|---|
| 1 import file2; | import file1; | 1 |
| 2 | | 2 |
| 3 type A = int ; | type B = bool ; | 3 |
| 4 | | 4 |
| 5 func foo(): int | func bar(): void | 5 |
| 6 i:A; | type C = int ; | 6 |
| 7 return 0; | b:B; | 7 |
| 8 end foo | c:C; | 8 |
| 9 | end bar | 9 |
| 10 a:B; | | |

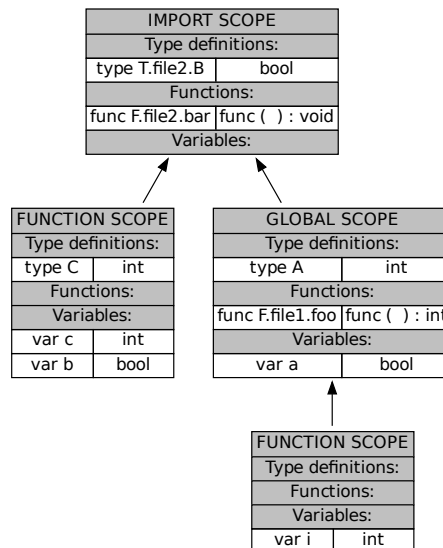
The unresolved merge table when making the import pass for `file1` is seen below:



Both `file1` and `file2` are included in the merge table, since `file1` needs `file2` to be in the merge table which in turn needs `file1` to be in the merge table. Notice that the type identifiers for `a`, `b` and `i` have been updated to use unique identifiers as opposed to the variable `c`. This is due to the fact that the type `C` is not defined in a global scope or in an import. Also notice that the name of the symbols in the global scope are using their unique names.

Because the merge table is resolved before copying the needed symbols to import scopes we ensure that we can use types indirectly.

The final symbol table for `file1` after the import phase is seen below:



Notice how an import scope containing the symbols of `file2` is inserted as parent

scope to the global scope, and that the type of `a` is correctly resolved to `bool`.

8.2 Nested imports

As mentioned in the introduction to the vitality programming language, we allow nested imports, meaning that import statements can be placed in any scope. This is handled in almost the same way as the example above only differing in where to place the import scope. Thus no more complexity is added towards implementing this feature.

8.3 Import collisions

So far we have not considered what happens if multiple imports specify variables, types or functions with the same name. If only a single import defines the symbol the unique name can easily be found. However if multiple imported modules specify symbols of the same type with the same name either of the following three cases happen:

- The conflicting symbols are variables:
In this case multiple definitions are not allowed and an error is reported.
- The conflicting symbols are functions:
Since function overloading is allowed, this is valid as long as the functions have different parameter lists otherwise the type checker will find an error because of equally matching functions. See the section on type checking.
- The conflicting symbols are type definitions:
To handle import collisions of type definitions, additional type definitions are inserted. That is if a type `A` is equal to the imported type `B` which is imported from both `file1` and `file2`, then the type definition `type A = B;` is replaced by a type definition for each import `B` is found in: `type A = T.file1.B;` and `type A = T.file2.B;`. If conflicting types for `B` exist errors will be reported when the symbol table is resolved.

We do allow all three cases without any action if the symbol is not used. That is we can import two modules both declaring a global variable with the same name since the unique name ensures no collision by default, there is no problem in allowing this. If it is used errors will be reported as described above.

8.4 Viti, Vitaly Interface Files

To improve import parsing performance and provide a way to generate interfaces for libraries the viti file format has been designed. This is essentially a valid vit file, without function bodies and types and symbols not reachable from the global scope. Consider the following example from the vitality standard library:

| std/sort.vit | | std/sort.viti: | |
|--|--|--|---|
| 1 package std; | | 1 package std; | 1 |
| 2 | | import std.object; | 2 |
| 3 import object; | | import std.indexable; | 3 |
| 4 import indexable; | | import std.comparator; | 4 |
| 5 import comparator; | | | 5 |
| 6 | | func sort(ary:indexable, c:comparator): | 6 |
| 7 func sort(ary:Indexable, c:Comparator): | | void | 6 |
| void | | end sort | 7 |
| 8 | | | |
| 9 | | | |
| 10 | | | |
| 11 func quicksort(f:int, l:int):void | | | |
| 12 if (f < l) then { | | | |
| 13 q:int = partition(f, l); | | | |
| 14 quicksort(f, q - 1); | | | |
| 15 quicksort(q + 1, l); | | | |
| 16 } | | | |
| 17 end quicksort | | | |
| 18 | | | |
| 19 quicksort(0, ary.size() - 1); | | | |
| 20 end sort | | | |

It is clear how all implementation details of the vit file is omitted, but all the information needed to do type checking, imports and translation of symbol names is still preserved. Thus the viti file provides an interface to the original vit file. It is also obvious how this will significantly reduce scanning/parsing time and reduce the overall size of the AST for imported modules. Notice also that all functions type definitions and variable declarations appears in the same order in the viti file as in the vit file. This ensures that unnamed records are properly named when the viti file is parsed.

As a side effect of this implementation one could also declare private variables, functions and types by a simple nesting. See the following example:

| | |
|---|--|
| 1 a:int; # This is public | |
| 2 | |
| 3 func foo():void # This is public | |
| 4 end foo | |
| 5 | |
| 6 type B = bool; # This is public | |
| 7 | |
| 8 { | |
| 9 type C = array of B; # This is private | |
| 10 | |
| 11 b:B; # This is private | |
| 12 } | |

Notice that since only the outermost scope is accessible via imports type `C` and variable `b` are kept private if the file is imported.

8.5 Recursive compilation

Another feature of the compiler is its ability to compile files recursively and link automatically. When compiling a file with the command line argument `-r` set, all imported files, both directly and indirectly imported, are compiled and linked into the executable. Thus there is no need to specify a makefile system or otherwise specify dependencies. If some viti files are used when importing, these are however not compiled, since these only provide interfaces to already compiled files. It is in this case up to the user to link the corresponding object files manually.

The feature simplifies the compilation task significantly, since only one file needs to be specified as argument to the compiler, even when building larger projects.

9 Type Checking

After the symbol table has been created and resolved the type checker, which you can find in the appendix `src/ast/ast_visitor_type_check.c`, is used to: verify that identifiers in the program are used correctly, verify that expressions evaluate into valid types, verify that arguments given to function calls are compatible with the parameters the functions was defined with, etc.

In most cases it is okay to use different types in an expression, for instance:

```
1 i:int = 300; b:bool = true;
2 write i - b;
```

By default, when compiling that program `vitaly` gives a warning saying that the `bool` `b` is implicitly casted to an `int`, and when the compiled program is executed it outputs:

```
1 299
```

However, we cannot cast a reference type like `string`, `array` or `record` to an `int`, on the other hand casting a reference type to a `bool` is fine.

9.1 Structural Equivalence

The `vitaly` type checker compares records by structure, e.g. when deciding whether two record types are equivalent the type checker compares the record fields one by one, starting from the first field declared in the records. If all the record fields are equivalent the records are equivalent otherwise the records are distinct.

Determining whether two record structures are equivalent is basically the same problem as determining whether two deterministic finite automats are equivalent.

We now describe an efficient algorithm to do structural comparison between records.

In the description of the algorithm, a base type can be one of the following:

- `record`
- `array`
- `string`
- `char`
- `bool`
- `int`

So type `record` has base type `record` and type `array` has base type `array`, etc.

We can get the element type of an array type by indexing it with `1`. E.g. If we want the type of the elements of an array with name `ary` we get it with the syntax `ary[1]`.

We get the type of a record field in much the same way. Say we want the type of field number `5` of the record type called `rec`, we do as follows `rec[5]`.

The first thing we do is to initialize a map `cmp_results`, mapping two records to a comparison result. The possible comparison results are `true` (the records are equivalent) and `false` (the records are distinct).

The procedure `compare(lhs, rhs)` for structurally comparing two types `lhs` and `rhs` is as follows:

1. If the base type of `lhs` is different from the base type of `rhs` return `false`
2. If the base type of `lhs` is different from `record` and different from `array` return `true`
3. If the base type of `lhs` is `array` recursively call `compare(lhs[1], rhs[1])` and return the result.
4. Then the base type of `lhs` and `rhs` is `record`. If the number of fields in the records `lhs` and `rhs` are different return `false`
5. Make a lookup in `cmp_results`, if the map contains a comparison result between `lhs` and `rhs` then return that result.
6. In the map `cmp_results` insert the comparison result `true` between records `lhs` and `rhs`.
7. For each `i = 1` to the number of fields in `lhs` do
 - Recursively call `compare(lhs[i], rhs[i])`, if the result is `false` then replace the comparison result between `lhs` and `rhs` in the map `cmp_results` with `false` and return `false`
8. Then the records `lhs` and `rhs` are equivalent, keep the comparison result `true` in the map `cmp_results` and return `true`

In vitality the field names of the records are significant, however the algorithm is easily extended to take this into account.

Vitality records may contain functions, we can extend the algorithm to take functions into consideration by comparing the functions parameter types and return type.

Vitality also supports record inheritance. Extending the algorithm to take this into consideration is not a problem either, since we can compare inherited records in the same way we compared the record field types.

9.2 Overloaded Function Selection

Faced with a function call, the type checker must be able to decide which overloaded version of a function best matches the given argument types.

The algorithm uses a metric called `score` to determine how well a function matches given arguments. A `score` of 0 means that the function does not match the given arguments and a `score` of $(n + 1)^2 + 1$, where n is the number of arguments given, is the best possible `score`.

Starting with `score = 1`.

For each argument do the following:

1. If the argument and the parameter is equivalent add $n + 1$ to the score.
2. Else if the argument is a record and the parameter is a base to the argument, then it is possible to implicitly cast the argument record to the parameter record, add n to the score.
3. Else if the argument can be implicitly casted to the parameter, e.g. if the argument has type `int` and the parameter type is `bool`, continue.

4. Else return with `score = 0`, the argument and parameter is not compatible then.

When two or more overloaded versions of a function in the same scope have the same `score > 0`, then the type checker will report an error because the function call is ambiguous.

When two or more overloaded versions of a function in different scopes have the same `score > 0`, then the type checker will select the function in the closest scope.

And, obviously, when all the overloaded versions of a function have `score = 0`, the type checker will report an error, telling the arguments did not match the function parameters.

Example:

```

1  # Global scope.
2
3  type B = record of {};
4  type D = record of B {};
5
6  func f(d:D, b:B):void
7  end f
8
9  func f(b:B, d:D):void
10 end f
11
12 func f(b:bool, b:B):void
13 end f
14
15 func f(b:B):void
16 end f
17
18 func f(d:D):void
19 end f
20
21 d:D;
22
23 {
24   # Inner scope.
25
26   func f(b:B):void
27   end f
28
29   f(d);           # Call f(d:D) from global scope.
30
31   f(cast(B) d);   # Call f(b:B) from inner scope.
32
33   f(d, d);        (* Ambiguous, f(b:B, d:D) and f(d:D, b:B) from
34                     global scope matches equally well. *)
35
36   f(cast(B) d, d); # Call f(b:B, d:D) from global scope.
37
38   f(cast(bool) d, d); # Call f(b:bool, b:B) from global scope.
39 }
40
41 f(cast(B) d);     # Call f(b:B) from global scope.

```

10 Intermediate Code Generation

The intermediate code in the vitality compiler is called Abstract Intel Assembly (AIA). It uses a 3 operand instruction set with 2 source operands and one destination operand.

Consider the vitality program:

```

1  func b(p:bool):bool
2    tmp:bool = p;

```

```

3     return tmp;
4 end b
5
6 result:string;
7 if b(true) then
8     result = "yes";
9 else
10    result = "no";

```

Compiling that program with the `-dump=init-ic` option vitally dumps the initial AIA code representation of the program, a simplified version of the dump is shown below:

```

1 .section .init
2 .movz @size(1) $1 -> @size(4) %0
3 .mov @size(4) %0 -> @size(4) %1
4 .mov @size(4) %1 -> @size(4) @arg(0)
5 .call @size(4) Fb.tmp.b -> @size(1) %2
6 .cmp @size(1) %2, $0
7 .je @size(4) .IF.else.5, .IF.then.3
8 .IF.then.3:
9 .mov @size(4) $.STR.7 -> @size(4) V.tmp.result
10 .jmp @size(4) .IF.end.4
11 .IF.else.5:
12 .mov @size(4) $.STR.8 -> @size(4) V.tmp.result
13 .jmp @size(4) .IF.end.4
14 .IF.end.4:
15
16 .section .text
17 @procedure(Fb.tmp.b)
18 @param(VFb.tmp.b.p)
19 @var(VFb.tmp.b.ret)
20 Fb.tmp.b:
21 .mov @size(1) @local(VFb.tmp.b.p) -> @size(1) @local(VFb.tmp.b.ret)
22 .ret @size(1) @local(VFb.tmp.b.ret)
23
24 .section .data
25 V.tmp.result:
26 .long 0
27
28 .section .rodata
29 .STR.7:
30 .byte 121,101,115,0
31 .STR.8:
32 .byte 110,111,0

```

It is very much resembles x86-32 gas assembly. Although it's quite bizzare, AIA is a complete programming language in its own right.

Besides the 3 operand instruction set, some noticeable differences between x86-32 gas assembly and Abstract Intel Assembly (AIA) are:

- It is more high level than gas assembly, e.g. in AIA code there is no stack.
- There is an infinite number of registers `%0`, `%1`, ...
- If an instruction has a destination operand, that operand is right of the arrow `->`
- `@size` is used to specify the byte size of operands.
- Conditional jumps have 2 jump targets. The label to the left is taken on `true`, the label to the right is taken on `false`. There is a conditional jump example on line 7.
- When passing arguments to functions the first argument is moved to `@arg(0)`, the second argument moved to `@arg(1)`, etc. See line 4.

- We declare functions using `@procedure` (line 17), we declare the function parameters using `@param` (line 18), we declare local function variables using `@var` (line 19)
- Functions reference their local variables using `@local`, as we can see on line 21 and 22.

An Abstract Syntax Tree (AST) pass implemented in the file `src/ast/ast_visitor_aia.c` is used to generate intermediate AIA code representation.

10.1 Nesting Functions

AIA code is aware of nested functions, a nested function can access its parent function's local variables using `@display_ref` for example:

```

1  @procedure(outer)
2      @push_display(-1)
3      @var(outer.i)
4  outer:
5      .ret
6
7  @procedure(inner)
8      @param(.disp.0)
9  inner:
10     .mov @size(4) $0 -> @size(4) @display_ref(@local(.disp.0)::outer::outer.i)
11     .ret

```

`@push_display(-1)` is used to indicate that functions nested inside `outer` can reference local variables inside `outer`.

Function `inner` accesses the local variable `outer.i` of function `outer` using `@display_ref`.

The way this is done in AIA code might seem rather random, but there is a method behind the madness.

While Intel was designing the 80206 processor the Pascal programming language was popular, Pascal is a block structured language, i.e. Pascal is also allowing nested functions. Thus the Intel engineers studied the problem with nested functions accessing local variables of parent functions carefully.

They came up with two new instructions `enter` and `leave`. While the `leave` instruction is simple and is equivalent with the standard function epilogue:

```

1  movl %ebp, %esp
2  popl %ebp

```

The `enter` instruction is quite complex. The instruction takes two immediate constant operands, the first being the number of bytes to reserve for local variables and the second operand being the lexical nesting of the function. The `enter $L, $0` instruction is equivalent with the standard function prologue:

```

1  pushl %ebp          # Save frame pointer for caller
2  movl  %esp, %ebp    # Setup own frame pointer
3  subl  $L, %esp      # Make room for L bytes of local variables

```

The `enter $L, $1` instruction is equivalent with:

```

1  pushl %ebp          # Save frame pointer for caller
2  movl  %esp, %ebp    # Setup own frame pointer
3  pushl %ebp          # Preserve own frame pointer
4  subl  $L, %esp      # Make room for L bytes of local variables

```

The `enter $L, $2` instruction is equivalent with:

```

1  pushl %ebp      # Save frame pointer for caller
2  pushl -4(%ebp)  # Save frame pointer for caller's caller
3  leal 4(%esp), %ebp # Setup own frame pointer
4  pushl %ebp      # Preserve own frame pointer
5  subl $L, %esp   # Make room for L bytes of local variables

```

And the `enter $L, $3` instruction is equivalent with:

```

1  pushl %ebp      # Save frame pointer for caller
2  pushl -4(%ebp)  # Save frame pointer for caller's caller
3  pushl -8(%ebp)  # Save frame pointer for caller's caller's caller
4  leal 8(%esp), %ebp # Setup own frame pointer
5  pushl %ebp      # Preserve own frame pointer
6  subl $L, %esp   # Make room for L bytes of local variables

```

The idea with the `enter` instruction is good. Using the `enter` instruction a nested function can access any parent function's local variables in constant time regardless of how deep the function is nested.

The biggest problem with the `enter` instruction is that the C/C++ programming languages became more popular than Pascal shortly after Intel introduced the `enter` instruction, so Intel never bothered to optimize `enter`.

Thus, if we want the kind of prologue the `enter` instruction implements we should push the frame pointers ourselves as shown above. Pushing the frame pointers ourselves also introduces more flexibility which the vitality compiler makes use of, as we will soon see.

This rather automated method is alright since it guarantees constant time access to variables of parent functions, but what if the nested function doesn't access some parent function's variables?

The vitality compiler has an AST pass dedicated to discovering which local variables of parent functions are accessed by nested functions, to avoid preserving frame pointers when it isn't necessary. This AST pass is implemented in `src/ast/ast_visitor_dependency.c`.

Also, the vitality compiler does not setup the frame pointer `ebp`, it uses `esp` to access parameters and local variables instead.

Thus the prologue for a function without nested functions simply look as follows:

```

1  subl $L + A, %esp

```

where `L` is the number of bytes needed for local variables and `A` is the number of bytes needed by the function call taking most arguments.

For example, if our function has one `int` local variable and our function makes two function calls. One function call takes 1 `int` argument, and the other function call takes 2 `int` arguments. Then the prologue would be:

```

1  subl $4 + 2*4, %esp

```

The first 4 bytes are subtracted to make room for the local `int` variable, the other `2*4` bytes are subtracted to make room for the arguments to the function taking two `int` arguments. If the function has one or more nested functions accessing its local variables the prologue look as follows:

```

1  subl $L + A + 4, %esp
2  # Preserve own frame pointer as first parameter to the nested functions
3  movl %esp, A(%esp)

```

Consider the following vitaly code:

```

1 func a():void
2   i:int;
3   func b():void
4     j:int;
5     func c():void
6       i = 1;
7       j = 2;
8     end c
9     c();
10  end b
11  b();
12 end a

```

Here function `b` should preserve the frame pointer for `a` and preserve its own frame pointer. Which is done as follows:

```

1 subl $L + A + 8, %esp
2 # Preserve own frame pointer as first parameter to the nested function c.
3 movl %esp, A(%esp)
4 movl P(%esp), %eax
5 # Preserve frame pointer for function a as second argument to c.
6 movl %eax, A + 4(%esp)

```

Where `P` is the offset to `a`'s frame pointer which was pushed on the stack by `a`. By now, the previously discussed Abstract Intel Assembly (AIA) syntax `@push_display(-1)` probably makes more sense. It means the function should preserve its own frame pointer for a nested function.

There are more interesting details regarding how vitaly sets up frame pointers for nested functions, this was an introduction.

10.2 Records and Multiple Inheritance

Multiple inheritance is the most advanced feature of the vitaly programming language.

First we will consider a record in its simplest form:

```

1 type D = record of {
2   i:int;
3   j:int;
4 };

```

The memory layout of such a record is:

```

1 |-----|
2 |  i  |
3 |-----|
4 |  j  |
5 |-----|

```

When `D` inherits another record:

```

1 type A = record of {
2   # ...
3 };
4 type D = record of A {
5   a:int;
6   b:int;
7 };

```

then the memory layout of `D` is:


```

1 |-----|
2 |   A   |
3 |-----|
4 |   i   |
5 |-----|
6 |   j   |
7 |-----|

```

And inheriting multiple records:

```

1 type A = record of { (* ... *) };
2 type B = record of { (* ... *) };
3 type C = record of { (* ... *) };
4 type D = record of A, B, C {
5     a:int;
6     b:int;
7 };

```

Gives D the memory layout:

```

1 |-----|
2 |   A   |
3 |-----|
4 |   B   |
5 |-----|
6 |   C   |
7 |-----|
8 |   i   |
9 |-----|
10 |   j   |
11 |-----|

```

It is easy to see that we can cast a pointer `d` pointing to a record of type `D` to a record of type `A` without doing anything. We can cast `d` to a record of type `B` by adding the offset to the `B` record, and cast to `C` by adding the offset to `C`.

If we want to cast the other way, e.g. if we want to cast a pointer to a record of type `B` to `D`, we subtract the offset to `B` instead.

If we have a record with a function inside:

```

1 type R = record of {
2     i:int;
3     func f():void (* .. *) end f
4     func g():void (* .. *) end g
5     j:int;
6 };

```

Then a pointer to a table containing pointers to the functions inside the record is inserted, we refer to this table of functions as a Virtual Method Table (VMT):

```

1 R:          vmt:
2 |-----| |-----|
3 | vmt | | f |
4 |-----| |-----|
5 | i | | g |
6 |-----| |-----|
7 | j |
8 |-----|

```

When allocating the record the field `vmt` is initialized with a pointer to the record's VMT such that the record can reference its functions.

Inheriting one record with a VMT:

```

1 type A = record of {
2     func f():void (* .. *) end f

```

```

3   func g():void (* .. *) end g
4 };
5 type D = record of A {
6   i:int;
7   func h():void (* .. *) end h
8   j:int;
9 };

```

Then the memory layout for `A` is:

```

1 A:      amt:
2 |-----| |-----|
3 | amt | | A.f |
4 |-----| |-----|
5 |      | | A.g |
6 |-----| |-----|

```

where `amt` is the Virtual Method Table (VMT) for `A`. And the memory layout for `D` becomes:

```

1 D:      dmt:
2 |-----| |-----|
3 | A | | A.f |
4 |-----| |-----|
5 | i | | A.g |
6 |-----| |-----|
7 | j | | D.h |
8 |-----| |-----|

```

where `dmt` is the VMT for `D`. When allocating a record of type `D` the VMT inside `A` is initialized to point to `dmt`.

Notice that the offsets to the functions inherited from `A` are located at the same offsets in the VMT as they were in `amt`. Thus having a pointer to a record of type `D` we can simply treat it as a pointer to a record of type `A`.

Overriding a function:

```

1 type A = record of {
2   func f():void (* .. *) end f
3   func g():void (* .. *) end g
4 };
5 type D = record of A {
6   i:int;
7   func f():void (* .. *) end f
8   func h():void (* .. *) end h
9   j:int;
10 };

```

Record `A` has the same memory layout, but the memory layout for `D` becomes:

```

1 D:      dmt:
2 |-----| |-----|
3 | A | | D.f |
4 |-----| |-----|
5 | i | | A.g |
6 |-----| |-----|
7 | j | | D.h |
8 |-----| |-----|

```

Notice that `A.f` has been overridden by `D.f` in the VMT for `D` (`dmt`).

We can still treat a pointer to a record of type `D` as a pointer to a record of type `A`, when calling function `f` the version defined by `D` (`D.f`) is invoked.

Inheriting multiple records with VMTs:

```

1 type A = record of {

```

```

2   func f():void (* .. *) end f
3 };
4 type B = record of {
5   func g():void (* .. *) end g
6 };
7 type D = record of A, B {
8   func h():void (* .. *) end h
9   i:int;
10  j:int;
11 };

```

We know the memory layout of *A* and *B*. The layout of *D* becomes:

```

1 D:      dmt:
2 |-----| |-----|
3 |  A  | | A.f |
4 |-----| |-----|
5 |  B  | | D.h |
6 |-----| |-----|
7 |  i  | | B.g |
8 |-----| |-----|
9 |  j  |
10 |-----|

```

Allocating a record of type *D* we initialize the Virtual Method Table (VMT) in *A* with a pointer to *dmt* and initialize the VMT in *B* with the pointer to *dmt* + offset to *B.g*.

Overriding functions with multiple inheritance:

```

1 type A = record of {
2   func f():void (* .. *) end f
3 };
4 type B = record of {
5   func g():void (* .. *) end g
6 };
7 type D = record of A, B {
8   func f():void (* .. *) end f
9   func g():void (* .. *) end g
10  func h():void (* .. *) end h
11  i:int;
12  j:int;
13 };

```

We still know the memory layout of *A* and *B*. And the layout of *D* is:

```

1 D:      dmt:
2 |-----| |-----|
3 |  A  | | D.f |
4 |-----| |-----|
5 |  B  | | D.g |
6 |-----| |-----|
7 |  i  | | D.h |
8 |-----| |-----|
9 |  j  | | D.t |
10 |-----|

```

Still, when a record of type *D* is allocated the VMT inside *A* is initialized with a pointer to *dmt*, but now the VMT in *B* is initialized with the pointer to *dmt* + offset to *D.t*, we talk about function *D.t* later.

We can still treat a pointer to a record of type *D* as a pointer to a record of type *A*, and when function *f* is called the version of *f* defined in *D* is called.

The function *D.t* is a trampoline to the function *D.g*. The trampoline function *D.t* casts a record pointer of type *B* to a record pointer of type *D* by subtracting the offset to *B* inside *D*. Afterwards the function *D.t* jumps to the function *D.g*.

An example of an AIA trampoline function is:

```
1 @trampoline(RT.tmp.B$FRT.tmp.D.g)
2   @param(.slf)
3 RT.tmp.B$FRT.tmp.D.g:
4   .add @size(4) @local(.slf), $-4 -> @size(4) %0
5   .mov @size(4) %0 -> @size(4) @local(.slf)
6   .jmp @size(4) FRT.tmp.D.g
```

Again, we have omitted some details, but this section gives a general idea about how to implement multiple inheritance.

11 Optimization

There are two main reasons for intermediate code (IC) representation. One reason for IC is to convert it into multiple different machine code targets. The other reason is that we can design the IC such that it is easier to do optimization on it.

While designing the Abstract Intel Assembly (AIA) code representation our main goal was optimization. We wanted an IC which was easy to traverse and modify. We did not have much time, thus an important design decision with AIA was to make it easily convertible to x86-32 assembly. Considering AIA was our first attempt, ever, designing an IC representation, we are satisfied with the result.

AIA might look like linear assembly code when it's dumped to a file, but really it is not, it is a Control Flow Graph (CFG) where jump instructions represent directed edges in the graph and basic blocks are nodes in the graph.

That is why conditional jump instructions have two target labels, it is also the reason we often find jumps to labels right beneath the jump instruction like in the following AIA if-statement:

```
1   .cmp @size(4) %2, $0
2   .je @size(4) .IF.end.4, .IF.then.3
3 .IF.then.3:
4   .mov @size(4) %2 -> @size(4) V.tmp.result
5   .jmp @size(4) .IF.end.4
6 .IF.end.4:
```

In real assembly we would never insert the jump instruction on line 5, but the AIA code needs the jump as an edge in the CFG.

The files `src/aia/aia.h`, `src/aia/aia_instr.h` and `src/aia/aia_operand.h` exposes a collection of functions we use when traversing and modifying AIA code, some examples are:

- We traverse an AIA function depth first with the `aia_func_for_each_block_depth()` function.
- We iterate the instructions inside a basic block using the `AIA_BLOCK_FOR_EACH_INSTRUCTION()` macro.
- We iterate through an instruction's operands with the `AIA_INSTR_FOR_EACH_OPERAND()` macro.
- We replace an AIA instruction with another using the `aia_instr_replace()` function.

11.1 Constant Propagation

The combined constant propagation and constant folding pass implemented in `src/aia/aia_const_prop.c` is used to simplify constant expressions and substitute variable references with constants when possible.

Constant propagation is one of the simplest AIA passes. The pass imaginary executes the code, simplifying constant expressions while keeping track of known values of variables. Consider the vitaly program:

```

1  i:int;
2  result:int = 1;
3
4  i = 10;
5  if i >= 10 then {
6      tmp:bool = i != 10;
7      i = i + 30;
8      if tmp then {
9          # Dead code
10         i = i - 123456;
11         result = i;
12     } else {
13         if !tmp then
14             i = i + 2 * result;
15     }
16     result = result * i;
17 } else {
18     # Dead code
19     i = 0;
20     result = 0;
21 }
22
23 write result; # Print 42

```

Before the first constant propagation pass the simplified AIA code looks as follows:

```

1  .section .init
2      .mov @size(4) $1 -> @size(4) V.cp18.result
3      .mov @size(4) $10 -> @size(4) V.cp18.i
4      .cmp @size(4) V.cp18.i, $10
5      .setge -> @size(1) %0
6      .cmp @size(1) %0, $0
7      .je @size(4) .IF.else.4, .IF.then.2
8  .IF.then.2:
9      .cmp @size(4) V.cp18.i, $10
10     .setne -> @size(1) %1
11     .mov @size(1) %1 -> @size(1) V.2.1.cp18.tmp
12     .add @size(4) V.cp18.i, $30 -> @size(4) %2
13     .mov @size(4) %2 -> @size(4) V.cp18.i
14     .cmp @size(1) V.2.1.cp18.tmp, $0
15     .je @size(4) .IF.else.7, .IF.then.5
16  .IF.then.5:
17     .sub @size(4) V.cp18.i, $123456 -> @size(4) %3
18     .mov @size(4) %3 -> @size(4) V.cp18.i
19     .mov @size(4) V.cp18.i -> @size(4) V.cp18.result
20     .jmp @size(4) .IF.end.6
21  .IF.else.7:
22     .cmp @size(1) V.2.1.cp18.tmp, $0
23     .sete -> @size(1) %4
24     .cmp @size(1) %4, $0
25     .je @size(4) .IF.end.9, .IF.then.8
26  .IF.then.8:
27     .imul @size(4) $2, V.cp18.result -> @size(4) %5
28     .add @size(4) V.cp18.i, %5 -> @size(4) %6
29     .mov @size(4) %6 -> @size(4) V.cp18.i
30     .jmp @size(4) .IF.end.9
31  .IF.end.9:
32     .jmp @size(4) .IF.end.6
33  .IF.end.6:
34     .imul @size(4) V.cp18.result, V.cp18.i -> @size(4) %7
35     .mov @size(4) %7 -> @size(4) V.cp18.result

```

```

36     .jmp @size(4) .IF.end.3
37 .IF.else.4:
38     .mov @size(4) $0 -> @size(4) V.cp18.i
39     .mov @size(4) $0 -> @size(4) V.cp18.result
40     .jmp @size(4) .IF.end.3
41 .IF.end.3:
42     .mov @size(4) V.cp18.result -> @size(4) @arg(0)
43     .call @size(4) _Vit_writeln
44
45 .section .data
46 .align 4
47 V.cp18.i:
48     .long 0
49 .align 4
50 V.cp18.result:
51     .long 0
52 V.2.1.cp18.tmp:
53     .byte 0

```

And after the last constant propagation pass the simplified AIA code is:

```

1  .section .init
2  .jmp @size(4) .IF.then.2
3  .IF.then.2:
4  .jmp @size(4) .IF.else.7
5  .IF.else.7:
6  .jmp @size(4) .IF.then.8
7  .IF.then.8:
8  .jmp @size(4) .IF.end.9
9  .IF.end.9:
10 .jmp @size(4) .IF.end.6
11 .IF.end.6:
12 .jmp @size(4) .IF.end.3
13 .IF.end.3:
14     .mov @size(4) $42 -> @size(4) @arg(0)
15     .call @size(4) _Vit_writeln
16
17 .section .data
18 .SECT.1:
19 .align 4
20 V.cp18.i:
21     .long 42
22 .align 4
23 V.cp18.result:
24     .long 42
25 V.2.1.cp18.tmp:
26     .byte 0

```

For completeness, here is the x86-32 assembly code the compiler produces:

```

1  .section .init
2  .SECT.0:
3      addl $-4, %esp
4  .IF.then.2:
5  .IF.else.7:
6  .IF.then.8:
7  .IF.end.9:
8  .IF.end.6:
9  .IF.end.3:
10 # line 24
11     movl $42, (%esp)
12     call _Vit_writeln
13     addl $4, %esp
14
15 .section .data
16 .SECT.1:
17 # line 1
18 .globl V.cp18.i
19 .type V.cp18.i, @object
20 .size V.cp18.i, 4
21 .align 4
22 V.cp18.i:

```

```

23     .long 42
24 # line 2
25 .globl V.cp18.result
26 .type V.cp18.result, @object
27 .size V.cp18.result, 4
28 .align 4
29 V.cp18.result:
30     .long 42
31 # line 6
32 .type V.2.1.cp18.tmp, @object
33 .size V.2.1.cp18.tmp, 1
34 V.2.1.cp18.tmp:
35     .byte 0

```

11.2 Instruction Elimination

The instruction elimination pass implemented in `src/aia/aia_instr_elim.c` is the compiler's dead code elimination pass. It uses the function `aia_instr_live_sets()` implemented in `src/aia/aia_instr_live_sets.c` to get a liveness analysis.

The idea of the pass is to find out whether an instruction's destination operand is live out of the instruction. If not, then it is often safe to remove (eliminate) the instruction.

Note that the `aia_instr_live_sets()` function also is used by other AIA passes, the `src/aia_warn_undefined.c` file is using the liveness analysis to give warnings about variables which might be uninitialized before use.

11.3 Register Variables

The register variables optimization pass is implemented in `src/x86_32/x86_32_reg_vars.c`. It is called register variable optimization because it locates when and where it is possible to keep variables in registers.

The register variables optimization is located in the `src/x86_32` directory since it optimizes on the AIA code after it has been converted to a form close to x86-32 assembly by the `src/x86_32/x86_32_normalize.c` file.

It is easy to keep a variable in a register, the tricky thing is to find out when the value of the variable might be expected to be in memory.

The optimization pass is global, meaning it's analyzing a whole function at a time. It keeps track of two per instruction sets, a use set and a def (definition) set.

The use set contains memory variables which are used or redefined by some function which might get called later. If a variable is in the use set we must move the value of the variable to memory, because a function might need it.

The def set contains memory variables which gets redefined by some function which might have been called. Initially all memory variables used in the current function are in the def set. If a variable is in the def set the variable might have been redefined by a function, thus if we have the value for the variable in a register that value is not valid any more.

Consider the vitaly program:

```

1 func f():void
2     func g():void
3         write i;
4     end g
5
6     func h():void
7         i = 10;
8     end h

```

```

9      i:int = 1; # move value of i into register esi.
10
11      # Do stuff here, with i kept in register esi ...
12
13      g(); # Save i in memory before this call.
14
15      # Do stuff here, with i in register esi ...
16
17      h(); # Save i in memory before this call.
18
19      # Previous function call might have redefined i, thus
20      # the value of i in register esi might not be valid anymore.
21      # Move i back into a register before using it next time...
22      end f
23

```

A 64 bit linux installation running on an Intel core i7 CPU, the vitality program located in `unit_tests/ExamplePrograms/knapsack/O_Knapsack.vit` compiled with register variables optimization disabled (`-O0 -fno-reg-vars`) takes approximately 11.5 seconds to execute. With register variable optimization enabled the program takes approximately 9.2 seconds to execute. That is quite significant.

Note that a C implementation of that program is located in `unit_tests/ExamplePrograms/knapsack/knapsack.c` compiled with `gcc -O3 -m32` that program takes approximately 8.9 seconds to execute.

12 Register allocation

Before code can be generated, we need to assign the temporaries introduced by the AIA to actual registers. To do this we perform a liveness analysis of registers and temporaries. For each live set the following must hold:

$$\text{in}[n] = \text{use}[n] \cup \left(\left(\bigcup_{s \in \text{succ}[n]} \text{in}[s] \right) - \text{def}[n] \right)$$

where $\text{in}[n]$ is the set of registers (and temporaries) that are live into the instruction n . The set $\text{use}[n]$ is the set of registers being used by instruction n that is the registers being used as source operands in instruction n . Similar the set $\text{def}[n]$ is the set of operands being defined by instruction n that is the registers being used as destination operands. To perform the analysis the instructions of the AIA are linked with a live set which are iteratively filled with operands according to the formula above. When the live sets have reached a state where they no longer change the liveness analysis is complete.

Below is a dump of initial liveness analysis shown:

```

1  # AIA block start
2  .WHILE.top.4: # live { 8 }
3      .mov @size(4) %8 -> @size(4) %eax # live { 8 }
4      .cdq @size(4) %eax -> @size(4) %edx # live { eax 8 }
5      .mov @size(4) $2 -> @size(4) %7 # live { eax 8 edx }
6      .idiv @size(4) %eax, %7 -> @size(4) %eax # live { 7 eax 8 edx }
7      .cmp @size(4) %eax, $0 # live { eax 8 }
8      .jne @size(4) .IF.else.7, .IF.then.5 # live { 8 }
9  # AIA block end

```

Notice how `%eax` is live into line 4 until it is last used in line 7. Similarly `%edx` is live in lines 5 and 6, and `%7` is only live in line 6. See also how the temporary `%8` is live through the whole block, the reason for this is outside the scope of this block. The

use of the liveness analysis becomes clear now as we can see that the two temporaries %7 and %8 must be assigned to different registers and further more they may not be assigned to the registers `eax` or `edx`. This is due to the fact that the temporaries and registers appear in the same live set at some instruction.

To implement this application of the liveness analysis we have implemented an interference graph, which captures the interferences of the live sets as edges and the registers and temporaries as nodes in the graph. Then the problem can be solved as a graph coloring problem where the colors represent actual registers. It is of course preferable to use the as few colors as possible when coloring the interference graph, since this corresponds to using the least number of registers. This is important due to the fact that the number of registers is limited.

To approximate a minimum graph coloring we have use color by simplification. This works by iteratively choosing a node of insignificant degree, meaning the node has fewer neighbours than the number of registers available. The chosen node is removed from the graph and queued on a stack, waiting to get colored. When the node is removed the neighbouring nodes will have their degree lowered and possibly become of insignificant degree. If at some point no nodes can be removed a node is chosen for a potential spill, removed from the graph, added to the stack and the algorithm continues. In our implementation we choose the node of highest degree among the remaining for spill.

It is clear that some measures must be taken for pre-colored nodes, that is register which are already assigned. For example register `eax` and `edx` in relation to division instructions. To handle this we simply insert these as normal operands in the graph, but we do not allow for them to be removed from the graph and put into the coloring stack.

After the stack is created we simply pop the nodes from the stack, rebuilding the interference graph while assigning registers to them.

If spills were necessary the spills are handled by introducing a spill variable in memory, which is inserted in place of the spilled register where ever it is used. Then another passes will take care of normalizing the affected instructions, and the whole process can be repeated. Instead of always introducing new variables when spill occur, we utilize the fact that some temporaries are replacements of existing variables, therefore if a mapping from a spilled temporary to a variable exists, this variable is used instead of creating a new. Thus the total amount of memory used by the executable is reduced.

Returning to the example from above, we have the following register allocation:

```
1 # AIA block start
2 .WHILE.top.4:
3     .mov @size(4) %ecx -> @size(4) %eax
4     .cdq @size(4) %eax -> @size(4) %edx
5     .mov @size(4) $2 -> @size(4) %ebx
6     .idiv @size(4) %eax, %ebx -> @size(4) %eax
7     .cmp @size(4) %eax, $0
8     .jne @size(4) .IF.else.7, .IF.then.5
9 # AIA block end
```

Notice how %8 is assigned to `%ecx` and %7 to `%ebx`, that they are not assigned to

the same register and that they are different from `eax` and `edx` as expected.

13 Unused Move Elimination

This optimization phase is actually run just before the register allocation but it uses the information obtained from liveness analysis, which we introduced in the previous section on register allocation. For this reason this section is placed after register allocation.

In this phase we try to eliminate `mov` instructions which could be avoided. The idea of this phase is to identify moves from a register or temporary to a temporary, where the temporary we are moving to can be replaced by the register we are moving from.

If we consider:

```
1 .mov @size(4) %1 -> @size(4) %2
2 .mov @size(4) %eax -> @size(4) %3
```

we would like to replace the temporary `%2` with the temporary `%1`, and replace `%3` by `%eax`. If this is possible the `mov` instruction is insignificant and can simply be removed.

In the following we call the register or temporary we are replacing with `A` and the temporary we are replacing `B`. The temporary `B` can be replaced by `A` if and only if the following conditions hold:

- `B` may not be redefined while `A` is live.
- If the `A` is a caller save register then no `call` instructions are allowed in the live range of `B`
- `A` may only be redefined in the live range of `B` if it is by a `mov` instruction where either `A` or `B` is the source.

We now consider the first condition. Assume `B` was replaced by `A`. Then if `B` was redefined while `A` is live, then we introduce a redefinition of `A` before its use hence we `A` is not guaranteed to be valid when it is used. Thus this condition must hold for a replacement to be valid.

If the temporary `B` were replaced by a caller save register and a call instruction is executed before `B` is used, we cannot guarantee `B` to be valid after the call instruction. Thus the second condition must hold.

Finally consider the last condition which is almost symmetric to the first condition only slightly more relaxed. If `B` is replaced by `A` then in either case of the condition corresponds to an insignificant move and hence this case is allowed. Otherwise the same reasoning as for the first condition applies.

Consider the following example from Knapsack

(`--dump=init-liveness-x86-32-ic`):

```
1 # AIA block start
2 .IF.then.34: # live { 251 252 245 246 247 248 249 }
3 .mov @size(4) %251 -> @size(4) %68 # live { 251 252 245 246 247 248 249 }
4 .add @size(4) %68, $1 -> @size(4) %68 # live { 252 68 245 246 247 248 249 }
5 .mov @size(4) %68 -> @size(4) %251 # live { 252 68 245 246 247 248 249 }
6 .mov @size(4) %251 -> @size(4) %69 # live { 251 252 245 246 247 248 249 }
7 .mov @size(4) %69 -> @size(4) @arg(0) # live { 251 252 69 245 246 247 248 249 }
8 .mov @size(4) %252 -> @size(4) %70 # live { 251 252 245 246 247 248 249 }
```

```

9      .mov @size(4) %70 -> @size(4) @arg(1) # live { 251 252 70 245 246 247 248 249 }
10     .call @size(4) Fii.O_Knapsack.exchange -> @size(1) %al # live { 251 252 245 246
      247 248 249 }
11     .mov @size(1) %al -> @size(1) %71 # live { 251 252 al 245 246 247 248 249 }
12     .jmp @size(4) .IF.end.35 # live { 251 252 245 246 247 248 249 }
13 # AIA block end

```

We start by trying to replace %68 with %251 through the live range of %68. We see that the conditions hold and do the replacement. Notice that %68 is defined in line 4, but %251 is not live. In a similar manner %69 is replaced by %251, %70 by %252 and %71 by %al. The result can be seen below (`--dump=unused-mov`):

```

1 # AIA block start
2 .IF.then.34: # live { 251 252 245 246 247 248 249 }
3   .mov @size(4) %251 -> @size(4) %251 # live { 251 252 245 246 247 248 249 }
4   .add @size(4) %251, $1 -> @size(4) %251 # live { 251 252 245 246 247 248 249 }
5   .mov @size(4) %251 -> @size(4) %251 # live { 251 252 245 246 247 248 249 }
6   .mov @size(4) %251 -> @size(4) %251 # live { 251 252 245 246 247 248 249 }
7   .mov @size(4) %251 -> @size(4) @arg(0) # live { 251 252 245 246 247 248 249 }
8   .mov @size(4) %252 -> @size(4) %252 # live { 251 252 245 246 247 248 249 }
9   .mov @size(4) %252 -> @size(4) @arg(1) # live { 251 252 245 246 247 248 249 }
10  .call @size(4) Fii.O_Knapsack.exchange -> @size(1) %al # live { 251 252 245 246
      247 248 249 }
11  .mov @size(1) %al -> @size(1) %al # live { 251 252 al 245 246 247 248 249 }
12  .jmp @size(4) .IF.end.35 # live { 251 252 245 246 247 248 249 }
13 # AIA block end

```

This gives us that lines 3, 5, 6 and 8 can be identified as unused moves and therefore can be removed. Note that a separate pass takes care of removing the introduced no-op `mov` instructions. To see the final impact of this pass, consider the following assembler output for our running example:

Without unused move elimination:

```

1 .IF.then.34:
2 # line 114
3   movl %esi, %eax
4   addl $1, %eax
5   movl %eax, %esi
6 # line 115
7   movl %esi, %eax
8   movl %eax, (%esp)
9   movl %edi, %eax
10  movl %eax, 4(%esp)
11  call Fii.O_Knapsack.exchange

```

With unused move elimination:

```

1 .IF.then.34:
2 # line 114
3   addl $1, %edi
4 # line 115
5   movl %edi, (%esp)
6   movl %ebx, 4(%esp)
7   call Fii.O_Knapsack.exchange

```

It is clear how the some `mov` instructions have been identified as unused and has been eliminated. The outputs above can be generated with the compile flags `-s` and `-s -Oign-unused-mov`.

14 Testing

Throughout the implementation of the compiler a number of test cases have been implemented. Some of these tests files are included in the `test_programs` directory. The vitality test programs contained in this directory are meant to test various aspects of the compiler such as error reporting, invalid files and optimization passes impact on emitted assembler.

Furthermore we have spend time to improve overall software quality of the compiler by ensuring it is free of memory leaks. This can be tested with Valgrind, like in the following example:

```
1 valgrind --leak-check=full ./vitaly --stubborn test_programs/*
```

This will try to compile all files in the `test_programs` directory. Because of the `--stubborn` flag the compiler will continue through all files even though errors have been reported. Finally valgrind will output something like:

```
1 ==12949== HEAP SUMMARY:
2 ==12949== in use at exit: 0 bytes in 0 blocks
3 ==12949== total heap usage: 644,394 allocs, 644,394 frees, 47,525,439 bytes
   allocated
4 ==12949==
5 ==12949== All heap blocks were freed -- no leaks are possible
6 ==12949==
7 ==12949== For counts of detected and suppressed errors, rerun with: -v
8 ==12949== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 2 from 2)
```

Indicating that everything allocated have been successfully freed before termination of the compiler.

Other test programs located in the `unit_test` directory, are all valid vitaly programs and can automatically be compiled, run and output correctness tested by:

```
1 unit_tests/test.sh
```

Notice to further check the compiler for memory leaks and invalid reads one can specify the command line argument `valg` when running `test.sh` which enables valgrind for each time a program is compiled. An output from running `test.sh` can be seen below:

```
1 vitaly ./string.vit
2 vitaly ./import7_1.vit
3 vitaly ./ctor01.vit
4 vitaly ./declarations.vit
5 vitaly ./std_std0.vit
6 vitaly ./import3_1.vit
7 vitaly ./import1_1.vit
8 vitaly ./func_overload02.vit
9 vitaly ./unused_import_collision2.vit
10 vitaly ./empty_statements.vit
11 vitaly ./finalize.vit
12 vitaly ./import7_2.vit
13 vitaly ./unused_import_collision1.vit
14 vitaly ./import4_1.vit
15 vitaly ./import1_2.vit
16 vitaly ./import2_1.vit
17 vitaly ./immediate_var_init.vit
18 vitaly ./ary0.vit
19 vitaly ./short_circuit2.vit
20 vitaly ./multiple_dot.vit
21 vitaly ./import2_3.vit
22 vitaly ./record_sep.vit
23 vitaly ./call_nest2.vit
24 vitaly ./empty_func_body.vit
25 vitaly ./std_string01.vit
26 vitaly ./std_errno.vit
27 vitaly ./call_nest1.vit
28 vitaly ./import2_2.vit
29 vitaly ./import5_3.vit
30 vitaly ./unused_import_collision3.vit
31 vitaly ./vector01.vit
32 vitaly ./expression_as_statement.vit
33 vitaly ./std_math.vit
34 vitaly ./thread01.vit
35 vitaly ./import5_2.vit
```

```
36 vitality ./import3_2.vit
37 vitality ./std_array.vit
38 vitality ./nest1.vit
39 vitality ./char.vit
40 vitality ./methods.vit
41 vitality ./import5_1.vit
42 vitality ./import6_2.vit
43 vitality ./import6_1.vit
44 vitality ./func_overload03.vit
45 vitality ./import4_2.vit
46 vitality ./func_overload01.vit
47 vitality ./import7_3.vit
48 vitality ./var_in_rec.vit
49 vitality ./std_comparator.vit
50 vitality ./empty_record_body.vit
51 vitality ./short_circuit1.vit
52 (51/51) successful tests
```

Notice that all the tests are successful. If a test failed, either errors from the compiler, or differences of output compared to the expected output would be reported. This test is particularly useful for a fast test if changes to the compiler introduced runtime and compile time bugs. For a more comprehensive test one should test compilation of the programs in `test_programs` since these as mentioned also include tests of error messages. The implementation of the test programs can be found in the mentioned directories.

A Source Code

A.1 src/main.c

```

1  #include <sys/stat.h>
2  #include <unistd.h>
3  #include <getopt.h>
4  #include <main.h>
5  #include <parser.h>
6  #include <errno.h>
7  #include <debug.h>
8  #include <string_builder.h>
9  #include <test/test_include.h>
10 #include <ast/symbol_table.h>
11 #include <ast/ast_visitor_type_check.h>
12 #include <ast/ast_visitor_import.h>
13 #include <ast/ast_visitor_aia.h>
14 #include <aia/aia_functions_return.h>
15 #include <aia/aia_normalize_addr.h>
16 #include <aia/aia_optimize.h>
17 #include <aia/aia_block_elim.h>
18 #include <x86_32/x86_32.h>
19 #include <import_handler.h>
20 #include <help_msg.h>
21
22 #undef DEBUG_TYPE
23 #define DEBUG_TYPE main
24
25 #ifndef DEFAULT_OBJ_DIR
26 #error Missing macro definition DEFAULT_OBJ_DIR
27 #endif
28
29 #define DEFAULT_MAX_MSG INT32_MAX
30
31 Command_Line_Options cmdopts = {
32     .vitaly_program_name = NULL,
33     .output_name = NULL,
34     .import_search_paths = VECTOR_STATIC_INIT(),
35     .working_directory = NULL,
36
37     .dump_parse_tree = false,
38     .dump_parse_tree_graph = false,
39     .dump_symbol_table_graph = false,
40     .dump_symbol_table = false,
41
42     .dump_init_ic = false,
43     .dump_norm_addr_ic = false,
44     .dump_const_prop_ic = false,
45     .dump_instr_elim_ic = false,
46
47     .dump_norm_x86_32_ic = false,
48     .dump_init_liveness_x86_32_ic = false,
49     .dump_liveness_x86_32_ic = false,
50     .dump_reg_alloc_x86_32_ic = false,
51     .dump_unused_mov_ic = false,
52     .dump_reg_vars_liveness_ic = false,
53     .dump_reg_vars_ic = false,
54     .dump_final_x86_32_ic = false,
55     .dump_def_to_use_ic = false,
56     .dump_warn_uninit_liveness_ic = false,
57
58     .dump_c_header = false,
59     .dump_asm = false,
60
61     .opt_func_access = true,
62     .opt_const_prop = true,
63     .opt_instr_elim = true,
64     .opt_unused_mov = true,
65     .opt_unused_set = true,
66     .opt_reg_vars = true,
67     .opt_def_to_use = true,
68

```

```

69     .warn_no_finalize = false,
70     .warn_implicit_cast = true,
71     .warn_ref_compare = true,
72     .warn_overflow = true,
73     .warn_div_zero = true,
74     .warn_uninitialized = true,
75     .warn_is_error = false,
76
77     .compile_only = false,
78     .assemble_only = false,
79     .keep_obj = false,
80
81     .library_init = false,
82     .link_libvitaly = true,
83     .with_threads = false,
84     .has_main = false,
85
86     .recursive_compile = false,
87     .generate_viti = false,
88
89     .stubborn = false,
90     .verbose_output = false,
91
92     .max_msg = DEFAULT_MAX_MSG
93 };
94
95 typedef enum Option_Parse_Status {
96     OPTION_PARSE_CONTINUE,
97     OPTION_PARSE_EXIT_SUCCESS,
98     OPTION_PARSE_EXIT_FAILURE
99 } Option_Parse_Status;
100
101 enum Option_Indices {
102     OUT_OPTION,
103     MAIN_OPTION,
104     LIB_INIT_OPTION,
105     IGN_LIB_INIT_OPTION,
106     NO_LIB_VIT_OPTION,
107     THREAD_OPTION,
108     IGN_THREAD_OPTION,
109     IGN_NO_LIB_VIT_OPTION,
110     LIB_PATH_OPTION,
111     LIB_OPTION,
112     IMPORT_PATH_OPTION,
113     IGN_MAIN_OPTION,
114     DUMP_OPTION,
115     OPTIMIZE_OPTION,
116     MAX_MSG_OPTION,
117     STUBBORN_OPTION,
118     IGN_STUBBORN_OPTION,
119     WARNING_OPTION,
120     COMPILE_ONLY_OPTION,
121     IGN_COMPILE_ONLY_OPTION,
122     ASM_ONLY_OPTION,
123     IGN_ASM_ONLY_OPTION,
124     KEEP_OBJ_OPTION,
125     IGN_KEEP_OBJ_OPTION,
126     VERBOSE_OPTION,
127     IGN_VERBOSE_OPTION,
128     HELP_OPTION,
129     RECURSIVE_COMPILE_OPTION,
130     IGN_RECURSIVE_COMPILE_OPTION,
131     GENERATE_VITI_OPTION,
132     IGN_GENERATE_VITI_OPTION,
133     NULL_OPTION // Must be last.
134 };
135
136 static const char short_options[] = "Irvskmhcxw:o:L:l:O:i:";
137
138 static struct option long_options[] = {
139     [OUT_OPTION] = {
140         "output",
141         required_argument,
142         NULL,

```

```

143     'o'
144 },
145 #define OUT_OPTION_CSTR QFY("-o") " (" QFY("--output") ") "
146 [MAIN_OPTION] = {
147     "main",
148     required_argument,
149     NULL,
150     'm'
151 },
152 [IGN_MAIN_OPTION] = {
153     "ign-main",
154     required_argument,
155     NULL,
156     0
157 },
158 [LIB_INIT_OPTION] = {
159     "lib-init",
160     no_argument,
161     NULL,
162     0
163 },
164 [IGN_LIB_INIT_OPTION] = {
165     "ign-lib-init",
166     no_argument,
167     NULL,
168     0
169 },
170 [THREAD_OPTION] = {
171     "thread",
172     no_argument,
173     NULL,
174     0
175 },
176 #define THREAD_OPTION_CSTR QFY("--thread")
177 [IGN_THREAD_OPTION] = {
178     "ign-thread",
179     no_argument,
180     NULL,
181     0
182 },
183 [NO_LIB_VIT_OPTION] = {
184     "no-libvit",
185     no_argument,
186     NULL,
187     'x'
188 },
189 #define NO_LIB_VIT_OPTION_CSTR QFY("-x") " (" QFY("--no-libvit") ") "
190 [IGN_NO_LIB_VIT_OPTION] = {
191     "ign-no-libvit",
192     no_argument,
193     NULL,
194     0
195 },
196 [LIB_PATH_OPTION] = {
197     "lib-path",
198     required_argument,
199     NULL,
200     'L'
201 },
202 [LIB_OPTION] = {
203     "lib",
204     required_argument,
205     NULL,
206     'l'
207 },
208 [IMPORT_PATH_OPTION] = {
209     "import-path",
210     required_argument,
211     NULL,
212     'i'
213 },
214 [DUMP_OPTION] = {
215     "dump",
216     required_argument,

```



```

217     NULL,
218     0
219 },
220 #define DUMP_OPTION_CSTR QFY("--dump")
221 [OPTIMIZE_OPTION] = {
222     "optimize",
223     required_argument,
224     NULL,
225     'O'
226 },
227 #define OPTIMIZE_OPTION_CSTR QFY("-O") " (" QFY("--optimize") ")"
228 [MAX_MSG_OPTION] = {
229     "max-msg",
230     required_argument,
231     NULL,
232     0
233 },
234 [STUBBORN_OPTION] = {
235     "stubborn",
236     no_argument,
237     NULL,
238     0
239 },
240 [IGN_STUBBORN_OPTION] = {
241     "ign-stubborn",
242     no_argument,
243     NULL,
244     0
245 },
246 [WARNING_OPTION] = {
247     "warning",
248     required_argument,
249     NULL,
250     'w'
251 },
252 #define WARNING_OPTION_CSTR QFY("-w") " (" QFY("--warning") ")"
253 [COMPILE_ONLY_OPTION] = {
254     "compile-only",
255     no_argument,
256     NULL,
257     'c'
258 },
259 #define COMPILE_ONLY_OPTION_CSTR QFY("-c") " (" QFY("--compile-only") ")"
260 [IGN_COMPILE_ONLY_OPTION] = {
261     "ign-compile-only",
262     no_argument,
263     NULL,
264     0
265 },
266 [ASM_ONLY_OPTION] = {
267     "asm-only",
268     no_argument,
269     NULL,
270     's'
271 },
272 #define ASM_ONLY_OPTION_CSTR QFY("-s") " (" QFY("--asm-only") ")"
273 [IGN_ASM_ONLY_OPTION] = {
274     "ign-asm-only",
275     no_argument,
276     NULL,
277     0
278 },
279 [KEEP_OBJ_OPTION] = {
280     "keep-obj",
281     no_argument,
282     NULL,
283     'k'
284 },
285 [IGN_KEEP_OBJ_OPTION] = {
286     "ign-keep-obj",
287     no_argument,
288     NULL,
289     0
290 },

```

```

291     [VERBOSE_OPTION] = {
292         "verbose",
293         no_argument,
294         NULL,
295         'v'
296     },
297     [IGN_VERBOSE_OPTION] = {
298         "ign-verbose",
299         no_argument,
300         NULL,
301         0
302     },
303     [RECURSIVE_COMPILE_OPTION] = {
304         "recursive",
305         no_argument,
306         NULL,
307         'r'
308     },
309 #define RECURSIVE_OPTION_CSTR QFY("-r") " (" QFY("--recursive") ")"
310     [IGN_RECURSIVE_COMPILE_OPTION] = {
311         "ign-recursive",
312         no_argument,
313         NULL,
314         0
315     },
316 #define IGN_RECURSIVE_OPTION_CSTR QFY("--ign-recursive")
317     [GENERATE_VITI_OPTION] = {
318         "gen-viti",
319         no_argument,
320         NULL,
321         'I'
322     },
323     [IGN_GENERATE_VITI_OPTION] = {
324         "ign-gen-viti",
325         no_argument,
326         NULL,
327         0
328     },
329     [HELP_OPTION] = {
330         "help",
331         optional_argument,
332         NULL,
333         'h'
334     },
335     [NULL_OPTION] = {
336         NULL,
337         0,
338         NULL,
339         0
340     }
341 };
342
343 static String assemble(Const_String asm_file, Const_String src_file);
344
345 VECTOR(gen_object_files);
346 VECTOR(cmd_object_files);
347 VECTOR(lib_paths);
348 VECTOR(lib_names);
349
350 static void __add_object_file(Vector *vec, String obj, Const_String err_msg)
351 {
352     if (obj) {
353         errno = 0;
354         if (!file_access_read(obj)) {
355             if (errno)
356                 report_error(obj, S("%S [%m]\n"), err_msg);
357             else
358                 report_error(obj, S("%S\n"), err_msg);
359             string_destroy(obj);
360         } else {
361             vector_append(vec, obj);
362         }
363     }
364 }

```

```

365
366 static void add_object_file(Vector *vec, String obj)
367 {
368     __add_object_file(vec, obj,
369         S("unable to open object file"));
370 }
371
372 void add_gen_object_file(String obj)
373 {
374     add_object_file(&gen_object_files, obj);
375 }
376
377 static void add_cmd_object_file(String obj)
378 {
379     if (cmdopts.assemble_only) {
380         report_warning(obj, S("file ignored since the "
381             ASM_ONLY_OPTION_CSTR
382             " option is enabled\n"));
383         goto err_out;
384     }
385     if (cmdopts.compile_only) {
386         report_warning(obj, S("file ignored since the "
387             COMPILER_ONLY_OPTION_CSTR
388             " option is enabled\n"));
389         goto err_out;
390     }
391     add_object_file(&cmd_object_files, obj);
392     return;
393 }
394
395 err_out:
396     string_destroy(obj);
397 }
398
399 static void add_cmd_library_file(String obj)
400 {
401     __add_object_file(&cmd_object_files, obj,
402         S("unable to open library file"));
403 }
404
405 static void assemble_add_object(Const_String asm_file)
406 {
407     if (cmdopts.assemble_only) {
408         report_warning(asm_file, S("assembly source file ignored since the "
409             ASM_ONLY_OPTION_CSTR " option is enabled\n"));
410         return;
411     }
412     String obj_file = assemble(asm_file, asm_file);
413     add_gen_object_file(obj_file);
414 }
415
416 static bool verify_dir_path(Const_String pathname,
417     Const_String path_type)
418 {
419     {
420         struct stat s;
421         if (pathname) {
422             int ret = stat(string_to_cstr(pathname), &s);
423             if (ret == -1) {
424                 report_error(pathname, S("cannot access %S [%m]\n"), path_type);
425                 return false;
426             } else if (!S_ISDIR(s.st_mode)) {
427                 report_error(pathname, S("%S expected to be a directory\n"),
428                     path_type);
429                 return false;
430             } else {
431                 return true;
432             }
433         }
434         return false;
435     }
436 }
437
438 static void add_lib_path(String path)
439 {

```

```

439     if (verify_dir_path(path, S("library path")))
440         vector_append(&lib_paths, path);
441     else
442         string_destroy(path);
443 }
444
445 static void add_lib_name(String lib)
446 {
447     vector_append(&lib_names, lib);
448 }
449
450 static void add_import_path(String path)
451 {
452     if (verify_dir_path(path, S("import path")))
453         vector_append(&cmdopts.import_search_paths, path);
454     else
455         string_destroy(path);
456 }
457
458 static bool get_int32_argument(const char *option_name,
459                             int32_t min_val, int32_t *result)
460 {
461     int32_t n;
462     errno = 0;
463     n = string_base10_to_int32(S(optarg));
464     if (errno) {
465         if (errno == EINVAL) {
466             print_error(S("option " QFY("%s") " requires integer argument, "
467                          "found " QFY("%s") "\n"),
468                        option_name, optarg);
469         } else if (errno == ERANGE) {
470             print_error(S("option " QFY("%s") " argument " QFY("%s")
471                          " %s integer\n"),
472                        option_name,
473                        optarg,
474                        n == INT32_MAX ? "overflows" : "underflows");
475         } else {
476             print_error(S("unable to convert option " QFY ("%s")
477                          " argument " QFY("%s") " to integer\n"),
478                        option_name, optarg);
479         }
480         goto err_out;
481     }
482     if (n < min_val) {
483         print_error(S("option " QFY("%s") " requires integer argument bigger "
484                      "than %" PRId32 " found " QFY("%s") "\n"),
485                    option_name, min_val, optarg);
486         goto err_out;
487     }
488     *result = n;
489     return true;
490 }
491
492 err_out:
493     return false;
494 }
495
496 static bool get_int32_argument_long_option(const char *option_name,
497                                           int32_t min_val, int32_t *result)
498 {
499     bool ret;
500     STRING(opt_str, "--");
501     string_append(opt_str, S(option_name));
502     ret = get_int32_argument(string_to_cstr(opt_str), min_val, result);
503     string_clear(opt_str);
504     return ret;
505 }
506
507 static UNUSED bool get_int32_argument_short_option(Int option_name,
508                                                    int32_t min_val, int32_t *result)
509 {
510     bool ret;
511     option_name &= ~0xff;
512     option_name <= 8;

```

```

513     option_name |= '-';
514     ret = get_int32_argument((char *)&option_name, min_val, result);
515     return ret;
516 }
517
518 static inline void set_warn_all(bool b)
519 {
520     cmdopts.warn_implicit_cast = b;
521     cmdopts.warn_no_finalize = b;
522     cmdopts.warn_ref_compare = b;
523     cmdopts.warn_overflow = b;
524     cmdopts.warn_div_zero = b;
525     cmdopts.warn_uninitialized = b;
526 }
527
528
529 static Option_Parse_Status parse_warn_option_argument(Const_String arg)
530 {
531     Option_Parse_Status ret = OPTION_PARSE_CONTINUE;
532     if (!string_compare(arg, S("implicit-cast"))) {
533         cmdopts.warn_implicit_cast = true;
534     } else if (!string_compare(arg, S("ign-implicit-cast"))) {
535         cmdopts.warn_implicit_cast = false;
536     } else if (!string_compare(arg, S("no-finalize"))) {
537         cmdopts.warn_no_finalize = true;
538     } else if (!string_compare(arg, S("ign-no-finalize"))) {
539         cmdopts.warn_no_finalize = false;
540     } else if (!string_compare(arg, S("ref-compare"))) {
541         cmdopts.warn_ref_compare = true;
542     } else if (!string_compare(arg, S("ign-ref-compare"))) {
543         cmdopts.warn_ref_compare = false;
544     } else if (!string_compare(arg, S("is-error"))) {
545         cmdopts.warn_is_error = true;
546     } else if (!string_compare(arg, S("ign-is-error"))) {
547         cmdopts.warn_is_error = false;
548     } else if (!string_compare(arg, S("overflow"))) {
549         cmdopts.warn_overflow = true;
550     } else if (!string_compare(arg, S("ign-overflow"))) {
551         cmdopts.warn_overflow = false;
552     } else if (!string_compare(arg, S("div-zero"))) {
553         cmdopts.warn_div_zero = true;
554     } else if (!string_compare(arg, S("ign-div-zero"))) {
555         cmdopts.warn_div_zero = false;
556     } else if (!string_compare(arg, S("uninitialized"))) {
557         cmdopts.warn_uninitialized = true;
558     } else if (!string_compare(arg, S("ign-uninitialized"))) {
559         cmdopts.warn_uninitialized = false;
560     } else if (!string_compare(arg, S("all"))) {
561         set_warn_all(true);
562     } else if (!string_compare(arg, S("ign-all"))) {
563         set_warn_all(false);
564     } else {
565         print_error(S("unrecognized " WARNING_OPTION_CSTR " option argument "
566                     QFY("%S") "\n"), arg);
567         ret = OPTION_PARSE_EXIT_FAILURE;
568     }
569     return ret;
570 }
571
572 static inline void set_optimize_all(bool b)
573 {
574     cmdopts.opt_func_access = b;
575     cmdopts.opt_const_prop = b;
576     cmdopts.opt_instr_elim = b;
577     cmdopts.opt_unused_mov = b;
578     cmdopts.opt_unused_set = b;
579     cmdopts.opt_reg_vars = b;
580     cmdopts.opt_def_to_use = b;
581 }
582
583 static Option_Parse_Status parse_optimize_option_argument(Const_String arg)
584 {
585     Option_Parse_Status ret = OPTION_PARSE_CONTINUE;
586     if (!string_compare(arg, S("func-access"))) {

```

```

587     cmdopts.opt_func_access = true;
588 } else if (!string_compare(arg, S("ign-func-access"))) {
589     cmdopts.opt_func_access = false;
590 } else if (!string_compare(arg, S("const-prop"))) {
591     cmdopts.opt_const_prop = true;
592 } else if (!string_compare(arg, S("ign-const-prop"))) {
593     cmdopts.opt_const_prop = false;
594 } else if (!string_compare(arg, S("instr-elim"))) {
595     cmdopts.opt_instr_elim = true;
596 } else if (!string_compare(arg, S("ign-instr-elim"))) {
597     cmdopts.opt_instr_elim = false;
598 } else if (!string_compare(arg, S("unused-mov"))) {
599     cmdopts.opt_unused_mov = true;
600 } else if (!string_compare(arg, S("ign-unused-mov"))) {
601     cmdopts.opt_unused_mov = false;
602 } else if (!string_compare(arg, S("unused-set"))) {
603     cmdopts.opt_unused_set = true;
604 } else if (!string_compare(arg, S("ign-unused-set"))) {
605     cmdopts.opt_unused_set = false;
606 } else if (!string_compare(arg, S("reg-vars"))) {
607     cmdopts.opt_reg_vars = true;
608 } else if (!string_compare(arg, S("ign-reg-vars"))) {
609     cmdopts.opt_reg_vars = false;
610 } else if (!string_compare(arg, S("def-to-use"))) {
611     cmdopts.opt_def_to_use = true;
612 } else if (!string_compare(arg, S("ign-def-to-use"))) {
613     cmdopts.opt_def_to_use = false;
614 } else if (!string_compare(arg, S("all"))) {
615     set_optimize_all(true);
616 } else if (!string_compare(arg, S("ign-all"))) {
617     set_optimize_all(false);
618 } else {
619     print_error(S("unrecognized " OPTIMIZE_OPTION_CSTR " option argument "
620                 QFY("%S") "\n"), arg);
621     ret = OPTION_PARSE_EXIT_FAILURE;
622 }
623 return ret;
624 }
625
626 static Option_Parse_Status parse_dump_option_argument(Const_String arg)
627 {
628     Option_Parse_Status ret = OPTION_PARSE_CONTINUE;
629
630     if (!string_compare(arg, S("parse-tree"))) {
631         cmdopts.dump_parse_tree = true;
632     } else if (!string_compare(arg, S("ign-parse-tree"))) {
633         cmdopts.dump_parse_tree = false;
634     } else if (!string_compare(arg, S("symbol-table"))) {
635         cmdopts.dump_symbol_table = true;
636     } else if (!string_compare(arg, S("ign-symbol-table"))) {
637         cmdopts.dump_symbol_table = false;
638     } else if (!string_compare(arg, S("parse-tree-graph"))) {
639         cmdopts.dump_parse_tree_graph = true;
640     } else if (!string_compare(arg, S("ign-parse-tree-graph"))) {
641         cmdopts.dump_parse_tree_graph = false;
642     } else if (!string_compare(arg, S("symbol-table-graph"))) {
643         cmdopts.dump_symbol_table_graph = true;
644     } else if (!string_compare(arg, S("ign-symbol-table-graph"))) {
645         cmdopts.dump_symbol_table_graph = false;
646     } else if (!string_compare(arg, S("c-header"))) {
647         cmdopts.dump_c_header = true;
648     } else if (!string_compare(arg, S("ign-c-header"))) {
649         cmdopts.dump_c_header = false;
650     } else if (!string_compare(arg, S("init-ic"))) {
651         cmdopts.dump_init_ic = true;
652     } else if (!string_compare(arg, S("ign-init-ic"))) {
653         cmdopts.dump_init_ic = false;
654     } else if (!string_compare(arg, S("norm-addr-ic"))) {
655         cmdopts.dump_norm_addr_ic = true;
656     } else if (!string_compare(arg, S("ign-norm-addr-ic"))) {
657         cmdopts.dump_norm_addr_ic = false;
658     } else if (!string_compare(arg, S("const-prop-ic"))) {
659         cmdopts.dump_const_prop_ic = true;
660     } else if (!string_compare(arg, S("ign-const-prop-ic"))) {

```

```

661     cmdopts.dump_const_prop_ic = false;
662 } else if (!string_compare(arg, S("instr-elim-ic"))) {
663     cmdopts.dump_instr_elim_ic = true;
664 } else if (!string_compare(arg, S("ign-instr-elim-ic"))) {
665     cmdopts.dump_instr_elim_ic = false;
666 } else if (!string_compare(arg, S("norm-x86-32-ic"))) {
667     cmdopts.dump_norm_x86_32_ic = true;
668 } else if (!string_compare(arg, S("ign-norm-x86-32-ic"))) {
669     cmdopts.dump_norm_x86_32_ic = false;
670 } else if (!string_compare(arg, S("liveness-x86-32-ic"))) {
671     cmdopts.dump_liveness_x86_32_ic = true;
672 } else if (!string_compare(arg, S("ign-liveness-x86-32-ic"))) {
673     cmdopts.dump_liveness_x86_32_ic = false;
674 } else if (!string_compare(arg, S("init-liveness-x86-32-ic"))) {
675     cmdopts.dump_init_liveness_x86_32_ic = true;
676 } else if (!string_compare(arg, S("ign-init-liveness-x86-32-ic"))) {
677     cmdopts.dump_init_liveness_x86_32_ic = false;
678 } else if (!string_compare(arg, S("reg-vars-liveness-ic"))) {
679     cmdopts.dump_reg_vars_liveness_ic = true;
680 } else if (!string_compare(arg, S("ign-reg-vars-liveness-ic"))) {
681     cmdopts.dump_reg_vars_liveness_ic = false;
682 } else if (!string_compare(arg, S("reg-vars-ic"))) {
683     cmdopts.dump_reg_vars_ic = true;
684 } else if (!string_compare(arg, S("ign-reg-vars-ic"))) {
685     cmdopts.dump_reg_vars_ic = false;
686 } else if (!string_compare(arg, S("unused-mov-ic"))) {
687     cmdopts.dump_unused_mov_ic = true;
688 } else if (!string_compare(arg, S("ign-unused-mov-ic"))) {
689     cmdopts.dump_unused_mov_ic = false;
690 } else if (!string_compare(arg, S("reg-alloc-x86-32-ic"))) {
691     cmdopts.dump_reg_alloc_x86_32_ic = true;
692 } else if (!string_compare(arg, S("ign-reg-alloc-x86-32-ic"))) {
693     cmdopts.dump_reg_alloc_x86_32_ic = false;
694 } else if (!string_compare(arg, S("def-to-use-ic"))) {
695     cmdopts.dump_def_to_use_ic = true;
696 } else if (!string_compare(arg, S("ign-def-to-use-ic"))) {
697     cmdopts.dump_def_to_use_ic = false;
698 } else if (!string_compare(arg, S("warn-uninit-liveness-ic"))) {
699     cmdopts.dump_warn_uninit_liveness_ic = true;
700 } else if (!string_compare(arg, S("ign-warn-uninit-liveness-ic"))) {
701     cmdopts.dump_def_to_use_ic = false;
702 } else if (!string_compare(arg, S("final-x86-32-ic"))) {
703     cmdopts.dump_final_x86_32_ic = true;
704 } else if (!string_compare(arg, S("ign-final-x86-32-ic"))) {
705     cmdopts.dump_final_x86_32_ic = false;
706 } else if (!string_compare(arg, S("asm"))) {
707     cmdopts.dump_asm = true;
708 } else if (!string_compare(arg, S("ign-asm"))) {
709     cmdopts.dump_asm = false;
710 } else {
711     print_error(S("unrecognized " DUMP_OPTION_CSTR " option argument "
712                  QFY("%S") "\n"), arg);
713     ret = OPTION_PARSE_EXIT_FAILURE;
714 }
715
716 return ret;
717 }
718
719 static inline Option_Parse_Status parse_list_option_arguments(
720     Option_Parse_Status (*option_parser)(Const_String arg))
721 {
722     Option_Parse_Status ret = OPTION_PARSE_CONTINUE;
723
724     STRING(optarg_str, optarg);
725     Vector *args = string_split(optarg_str, S(", "));
726     string_clear(optarg_str);
727
728     String arg;
729     VECTOR_FOR_EACH_ENTRY(args, arg)
730     if ((ret = option_parser(arg)) != OPTION_PARSE_CONTINUE)
731         break;
732     vector_destroy(args, (Vector_Destructor)string_destroy);
733
734     return ret;

```

```

735 }
736
737 static int unrecognized_short_option_count;
738
739 static Option_Parse_Status option_parse_error(char *argv[])
740 {
741     int i;
742     const char *tmp, *tmp_str;
743     Option_Parse_Status ret = OPTION_PARSE_EXIT_FAILURE;
744
745     tmp = argv[optind - 1];
746     if (strstr(tmp, "--") == tmp) {
747         tmp_str = tmp + 2;
748         for (i = 0; i < NULL_OPTION; i++) {
749             if (strcmp(tmp_str, long_options[i].name) == 0) {
750                 print_error(S("option " QFY("%s")
751                     " requires an argument\n"), tmp);
752                 goto out;
753             }
754         }
755         print_error(S("unrecognized option " QFY("%s") "\n"), tmp);
756         goto out;
757     } else if (strchr(short_options, optopt)) {
758         print_error(S("option " QFY("-%c") " requires an argument\n"), optopt);
759         goto out;
760     }
761
762     ++unrecognized_short_option_count;
763     print_error(S("unrecognized option " QFY("-%c") "\n"), optopt);
764
765 out:
766     return ret;
767 }
768
769 static Option_Parse_Status parse_help_option_argument(Const_String arg)
770 {
771     Option_Parse_Status ret = OPTION_PARSE_EXIT_SUCCESS;
772
773     if (!string_compare(arg, S("dump"))) {
774         print_message(dump_help_msg);
775     } else if (!string_compare(arg, S("warning"))) {
776         print_message(warn_help_msg);
777     } else if (!string_compare(arg, S("optimize"))) {
778         print_message(opt_help_msg);
779     } else {
780         print_error(S("unrecognized " DUMP_OPTION_CSTR " option argument "
781             QFY("%s") "\n"), arg);
782         ret = OPTION_PARSE_EXIT_FAILURE;
783     }
784     return ret;
785 }
786
787 static Option_Parse_Status parse_help_option()
788 {
789     Option_Parse_Status ret = OPTION_PARSE_EXIT_SUCCESS;
790     if (optarg)
791         ret = parse_help_option_argument(S(optarg));
792     else
793         print_message(help_msg, cmdopts.vitaly_program_name);
794     return ret;
795 }
796
797 static Option_Parse_Status parse_long_option(int idx)
798 {
799     Option_Parse_Status ret = OPTION_PARSE_CONTINUE;
800     switch (idx) {
801     case DUMP_OPTION:
802         ret = parse_list_option_arguments(parse_dump_option_argument);
803         break;
804     case MAX_MSG_OPTION:
805         if (!get_int32_argument_long_option(long_options[idx].name, 1,
806             &cmdopts.max_msg))
807             ret = OPTION_PARSE_EXIT_FAILURE;
808         break;

```



```

809     case STUBBORN_OPTION:
810         cmdopts.stubborn = true;
811         break;
812     case IGN_STUBBORN_OPTION:
813         cmdopts.stubborn = false;
814         break;
815     case IGN_COMPILE_ONLY_OPTION:
816         cmdopts.compile_only = false;
817         break;
818     case IGN_ASM_ONLY_OPTION:
819         cmdopts.assemble_only = false;
820         break;
821     case IGN_KEEP_OBJ_OPTION:
822         cmdopts.keep_obj = false;
823         break;
824     case IGN_MAIN_OPTION:
825         cmdopts.has_main = false;
826         break;
827     case IGN_VERBOSE_OPTION:
828         cmdopts.verbose_output = false;
829         break;
830     case IGN_GENERATE_VITI_OPTION:
831         cmdopts.generate_viti = false;
832         break;
833     case IGN_RECURSIVE_COMPILE_OPTION:
834         cmdopts.recursive_compile = false;
835         break;
836     case LIB_INIT_OPTION:
837         cmdopts.library_init = true;
838         break;
839     case IGN_LIB_INIT_OPTION:
840         cmdopts.library_init = false;
841         break;
842     case IGN_NO_LIB_VIT_OPTION:
843         cmdopts.link_libvitaly = true;
844         break;
845     case THREAD_OPTION:
846         cmdopts.with_threads = true;
847         break;
848     case IGN_THREAD_OPTION:
849         cmdopts.with_threads = false;
850         break;
851     case NO_LIB_VIT_OPTION:
852         /* Should not happen. */
853     case KEEP_OBJ_OPTION:
854         /* Should not happen. */
855     case OPTIMIZE_OPTION:
856         /* Should not happen. */
857     case WARNING_OPTION:
858         /* Should not happen. */
859     case MAIN_OPTION:
860         /* Should not happen. */
861     case COMPILE_ONLY_OPTION:
862         /* Should not happen. */
863     case ASM_ONLY_OPTION:
864         /* Should not happen. */
865     case OUT_OPTION:
866         /* Should not happen. */
867     case VERBOSE_OPTION:
868         /* Should not happen. */
869     case HELP_OPTION:
870         /* Should not happen. */
871     case NULL_OPTION:
872         /* Should not happen. */
873         break;
874     }
875
876     return ret;
877 }
878
879 static Option_Parse_Status set_output_name()
880 {
881     string_destroy(cmdopts.output_name);
882     cmdopts.output_name = string_alloc(S(optarg));

```

```

883     return OPTION_PARSE_CONTINUE;
884 }
885
886 static bool verify_long_option_name(int option_index UNUSED,
887     char *argv[] UNUSED)
888 {
889     #if 0
890     if (!option_index)
891         return true;
892
893     int arg_idx;
894     if (long_options[option_index].has_arg)
895         arg_idx = optind - 2;
896     else
897         arg_idx = optind - 1;
898
899     if (strcmp(long_options[option_index].name, argv[arg_idx] + 2)) {
900         print_error(S("unrecognized option " QFY("%s") "\n"), argv[arg_idx]);
901         return false;
902     }
903     #endif
904
905     return true;
906 }
907
908 static Option_Parse_Status merge_new_parse_status(Option_Parse_Status prev_opt,
909     Option_Parse_Status new_opt)
910 {
911     Option_Parse_Status ret = prev_opt;
912
913     switch (new_opt) {
914     case OPTION_PARSE_CONTINUE:
915         break;
916     case OPTION_PARSE_EXIT_SUCCESS:
917         if (ret != OPTION_PARSE_EXIT_FAILURE)
918             ret = OPTION_PARSE_EXIT_SUCCESS;
919         break;
920     case OPTION_PARSE_EXIT_FAILURE:
921         ret = OPTION_PARSE_EXIT_FAILURE;
922         break;
923     };
924
925     return ret;
926 }
927
928 static Option_Parse_Status parse_cmd_options(int argc, char *argv[])
929 {
930     int c, option_index;
931     Option_Parse_Status ret = OPTION_PARSE_CONTINUE;
932     Option_Parse_Status tmp_opt;
933
934     cmdopts.vitaly_program_name = string_basename(S(argv[0]));
935
936     opterr = 0;
937     option_index = 0;
938     while ((c = getopt_long(argc, argv, short_options,
939         long_options, &option_index)) != -1) {
940
941         if (!verify_long_option_name(option_index, argv)) {
942             ret = OPTION_PARSE_EXIT_FAILURE;
943             goto skip_arg;
944         }
945
946         switch (c) {
947         case 0:
948             tmp_opt = parse_long_option(option_index);
949             ret = merge_new_parse_status(ret, tmp_opt);
950             break;
951
952         case 'h':
953             tmp_opt = parse_help_option();
954             ret = merge_new_parse_status(ret, tmp_opt);
955             break;
956

```

```

957     case 'w':
958         tmp_opt = parse_list_option_arguments(parse_warn_option_argument);
959         ret = merge_new_parse_status(ret, tmp_opt);
960         break;
961
962     case 'c':
963         cmdopts.compile_only = true;
964         break;
965
966     case 'r':
967         cmdopts.recursive_compile = true;
968         break;
969
970     case 's':
971         cmdopts.assemble_only = true;
972         break;
973
974     case 'k':
975         cmdopts.keep_obj = true;
976         break;
977
978     case 'x':
979         cmdopts.link_libvitaly = false;
980         break;
981
982     case 'o':
983         tmp_opt = set_output_name();
984         ret = merge_new_parse_status(ret, tmp_opt);
985         break;
986
987     case 'm':
988         cmdopts.has_main = true;
989         break;
990
991     case 'L':
992         add_lib_path(string_alloc(S(optarg)));
993         break;
994
995     case 'l':
996         add_lib_name(string_alloc(S(optarg)));
997         break;
998
999     case 'i':
1000         add_import_path(string_alloc(S(optarg)));
1001         break;
1002
1003     case 'I':
1004         cmdopts.generate_viti = true;
1005         break;
1006
1007     case 'v':
1008         cmdopts.verbose_output = true;
1009         break;
1010
1011     case 'O':
1012         tmp_opt = parse_list_option_arguments(parse_optimize_option_argument);
1013         ret = merge_new_parse_status(ret, tmp_opt);
1014         break;
1015
1016     default:
1017         tmp_opt = option_parse_error(argv);
1018         ret = merge_new_parse_status(ret, tmp_opt);
1019         break;
1020 }
1021
1022 if (unrecognized_short_option_count >= 3)
1023     goto out;
1024
1025 skip_arg:
1026     option_index = 0;
1027 }
1028
1029 out:
1030     return ret;

```

```

1031 }
1032
1033 static CONST_STRING(x86_32_as_fmt, "as -g --32 -o %S %S");
1034
1035 /* Returns name of the object file on success. NULL on error. */
1036 static String assemble(Const_String asm_fname, Const_String src_fname)
1037 {
1038     String obj_file;
1039     if (cmdopts.compile_only || cmdopts.keep_obj) {
1040         if (cmdopts.compile_only && cmdopts.output_name) {
1041             obj_file = string_duplicate(cmdopts.output_name);
1042         } else {
1043             obj_file = string_duplicate(src_fname);
1044             string_replace_from(obj_file, '.', OBJ_SUFFIX_STR);
1045         }
1046     } else {
1047         obj_file = string_to_tmp_file(OBJ_SUFFIX_STR);
1048     }
1049
1050     String cmd = string_from_format(x86_32_as_fmt, obj_file, asm_fname);
1051
1052     if (cmdopts.verbose_output)
1053         file_print_message(stdout, S("%S\n"), cmd);
1054
1055     Int sys_ret = sys_cmd(cmd);
1056
1057     if (sys_ret) {
1058         string_destroy(obj_file);
1059         obj_file = NULL;
1060     }
1061
1062     string_destroy(cmd);
1063     return obj_file;
1064 }
1065
1066 /* Returns name of the object file on success. NULL on error. */
1067 static String compile(Ast *ast)
1068 {
1069     import_handler_init();
1070     String ret = __compile(ast, false);
1071     import_handler_clear();
1072     return ret;
1073 }
1074
1075 String __compile(Ast *ast, bool is_import)
1076 {
1077     Aia *aia;
1078     Symbol_Table *t;
1079     String ret = NULL;
1080
1081     if (!ast_is_valid(ast))
1082         goto ast_out;
1083
1084     if (!ast_visitor_import_handle(ast, is_import))
1085         goto ast_out;
1086
1087     t = ast_get_resolve_symbol_table(ast);
1088
1089     if (cmdopts.dump_symbol_table)
1090         symbol_table_dump(t, ast_get_file_name(ast));
1091
1092     if (cmdopts.dump_symbol_table_graph)
1093         symbol_table_dump_graph(t, ast_get_file_name(ast));
1094
1095     ast_visitor_type_check(ast);
1096
1097     if (cmdopts.dump_parse_tree)
1098         ast_dump_parse_tree(ast);
1099
1100     if (cmdopts.dump_parse_tree_graph)
1101         ast_dump_parse_tree_graph(ast);
1102
1103     aia = ast_visitor_aia_create(ast, ast_get_file_name(ast));
1104 }

```

```

1105     if (!aia_is_valid(aia))
1106         goto ast_out;
1107
1108     if (!is_import)
1109         ast_destroy(ast);
1110
1111     ast = NULL;
1112
1113     if (!aia_functions_return(aia))
1114         goto aia_out;
1115
1116     aia_normalize_addr(aia);
1117
1118     aia_optimize(aia);
1119
1120     String asm_file = x86_32_gen(aia);
1121     if (!asm_file)
1122         goto aia_out;
1123
1124     ret = assemble(asm_file, aia_get_file_name(aia));
1125     if (!cmdopts.dump_asm)
1126         file_unlink_temp(asm_file);
1127     string_destroy(asm_file);
1128
1129 aia_out:
1130     aia_destroy(aia);
1131
1132 ast_out:
1133     if (!is_import)
1134         ast_destroy(ast);
1135
1136     return ret;
1137 }
1138
1139 CONST_STRING(link_fmt,
1140     "gcc " //
1141     "-m32 " // link x86-32 code
1142     "-o%S " // executable name
1143     "-L" DEFAULT_OBJ_DIR " " // libvitaly directory
1144     DEFAULT_OBJ_DIR "/ini.o " // start of _init() and _fini()
1145     DEFAULT_OBJ_DIR "/lib.o " // write, allocate, etc.
1146     DEFAULT_OBJ_DIR "/%S.o " // which main function to use
1147     "%S " // object files & libraries
1148     "%S " // further options (" " is fine)
1149     "%S " // optional libvitaly.a
1150     "%S " // optional string to link threads.
1151     DEFAULT_OBJ_DIR "/end.o"); // end of _init() and _fini()
1152
1153 CONST_STRING(def_exec_name, "a.out");
1154
1155 static void object_link_add_libs(String_Builder *sb)
1156 {
1157     Const_String tmp;
1158
1159     VECTOR_FOR_EACH_ENTRY(&lib_paths, tmp) {
1160         string_builder_append(sb, S(" -L "));
1161         string_builder_append(sb, tmp);
1162     }
1163     VECTOR_FOR_EACH_ENTRY(&lib_names, tmp) {
1164         string_builder_append(sb, S(" -l "));
1165         string_builder_append(sb, tmp);
1166     }
1167 }
1168
1169 static void object_link()
1170 {
1171     if (cmdopts.compile_only ||
1172         cmdopts.assemble_only ||
1173         get_error_count())
1174         return;
1175
1176     String_Builder objs = STRING_BUILDER_INIT();
1177
1178     String ob;

```

```

1179     VECTOR_FOR_EACH_ENTRY(&cmd_object_files, ob) {
1180         string_builder_append_char(&objs, ' ');
1181         string_builder_append(&objs, ob);
1182     }
1183
1184     VECTOR_FOR_EACH_ENTRY(&gen_object_files, ob) {
1185         string_builder_append_char(&objs, ' ');
1186         string_builder_append(&objs, ob);
1187     }
1188
1189     Const_String start_file;
1190     if (cmdopts.has_main)
1191         start_file = S("vitmain");
1192     else
1193         start_file = S("retmain");
1194
1195     String_Builder optional = STRING_BUILDER_INIT();
1196     string_builder_assign(&optional, S(""));
1197
1198     object_link_add_libs(&optional);
1199
1200     Const_String executable;
1201     if (cmdopts.output_name)
1202         executable = cmdopts.output_name;
1203     else
1204         executable = def_exec_name;
1205
1206     Const_String libvit;
1207     if (cmdopts.link_libvitaly)
1208         libvit = S("-Wl,--whole-archive -lvitaly -Wl,--no-whole-archive");
1209     else
1210         libvit = S("");
1211
1212     Const_String pthread;
1213     if (cmdopts.with_threads)
1214         pthread = S("-pthread "
1215             "-Wl,--whole-archive -lvitaly-thread -Wl,--no-whole-archive");
1216     else
1217         pthread = S("");
1218
1219     String cmd = string_from_format(link_fmt,
1220         executable,
1221         start_file,
1222         string_builder_const_str(&objs),
1223         string_builder_const_str(&optional),
1224         libvit,
1225         pthread);
1226
1227     if (cmdopts.verbose_output)
1228         file_print_message(stdout, S("%S\n"), cmd);
1229
1230     sys_cmd(cmd);
1231
1232     string_destroy(cmd);
1233     string_builder_clear(&objs);
1234     string_builder_clear(&optional);
1235
1236     if (!cmdopts.keep_obj) {
1237         VECTOR_FOR_EACH_ENTRY(&gen_object_files, ob)
1238             file_unlink_temp(ob);
1239     }
1240 }
1241
1242 static void unrecognized_suffix(Const_String file)
1243 {
1244     String base = string_basename(file);
1245     Const_String suffix = STRING_AFTER_LAST(base, '.');
1246     if (string_compare(suffix, base))
1247         report_error(file, S("unrecognized file extension " QFY("%S") "\n"),
1248             suffix);
1249     else
1250         report_error(file, S("file name missing " QFY(".") " extension\n"));
1251     string_destroy(base);
1252 }

```

```

1253
1254 static Ast *parse_input_file(Const_String file)
1255 {
1256     Ast *ast = NULL;
1257     if (string_ends_with(file, SOURCE_SUFFIX_STR))
1258         ast = parse(file);
1259     else if (string_ends_with(file, OBJ_SUFFIX_STR))
1260         add_cmd_object_file(string_duplicate(file));
1261     else if (string_ends_with(file, STATIC_LIB_SUFFIX_STR))
1262         add_cmd_library_file(string_duplicate(file));
1263     else if (string_ends_with(file, ASM1_SUFFIX_STR))
1264         assemble_add_object(file);
1265     else if (string_ends_with(file, ASM2_SUFFIX_STR))
1266         assemble_add_object(file);
1267     else
1268         unrecognized_suffix(file);
1269     return ast;
1270 }
1271
1272 #ifndef STDIN_INPUT_ENABLED
1273 #define STDIN_BUF_SIZE 1024
1274 static String stdin_to_tmp_file()
1275 {
1276     ssize_t nread;
1277     char buf[STDIN_BUF_SIZE];
1278     String fname = string_to_unique_file(STDIN_FILE_SUFFIX_STR);
1279     FILE *tmpf = file_open(fname, S("w"));
1280     if (!tmpf) {
1281         fatal_error(S("unable to create temporary file %S "
1282             "for stdin input [%m]\n"),
1283             fname);
1284     }
1285
1286     do {
1287         nread = read(STDIN_FILENO, buf, STDIN_BUF_SIZE);
1288         if (nread == -1)
1289             fatal_error(S("error writing to temporary file %S [%m]\n"), fname);
1290         fwrite(buf, 1, nread, tmpf);
1291     } while (nread);
1292
1293     file_close(tmpf);
1294     return fname;
1295 }
1296 #endif
1297
1298 static inline void too_many_src_files_error()
1299 {
1300     fatal_error(S("only one source file allowed when the "
1301         OUT_OPTION_CSTR " option is specified with the "
1302         COMPILE_ONLY_OPTION_CSTR " or "
1303         ASM_ONLY_OPTION_CSTR " option\n"));
1304 }
1305
1306 static void verify_cmd_options(int argc, char *argv[])
1307 {
1308     int vit_src_cnt = 0;
1309     int asm_src_cnt = 0;
1310
1311     inline void vit_file(Const_String file UNUSED)
1312     {
1313         ++vit_src_cnt;
1314     }
1315
1316     inline void asm_file(Const_String file)
1317     {
1318         if (cmdopts.assemble_only)
1319             report_warning(file, S("assembly source file ignored since the "
1320                 ASM_ONLY_OPTION_CSTR " option is enabled\n"));
1321         ++asm_src_cnt;
1322     }
1323
1324     inline int src_file_count()
1325     {
1326         return vit_src_cnt + asm_src_cnt;

```

```

1327     }
1328
1329     if (cmdopts.compile_only && cmdopts.output_name) {
1330         for (int i = optind; i < argc; i++) {
1331             CONST_STRING(src, argv[i]);
1332
1333             if (string_ends_with(src, SOURCE_SUFFIX_STR))
1334                 vit_file(src);
1335             else if (string_ends_with(src, ASM1_SUFFIX_STR))
1336                 asm_file(src);
1337             else if (string_ends_with(src, ASM2_SUFFIX_STR))
1338                 asm_file(src);
1339
1340             if (src_file_count() > 1)
1341                 too_many_src_files_error();
1342         }
1343     }
1344
1345     if (cmdopts.recursive_compile && cmdopts.output_name) {
1346         if (cmdopts.assemble_only) {
1347             fatal_error(S("option " RECURSIVE_OPTION_CSTR
1348                          " disallowed when both " ASM_ONLY_OPTION_CSTR
1349                          " and " OUT_OPTION_CSTR " options are enabled\n"));
1350         } else if (cmdopts.compile_only) {
1351             fatal_error(S("option " RECURSIVE_OPTION_CSTR
1352                          " disallowed when both " COMPILE_ONLY_OPTION_CSTR
1353                          " and " OUT_OPTION_CSTR " options are enabled\n"));
1354         }
1355     }
1356
1357     if (cmdopts.compile_only && cmdopts.assemble_only)
1358         report_warning(cmdopts.vitaly_program_name,
1359                        S("option " COMPILE_ONLY_OPTION_CSTR
1360                         " ignored since the " ASM_ONLY_OPTION_CSTR
1361                         " option is enabled\n"));
1362
1363     if (cmdopts.with_threads && !cmdopts.link_libvitaly) {
1364         report_warning(cmdopts.vitaly_program_name,
1365                        S("option " THREAD_OPTION_CSTR
1366                         " has no effect when the " NO_LIB_VIT_OPTION_CSTR
1367                         " option is enabled\n"));
1368         cmdopts.with_threads = false;
1369     }
1370 }
1371
1372 static bool process_input_file(Const_String in_file)
1373 {
1374     Ast *ast;
1375
1376     report_reset();
1377     ast = parse_input_file(in_file);
1378     String obj = compile(ast);
1379     add_gen_object_file(obj);
1380     show_reports_clear();
1381
1382     if (was_error_reported() && !cmdopts.stubborn)
1383         return false;
1384     if (report_exhausted())
1385         return false;
1386     return true;
1387 }
1388
1389 static void process_stdin()
1390 {
1391     #ifdef STDIN_INPUT_ENABLED
1392         String stdin_file = stdin_to_tmp_file();
1393         process_input_file(stdin_file);
1394         file_unlink_temp(stdin_file);
1395         string_destroy(stdin_file);
1396     #else
1397         fatal_error(S("no input files, try\n\t%S --help\n"),
1398                    cmdopts.vitaly_program_name);
1399     #endif
1400 }

```



```

1401
1402 static void process_input_files(int argc, char *argv[])
1403 {
1404     for (int i = optind; i < argc; i++) {
1405         if (!process_input_file(S(argv[i])))
1406             return;
1407     }
1408 }
1409
1410 static inline void destroy_string_vec(Vector *v)
1411 {
1412     vector_for_each_destroy(v, (Vector_Destructor)string_destroy);
1413 }
1414
1415 void cd_working_dir()
1416 {
1417     if (chdir(string_to_cstr(cmdopts.working_directory)))
1418         fatal_error(S("unable to change to directory %S [%m]"),
1419                     cmdopts.working_directory);
1420 }
1421
1422 static inline void save_working_dir()
1423 {
1424     char *dir = getcwd(NULL, 0);
1425     cmdopts.working_directory = string_alloc(S(dir));
1426     free_mem(dir);
1427 }
1428
1429 static inline void add_default_import_paths()
1430 {
1431     if (cmdopts.link_libvitaly)
1432         vector_append(&cmdopts.import_search_paths,
1433                     string_alloc(S(DEFAULT_OBJ_DIR)));
1434     if (cmdopts.with_threads)
1435         vector_append(&cmdopts.import_search_paths,
1436                     string_alloc(S(DEFAULT_OBJ_DIR "/_vit_thread")));
1437 }
1438
1439 int main(int argc, char *argv[])
1440 {
1441     int ret;
1442
1443     switch (parse_cmd_options(argc, argv)) {
1444     case OPTION_PARSE_CONTINUE:
1445         break;
1446     case OPTION_PARSE_EXIT_SUCCESS:
1447         ret = EXIT_SUCCESS;
1448         goto out;
1449     case OPTION_PARSE_EXIT_FAILURE:
1450         ret = EXIT_FAILURE;
1451         goto out;
1452     }
1453     verify_cmd_options(argc, argv);
1454
1455     save_working_dir();
1456     add_default_import_paths();
1457
1458     if (optind >= argc)
1459         process_stdin();
1460     else
1461         process_input_files(argc, argv);
1462
1463     parse_cleanup();
1464     object_link();
1465     report_print();
1466
1467     ret = get_error_count() ? EXIT_FAILURE : EXIT_SUCCESS;
1468 out:
1469     destroy_string_vec(&cmdopts.import_search_paths);
1470     destroy_string_vec(&gen_object_files);
1471     destroy_string_vec(&cmd_object_files);
1472     destroy_string_vec(&lib_names);
1473
1474

```

```

1475     string_destroy(cmdopts.vitaly_program_name);
1476     string_destroy(cmdopts.output_name);
1477     string_destroy(cmdopts.working_directory);
1478
1479     return ret;
1480 }

```

A.2 src/main.h

```

1  #ifndef MAIN_H
2  #define MAIN_H
3
4  #include <std_define.h>
5  #include <std_include.h>
6  #include <vector.h>
7  #include <ast/ast.h>
8
9  typedef struct Command_Line_Options {
10     String vitaly_program_name;
11     String output_name;
12     Vector import_search_paths;
13     String working_directory;
14     bool dump_parse_tree;
15     bool dump_parse_tree_graph;
16     bool dump_symbol_table_graph;
17     bool dump_symbol_table;
18     bool dump_c_header;
19     bool dump_init_ic;
20     bool dump_norm_addr_ic;
21     bool dump_const_prop_ic;
22     bool dump_instr_elim_ic;
23     bool dump_norm_x86_32_ic;
24     bool dump_liveness_x86_32_ic;
25     bool dump_reg_vars_liveness_ic;
26     bool dump_reg_vars_ic;
27     bool dump_init_liveness_x86_32_ic;
28     bool dump_unused_mov_ic;
29     bool dump_reg_alloc_x86_32_ic;
30     bool dump_final_x86_32_ic;
31     bool dump_def_to_use_ic;
32     bool dump_warn_uninit_liveness_ic;
33     bool dump_asm;
34     bool opt_func_access;
35     bool opt_const_prop;
36     bool opt_instr_elim;
37     bool opt_unused_mov;
38     bool opt_unused_set;
39     bool opt_reg_vars;
40     bool opt_def_to_use;
41     bool stubborn;
42     bool warn_implicit_cast;
43     bool warn_no_finalize;
44     bool warn_ref_compare;
45     bool warn_overflow;
46     bool warn_is_error;
47     bool warn_div_zero;
48     bool warn_uninitialized;
49     bool compile_only;
50     bool assemble_only;
51     bool keep_obj;
52     bool verbose_output;
53     bool library_init;
54     bool link_libvitaly;
55     bool with_threads;
56     bool has_main;
57     bool recursive_compile;
58     bool generate_viti;
59     int32_t max_msg;
60 } Command_Line_Options;
61

```

```

62 String __compile(Ast *ast, bool is_import);
63
64 void add_gen_object_file(String obj);
65
66 extern Command_Line_Options cmdopts;
67
68 void cd_working_dir();
69
70 #endif // MAIN_H

```

A.3 Scanning/Parsing

:

A.3.1 src/ast/ast.c

```

1  #include "ast_visitor.h"
2  #include "ast_visitor_delete.h"
3  #include "ast_visitor_symbol_table.h"
4  #include "ast_visitor_print.h"
5  #include "ast_visitor_print_graph.h"
6  #include "symbol_table.h"
7  #include "ast_string.h"
8
9  Const_String const __ast_expr_type_strings[AST_EXPR_TYPE_COUNT] = {
10     [AST_EXPR_TYPE_UNSPECIFIED] = S("unspecified"),
11     [AST_EXPR_TYPE_UNKNOWN] = S("unknown"),
12     [AST_EXPR_TYPE_VOID] = S("void"),
13     [AST_EXPR_TYPE_INT] = S("int"),
14     [AST_EXPR_TYPE_BOOL] = S("bool"),
15     [AST_EXPR_TYPE_CHAR] = S("char"),
16     [AST_EXPR_TYPE_STRING] = S("string"),
17     [AST_EXPR_TYPE_REC] = S("record"),
18     [AST_EXPR_TYPE_ARY] = S("array"),
19     [AST_EXPR_TYPE_NULL] = S("null")
20 };
21
22 #define AST_NODE_ACCEPT_VISITOR_DEF(node_type) \
23     AST_NODE_ACCEPT_VISITOR_DECL(node_type) \
24     { \
25         v->AST_VISITOR_FUNC(node_type)(v, n); \
26     }
27
28 String ast_module_string(Const_String initial)
29 {
30     String str = string_duplicate(initial);
31     string_replace_all(str, '.', '/');
32     return str;
33 }
34
35 static inline bool ast_string_has_expected_last(Ast_Node *n, String str,
36     Int expected)
37 {
38     Uns len = string_length(str);
39     if (string_get(str, len - 1) != (char)expected) {
40         expected &= (Int)0xff;
41         report_error_location(
42             S("expected terminating " QFY("%s") " before end of line\n"),
43             &expected);
44         string_set(str, len - 1, '\0');
45         return false;
46     }
47     return true;
48 }
49
50 int32_t ast_char_to_int32(Ast_Expr_Char *n, String char_str)
51 {
52     Uns len;

```

```

53     int32_t ret = 0;
54
55     if (!ast_string_has_expected_last(&n->ast_node, char_str, '\\'))
56         goto out;
57     string_remove_first_last(char_str);
58
59     String orig = string_duplicate(char_str);
60
61     len = string_length(char_str);
62     if (len == 0) {
63         report_error_location(&n->ast_node.location,
64             S("Expected " QFY("char") " inside single quotes '\\n"));
65     } else if (len > 2 || (len == 2 && string_get(char_str, 0) != '\\')) {
66         report_error_location(&n->ast_node.location,
67             S("Invalid char " QFY("%S") "\\n"), orig);
68     } else if (!__ast_string_unescape(&n->ast_node, char_str)) {
69         ret = string_get(char_str, 0);
70     }
71
72     string_destroy(orig);
73
74 out:
75     return ret;
76 }
77
78 String ast_get_expr_string(Ast_Expr_String *n, String str)
79 {
80     String ret = string_duplicate(str);
81     if (ast_string_has_expected_last(&n->ast_node, ret, '\\')) {
82         string_remove_first_last(ret);
83         __ast_string_unescape(&n->ast_node, ret);
84     }
85     return ret;
86 }
87
88 void ast_release_nodes(Ast *ast)
89 {
90     if (ast)
91         if (ast->root) {
92             ast_visitor_delete_accept_visitor(ast->root);
93             ast->root = NULL;
94         }
95 }
96
97 void ast_release_symbol_table(Ast *ast)
98 {
99     if (ast)
100         if (ast->symbol_table) {
101             symbol_table_destroy(ast->symbol_table);
102             ast->symbol_table = NULL;
103         }
104 }
105
106 void ast_destroy(Ast *ast)
107 {
108     ast_release_nodes(ast);
109     ast_release_symbol_table(ast);
110     if (ast) {
111         string_destroy(ast->dir);
112         string_destroy(ast->file);
113         free_mem(ast);
114     }
115 }
116
117 Symbol_Table *ast_get_symbol_table(Ast *ast)
118 {
119     if (!ast->symbol_table)
120         ast->symbol_table = ast_visitor_symbol_table_gen(ast);
121     return ast->symbol_table;
122 }
123
124 Const_String ast_get_dirname(Ast *ast)
125 {
126     if (!ast->dir)

```

```

127     ast->dir = string_dirname(ast_get_file_name(ast));
128     return ast->dir;
129 }
130
131 Const_String ast_get_package(Ast *ast)
132 {
133     return ast->package;
134 }
135
136 Symbol_Table *ast_get_resolve_symbol_table(Ast *ast)
137 {
138     Symbol_Table *symbol_table = ast_get_symbol_table(ast);
139     symbol_table_resolve(symbol_table);
140     return symbol_table;
141 }
142
143 Symbol_Table *ast_move_symbol_table(Ast *ast)
144 {
145     Symbol_Table *ret = ast->symbol_table;
146     ast->symbol_table = NULL;
147     return ret;
148 }
149
150 void ast_dump_parse_tree(Ast *ast)
151 {
152     ast_visitor_print_accept_visitor(ast->root, ast->file);
153 }
154
155 void ast_dump_parse_tree_graph(Ast *ast)
156 {
157     ast_visitor_print_graph_accept_visitor(ast->root, ast->file);
158 }
159
160 AST_NODE_ACCEPT_VISITOR_DEF(AST_EXPR_LOR);
161 AST_NODE_ACCEPT_VISITOR_DEF(AST_EXPR_LAND);
162 AST_NODE_ACCEPT_VISITOR_DEF(AST_EXPR_EQ);
163 AST_NODE_ACCEPT_VISITOR_DEF(AST_EXPR_NEQ);
164 AST_NODE_ACCEPT_VISITOR_DEF(AST_EXPR_GT);
165 AST_NODE_ACCEPT_VISITOR_DEF(AST_EXPR_LT);
166 AST_NODE_ACCEPT_VISITOR_DEF(AST_EXPR_GTEQ);
167 AST_NODE_ACCEPT_VISITOR_DEF(AST_EXPR_LTEQ);
168 AST_NODE_ACCEPT_VISITOR_DEF(AST_EXPR_PLUS);
169 AST_NODE_ACCEPT_VISITOR_DEF(AST_EXPR_MINUS);
170 AST_NODE_ACCEPT_VISITOR_DEF(AST_EXPR_MUL);
171 AST_NODE_ACCEPT_VISITOR_DEF(AST_EXPR_DIV);
172 AST_NODE_ACCEPT_VISITOR_DEF(AST_EXPR_CAST);
173 AST_NODE_ACCEPT_VISITOR_DEF(AST_EXPR_LNOT);
174 AST_NODE_ACCEPT_VISITOR_DEF(AST_EXPR_ABS);
175 AST_NODE_ACCEPT_VISITOR_DEF(AST_EXPR_INT);
176 AST_NODE_ACCEPT_VISITOR_DEF(AST_EXPR_BOOL);
177 AST_NODE_ACCEPT_VISITOR_DEF(AST_EXPR_NULL);
178 AST_NODE_ACCEPT_VISITOR_DEF(AST_VARIABLE_IDEN);
179 AST_NODE_ACCEPT_VISITOR_DEF(AST_EXPR_ARY_REF);
180 AST_NODE_ACCEPT_VISITOR_DEF(AST_EXPR_FUNC_CALL);
181 AST_NODE_ACCEPT_VISITOR_DEF(AST_EXPR_DOT_REF);
182 AST_NODE_ACCEPT_VISITOR_DEF(AST_SIMPLE_TYPE_INT);
183 AST_NODE_ACCEPT_VISITOR_DEF(AST_SIMPLE_TYPE_BOOL);
184 AST_NODE_ACCEPT_VISITOR_DEF(AST_TYPE_IDEN);
185 AST_NODE_ACCEPT_VISITOR_DEF(AST_VAR_DECL);
186 AST_NODE_ACCEPT_VISITOR_DEF(AST_TYPE_DEF);
187 AST_NODE_ACCEPT_VISITOR_DEF(AST_TYPE_ARY);
188 AST_NODE_ACCEPT_VISITOR_DEF(AST_TYPE_REC);
189 AST_NODE_ACCEPT_VISITOR_DEF(AST_STMT_LIST);
190 AST_NODE_ACCEPT_VISITOR_DEF(AST_FUNC_DEF);
191 AST_NODE_ACCEPT_VISITOR_DEF(AST_IF_STMT);
192 AST_NODE_ACCEPT_VISITOR_DEF(AST_IF_ELSE_STMT);
193 AST_NODE_ACCEPT_VISITOR_DEF(AST_ALLOC_REC);
194 AST_NODE_ACCEPT_VISITOR_DEF(AST_ALLOC_ARY);
195 AST_NODE_ACCEPT_VISITOR_DEF(AST_WHILE_STMT);
196 AST_NODE_ACCEPT_VISITOR_DEF(AST_RETURN_STMT);
197 AST_NODE_ACCEPT_VISITOR_DEF(AST_WRITE_STMT);
198 AST_NODE_ACCEPT_VISITOR_DEF(AST_ASSIGNMENT);
199 AST_NODE_ACCEPT_VISITOR_DEF(AST_FUNC_BODY);
200 AST_NODE_ACCEPT_VISITOR_DEF(AST_EXPR_CHAR);

```

```

201 AST_NODE_ACCEPT_VISITOR_DEF(AST_SIMPLE_TYPE_CHAR);
202 AST_NODE_ACCEPT_VISITOR_DEF(AST_EXPR_STRING);
203 AST_NODE_ACCEPT_VISITOR_DEF(AST_SIMPLE_TYPE_STRING);
204 AST_NODE_ACCEPT_VISITOR_DEF(AST_IMPORT_STRING);
205 AST_NODE_ACCEPT_VISITOR_DEF(AST_PACKAGE_STRING);
206 AST_NODE_ACCEPT_VISITOR_DEF(AST_REC_SELF_PTR);
207 AST_NODE_ACCEPT_VISITOR_DEF(AST_SIMPLE_TYPE_VOID);
208 AST_NODE_ACCEPT_VISITOR_DEF(AST_FIN_FUNC_DEF);
209 AST_NODE_ACCEPT_VISITOR_DEF(AST_DELETE);
210 AST_NODE_ACCEPT_VISITOR_DEF(AST_EXT_FUNC_DECL);
211 AST_NODE_ACCEPT_VISITOR_DEF(AST_FIN_STMT_LIST);
212 AST_NODE_ACCEPT_VISITOR_DEF(AST_REC_FUNC_DEF);
213 AST_NODE_ACCEPT_VISITOR_DEF(AST_EXPR_DIRECT_REF);
214 AST_NODE_ACCEPT_VISITOR_DEF(AST_ALLOC_REC_CALL);

```

:

A.3.2 src/ast/ast.h

```

1  #ifndef AST_H
2  #define AST_H
3
4  #include <std_def.h>
5  #include <std_include.h>
6  #include <vector.h>
7
8  typedef struct Symbol_Table Symbol_Table;
9
10 typedef struct Symbol_Table_Node Symbol_Table_Node;
11
12 typedef struct Ast_Visitor Ast_Visitor;
13
14 // Unary expression nodes:
15 #define AST_EXPR_LNOT    expr_lnot    // !expr
16 #define AST_EXPR_ABS     expr_abs     // |expr|
17 // Unary nodes:
18 #define AST_ALLOC_REC    alloc_rec    // allocate rec
19 #define AST_ALLOC_REC_CALL alloc_rec_call // allocate rec of record()
20 #define AST_DELETE       delete_ptr   // delete var
21 #define AST_RETURN_STMT  return_stmt  // return expr
22 #define AST_WRITE_STMT   write_stmt   // write expr
23
24 // Binary expression nodes:
25 #define AST_EXPR_LOR     expr_lor     // expr || expr
26 #define AST_EXPR_LAND    expr_land    // expr && expr
27 #define AST_EXPR_EQ      expr_eq      // expr == expr
28 #define AST_EXPR_NEQ     expr_neq     // expr != expr
29 #define AST_EXPR_GT      expr_gt      // expr > expr
30 #define AST_EXPR_LT      expr_lt      // expr < expr
31 #define AST_EXPR_GTEQ     expr_gteq    // expr >= expr
32 #define AST_EXPR_LTEQ     expr_lteq    // expr <= expr
33 #define AST_EXPR_PLUS     expr_plus    // expr + expr
34 #define AST_EXPR_MINUS    expr_minus   // expr - expr
35 #define AST_EXPR_MUL      expr_mul     // expr * expr
36 #define AST_EXPR_DIV      expr_div     // expr / expr
37 #define AST_EXPR_CAST     expr_cast    // (My_Type)expr
38 #define AST_EXPR_ARY_REF  expr_ary_ref // ary[expr]
39 #define AST_EXPR_DOT_REF  expr_dot_ref // rec.field
40 #define AST_EXPR_DIRECT_REF expr_direct_ref // record[type].field
41 // Binary nodes:
42 #define AST_VAR_DECL      var_decl     // i:int
43 #define AST_TYPE_DEF      type_decl    // type i = int
44 #define AST_ALLOC_ARY     alloc_ary    // allocate a of length 42
45 #define AST_IF_STMT       if_stmt      // if expr then stmt
46 #define AST_WHILE_STMT    while_stmt   // while expr do stmt
47 #define AST_ASSIGNMENT    assignment   // i = 42
48
49 // Ternary nodes:
50 #define AST_IF_ELSE_STMT  if_else_stmt // if ... else ...
51
52 // Function call node:

```

```

53 #define AST_EXPR_FUNC_CALL  expr_func_call  // foo(expr,..., expr)
54
55 // Statement lists:
56 #define AST_STMT_LIST      stmt_list        // list of decl and stmts
57 #define AST_FIN_STMT_LIST  fin_stmt_list    // list of decl and stmts
58 #define AST_REC_STMT_LIST  rec_stmt_list    // list of decl and stmts
59 #define AST_FUNC_BODY      func_body        // decls and stmts in func
60
61 // Function definition:
62 #define AST_FUNC_DEF        func_def         // func foo(e, ..., e) ...
63 // Extern function declaration:
64 #define AST_EXT_FUNC_DECL  ext_func_decl    // extern(C) func foo( ...
65 // Record finalize function definition:
66 #define AST_FIN_FUNC_DEF    fin_func_def     // func finalize()
67 // Record 'record' function definition:
68 #define AST_REC_FUNC_DEF    rec_func_def     // func record(e, ..., e)
69
70 // Nodes with constant values:
71 #define AST_EXPR_INT        expr_int         // 42
72 #define AST_EXPR_BOOL      expr_bool        // true or false
73 #define AST_EXPR_NULL      expr_null        // null
74 #define AST_EXPR_CHAR      expr_char        // 'A'
75 #define AST_EXPR_STRING    expr_string      // "Hello"
76 // Immediate identifier (iden = iden + 1)
77 #define AST_VARIABLE_IDEN   expr_iden
78 #define AST_REC_SELF_PTR    expr_rec_self   // record.foo();
79
80 // Contains module package name.
81 #define AST_PACKAGE_STRING  package_str     // package pack.name;
82 // Contains string for imported module.
83 #define AST_IMPORT_STRING  import_str      // import pack.name.file;
84
85 // Simple type nodes:
86 // int type (var i:int or type i = int)
87 #define AST_SIMPLE_TYPE_INT type_int
88 // bool type (var b:bool or type b = bool)
89 #define AST_SIMPLE_TYPE_BOOL type_bool
90 // char type (var c:char or type c = char)
91 #define AST_SIMPLE_TYPE_CHAR type_char
92 // string type (var s:string or type s = string)
93 #define AST_SIMPLE_TYPE_STRING type_string
94 // void type (func f():void ...)
95 #define AST_SIMPLE_TYPE_VOID type_void
96
97 // Complex type nodes:
98 // Identifier type (var i:iden or type i = iden)
99 #define AST_TYPE_IDEN      type_iden
100 // Array type (var a:array of something or type a = array of something)
101 #define AST_TYPE_ARY       type_ary
102 // Record type (var r:record of {...} or type r = record of {...})
103 #define AST_TYPE_REC       record_decl
104
105 #define __AST_NODE_ACCEPT_VISITOR_FUNC(node_type) \
106   __ast_ ## node_type ## _accept_visitor
107
108 #define AST_NODE_ACCEPT_VISITOR_FUNC(node_type) \
109   __AST_NODE_ACCEPT_VISITOR_FUNC(node_type)
110
111 #define AST_NODE_ACCEPT_VISITOR_DECL(node_type) \
112   void AST_NODE_ACCEPT_VISITOR_FUNC(node_type) (Ast_Node *n, Ast_Visitor *v)
113
114 /* Expression types. What type expressions evaluate into.
115  * Simple expressions like immediate int constants don't
116  * have an Ast_Expr_Type field because we already know its type. */
117 typedef enum Ast_Expr_Type {
118   /* Type not yet set. */
119   AST_EXPR_TYPE_UNSPECIFIED,
120   /* Expression has conflicting operand types so the expression
121    * evaluates to an unknown type. */
122   AST_EXPR_TYPE_UNKNOWN,
123   /* Expression operands evaluate to void. */
124   AST_EXPR_TYPE_VOID,
125   /* Expression operands evaluate to int. */
126   AST_EXPR_TYPE_INT,

```

```

127  /* Expression operands evaluate to bool. */
128  AST_EXPR_TYPE_CHAR,
129  /* Expression operands evaluate to string. */
130  AST_EXPR_TYPE_BOOL,
131  /* Expression operands evaluate to char. */
132  AST_EXPR_TYPE_STRING,
133  /* Expression operands evaluate to record type, used to alloc records. */
134  AST_EXPR_TYPE_REC,
135  /* Expression operands evaluate to array type, used to alloc arrays. */
136  AST_EXPR_TYPE_ARY,
137  /* Expression is the null constant, which can be used to assign
138   * records and arrays. */
139  AST_EXPR_TYPE_NULL // Must be last
140 } Ast_Expr_Type;
141
142 #define AST_EXPR_TYPE_COUNT (AST_EXPR_TYPE_NULL + 1)
143
144 static inline Const_String ast_expr_type_to_string(Ast_Expr_Type t)
145 {
146     extern Const_String const __ast_expr_type_strings[AST_EXPR_TYPE_COUNT];
147     return __ast_expr_type_strings[t];
148 }
149
150 typedef struct Ast_Node Ast_Node;
151
152 typedef struct Ast {
153     Ast_Node *root;
154     String file;
155     String dir;
156     Const_String package;
157     Symbol_Table *symbol_table;
158 } Ast;
159
160 static inline Ast_Node *ast_get_root(Ast *ast)
161 {
162     return ast->root;
163 }
164
165 static inline Ast *ast_alloc(Const_String file_name)
166 {
167     Ast *ast = ALLOC_NEW(Ast);
168     ast->file = string_duplicate(file_name);
169     ast->root = NULL;
170     ast->dir = NULL;
171     ast->package = NULL;
172     ast->symbol_table = NULL;
173     return ast;
174 }
175
176 static inline bool ast_is_valid(Ast *ast)
177 {
178     if (ast)
179         return ast->root;
180     return false;
181 }
182
183 static inline Const_String ast_get_file_name(Ast *ast)
184 {
185     assert(ast);
186     assert(ast->file);
187     return ast->file;
188 }
189
190 Const_String ast_get_dirname(Ast *ast);
191
192 Const_String ast_get_package(Ast *ast);
193
194 /* Returns NULL if ast is invalid. */
195 Symbol_Table *ast_get_symbol_table(Ast *ast);
196
197 Symbol_Table *ast_get_resolve_symbol_table(Ast *ast);
198
199 Symbol_Table *ast_move_symbol_table(Ast *ast);
200

```



```

201 void ast_dump_parse_tree(Ast *ast);
202
203 void ast_dump_parse_tree_graph(Ast *ast);
204
205 void ast_release_nodes(Ast *ast);
206
207 void ast_release_symbol_table(Ast *ast);
208
209 void ast_destroy(Ast *ast);
210
211 typedef void (*Ast_Node_Accept_Visitor)(Ast_Node *this_n, Ast_Visitor *v);
212
213 struct Ast_Node {
214     Ast_Node_Accept_Visitor accept_visitor;
215     Symbol_Table_Node *sym_table_node;
216     File_Location location;
217 };
218
219 static inline Symbol_Table_Node *ast_node_get_symbol_table_node(Ast_Node *n)
220 {
221     return n->sym_table_node;
222 }
223
224 static inline void ast_node_set_symbol_table_node(Ast_Node *n,
225     Symbol_Table_Node *sym_node)
226 {
227     n->sym_table_node = sym_node;
228 }
229
230 static inline File_Location *ast_node_get_file_location(Ast_Node *n)
231 {
232     return &n->location;
233 }
234
235 #define AST_NODE_FIELD ast_node
236
237 #define AST_NODE_STRUCT_BEGIN(name) \
238     typedef struct name { \
239         Ast_Node AST_NODE_FIELD;
240 #define AST_NODE_STRUCT_END(name) } name;
241
242 #define AST_NODE_OF(container) (&(container)->AST_NODE_FIELD)
243
244 #define AST_CONTAINER_OF(ast_node, type) \
245     CONTAINER_OF(ast_node, type, AST_NODE_FIELD)
246
247 #define AST_NODE_INIT(node_type, node_location) ((Ast_Node){ \
248     .accept_visitor = AST_NODE_ACCEPT_VISITOR_FUNC(node_type), \
249     .location = node_location, \
250     .sym_table_node = NULL \
251 })
252
253 #define AST_NODE_ALLOC(type_name, node_type, node_location) ({ \
254     type_name *___n = ALLOC_NEW(type_name); \
255     *AST_NODE_OF(___n) = AST_NODE_INIT(node_type, node_location); \
256     ___n; \
257 })
258
259 // Nodes with two operands.
260 AST_NODE_STRUCT_BEGIN(Ast_Node_Binary)
261     Ast_Node *lhs, *rhs;
262 AST_NODE_STRUCT_END(Ast_Node_Binary)
263
264 #define AST_NODE_BINARY_ALLOC(node_type, node_location, \
265     left_operand, right_operand) ({ \
266     Ast_Node_Binary *___n = AST_NODE_ALLOC(Ast_Node_Binary, \
267     node_type, node_location); \
268     ___n->lhs = left_operand; \
269     ___n->rhs = right_operand; \
270     AST_NODE_OF(___n); \
271 })
272
273 // Nodes with two operands.
274 typedef struct Ast_Expr_Binary {

```

```

275     Ast_Node_Binary bin_node; // Must be first
276     Ast_Expr_Type expr_type;
277 } Ast_Expr_Binary;
278
279 static inline void ast_expr_binary_set_expr_type(Ast_Expr_Binary *n,
280     Ast_Expr_Type t)
281 {
282     n->expr_type = t;
283 }
284
285 static inline Ast_Expr_Type ast_expr_binary_get_expr_type(Ast_Expr_Binary *n)
286 {
287     return n->expr_type;
288 }
289
290 static inline Const_String ast_expr_binary_get_expr_str(Ast_Expr_Binary *n)
291 {
292     Const_String ret =
293         ast_expr_type_to_string(ast_expr_binary_get_expr_type(n));
294     assert(ret);
295     return ret;
296 }
297
298 #define AST_EXPR_BINARY_OF(node_binary) \
299     CONTAINER_OF(node_binary, Ast_Expr_Binary, bin_node)
300
301 #define AST_EXPR_BINARY_ALLOC(node_type, node_location, \
302     left_operand, right_operand) ({ \
303     Ast_Expr_Binary *__n = ALLOC_NEW(Ast_Expr_Binary); \
304     *AST_NODE_OF(&__n->bin_node) = AST_NODE_INIT(node_type, node_location); \
305     __n->bin_node.lhs = left_operand; \
306     __n->bin_node.rhs = right_operand; \
307     __n->expr_type = AST_EXPR_TYPE_UNSPECIFIED; \
308     AST_NODE_OF(&__n->bin_node); \
309 })
310
311 // Nodes with three operands.
312 AST_NODE_STRUCT_BEGIN(Ast_Node_Ternary)
313     Ast_Node *lhs, *mid, *rhs;
314 AST_NODE_STRUCT_END(Ast_Node_Ternary)
315
316 #define AST_NODE_TERNARY_ALLOC(node_type, node_location, \
317     left_operand, mid_operand, right_operand) ({ \
318     Ast_Node_Ternary *__n = AST_NODE_ALLOC(Ast_Node_Ternary, \
319     node_type, node_location); \
320     __n->lhs = left_operand; \
321     __n->mid = mid_operand; \
322     __n->rhs = right_operand; \
323     AST_NODE_OF(__n); \
324 })
325
326 // Nodes with one operand.
327 AST_NODE_STRUCT_BEGIN(Ast_Node_Unary)
328     Ast_Node *expr;
329 AST_NODE_STRUCT_END(Ast_Node_Unary)
330
331 #define AST_NODE_UNARY_ALLOC(node_type, node_location, operand) ({ \
332     Ast_Node_Unary *__n = AST_NODE_ALLOC(Ast_Node_Unary, \
333     node_type, node_location); \
334     __n->expr = operand; \
335     AST_NODE_OF(__n); \
336 })
337
338 // Expression nodes with one operand.
339 typedef struct Ast_Expr_Unary {
340     Ast_Node_Unary una_node; // Must be first
341     Ast_Expr_Type expr_type;
342 } Ast_Expr_Unary;
343
344 static inline void ast_expr_unary_set_expr_type(Ast_Expr_Unary *n,
345     Ast_Expr_Type t)
346 {
347     n->expr_type = t;
348 }

```

```

349
350 static inline Ast_Expr_Type ast_expr_unary_get_expr_type(Ast_Expr_Unary *n)
351 {
352     return n->expr_type;
353 }
354
355 static inline Const_String ast_expr_unary_get_expr_str(Ast_Expr_Unary *n)
356 {
357     Const_String ret =
358         ast_expr_type_to_string(ast_expr_unary_get_expr_type(n));
359     assert(ret);
360     return ret;
361 }
362
363 #define AST_EXPR_UNARY_ALLOC(node_type, node_location, operand) ({ \
364     Ast_Expr_Unary *___n = ALLOC_NEW(Ast_Expr_Unary); \
365     *AST_NODE_OF(&___n->una_node) = AST_NODE_INIT(node_type, node_location); \
366     ___n->una_node.expr = operand; \
367     ___n->expr_type = AST_EXPR_TYPE_UNSPECIFIED; \
368     AST_NODE_OF(&___n->una_node); \
369 })
370
371 #define AST_EXPR_UNARY_OF(node_unary) \
372     CONTAINER_OF(node_unary, Ast_Expr_Unary, una_node)
373
374 typedef struct Symbol_Type_Func Symbol_Type_Func;
375
376 // Function call.
377 AST_NODE_STRUCT_BEGIN(Ast_Expr_Func_Call)
378     Ast_Node *identifier;
379     Vector *arguments;
380     Symbol_Type_Func *func; // set by type checker
381     Ast_Expr_Type expr_type;
382 AST_NODE_STRUCT_END(Ast_Expr_Func_Call)
383
384 static inline Symbol_Type_Func *ast_expr_func_call_get_func(
385     Ast_Expr_Func_Call *c)
386 {
387     return c->func;
388 }
389
390 static inline void ast_expr_func_call_set_expr_type(Ast_Expr_Func_Call *n,
391     Ast_Expr_Type t)
392 {
393     n->expr_type = t;
394 }
395
396 static inline Ast_Expr_Type ast_expr_func_call_get_expr_type(
397     Ast_Expr_Func_Call *n)
398 {
399     return n->expr_type;
400 }
401
402 static inline Const_String ast_expr_func_call_get_expr_str(
403     Ast_Expr_Func_Call *n)
404 {
405     Const_String ret =
406         ast_expr_type_to_string(ast_expr_func_call_get_expr_type(n));
407     assert(ret);
408     return ret;
409 }
410
411 /* Allocate a vector and pass to to this macro.
412  * The ast node will take care of deleting the vector. */
413 #define AST_EXPR_FUNC_CALL_ALLOC(node_type, node_location, iden, vec_args) ({ \
414     Ast_Expr_Func_Call *___n = AST_NODE_ALLOC(Ast_Expr_Func_Call, \
415         node_type, node_location); \
416     ___n->identifier = iden; \
417     ___n->arguments = vec_args; \
418     ___n->expr_type = AST_EXPR_TYPE_UNSPECIFIED; \
419     ___n->func = NULL; \
420     AST_NODE_OF(___n); \
421 })
422

```

```

423 // Record declaration node (record of {...}).
424 AST_NODE_STRUCT_BEGIN(Ast_Type_Rec)
425     Vector *extend_list;
426     Vector *body;
427     Symbol_Table_Node *body_node;
428 AST_NODE_STRUCT_END(Ast_Type_Rec)
429
430 /* Allocate vector arguments and pass them tp the macro.
431 * The AST node will take care of freeing the vectors. */
432 #define AST_TYPE_REC_ALLOC(node_type, node_location, \
433     vec_extend, vec_body) ({ \
434     Ast_Type_Rec *___n = AST_NODE_ALLOC(Ast_Type_Rec, \
435     node_type, node_location); \
436     ___n->extend_list = vec_extend; \
437     ___n->body = vec_body; \
438     AST_NODE_OF(___n); \
439 })
440
441 // Variable identifier node.
442 AST_NODE_STRUCT_BEGIN(Ast_Variable_Iden)
443     String iden;
444     Ast_Expr_Type expr_type;
445 AST_NODE_STRUCT_END(Ast_Variable_Iden)
446
447 static inline void ast_variable_iden_set_expr_type(Ast_Variable_Iden *n,
448     Ast_Expr_Type t)
449 {
450     n->expr_type = t;
451 }
452
453 static inline Ast_Expr_Type ast_variable_iden_get_expr_type(
454     Ast_Variable_Iden *n)
455 {
456     return n->expr_type;
457 }
458
459 static inline Const_String ast_variable_iden_get_expr_str(
460     Ast_Variable_Iden *n)
461 {
462     Const_String ret =
463         ast_expr_type_to_string(ast_variable_iden_get_expr_type(n));
464     assert(ret);
465     return ret;
466 }
467
468 #define AST_VARIABLE_IDEN_ALLOC(node_type, node_location, identifier) ({ \
469     Ast_Variable_Iden *___n = AST_NODE_ALLOC(Ast_Variable_Iden, \
470     node_type, node_location); \
471     ___n->iden = string_duplicate(identifier); \
472     ___n->expr_type = AST_EXPR_TYPE_UNSPECIFIED; \
473     AST_NODE_OF(___n); \
474 })
475
476 // 32 bit integer constant node.
477 AST_NODE_STRUCT_BEGIN(Ast_Expr_Int)
478     int32_t val;
479 AST_NODE_STRUCT_END(Ast_Expr_Int)
480
481 int32_t ast_string_to_int32(Ast_Expr_Int *n, String str);
482
483 #define AST_EXPR_INT_ALLOC(node_type, node_location, string_val) ({ \
484     Ast_Expr_Int *___n = AST_NODE_ALLOC(Ast_Expr_Int, \
485     node_type, node_location); \
486     ___n->val = ast_string_to_int32(___n, string_val); \
487     AST_NODE_OF(___n); \
488 })
489
490 // Boolean constant node.
491 AST_NODE_STRUCT_BEGIN(Ast_Expr_Bool)
492     bool val;
493 AST_NODE_STRUCT_END(Ast_Expr_Bool)
494
495 #define AST_EXPR_BOOL_ALLOC(node_type, node_location, bool_val) ({ \
496     Ast_Expr_Bool *___n = AST_NODE_ALLOC(Ast_Expr_Bool, \

```

```

497         node_type, node_location);          \
498         ____n->val = bool_val;              \
499         AST_NODE_OF(____n);                 \
500     })
501
502     // Char constant ('c')
503     AST_NODE_STRUCT_BEGIN(Ast_Expr_Char)
504         int32_t val;
505     AST_NODE_STRUCT_END(Ast_Expr_Char)
506
507     int32_t ast_char_to_int32(Ast_Expr_Char *n, String char_str);
508
509     #define AST_EXPR_CHAR_ALLOC(node_type, node_location, char_str) ({ \
510         Ast_Expr_Char *____n = AST_NODE_ALLOC(Ast_Expr_Char, \
511         node_type, node_location); \
512         ____n->val = ast_char_to_int32(____n, char_str); \
513         AST_NODE_OF(____n); \
514     })
515
516     // String constant ("my string")
517     AST_NODE_STRUCT_BEGIN(Ast_Expr_String)
518         String val;
519     AST_NODE_STRUCT_END(Ast_Expr_String)
520
521     String ast_get_expr_string(Ast_Expr_String *n, String str);
522
523     #define AST_EXPR_STRING_ALLOC(node_type, node_location, string_str) ({ \
524         Ast_Expr_String *____n = AST_NODE_ALLOC(Ast_Expr_String, \
525         node_type, node_location); \
526         ____n->val = ast_get_expr_string(____n, string_str); \
527         AST_NODE_OF(____n); \
528     })
529
530     String ast_module_string(Const_String initial);
531
532     // Module name (package.name).
533     AST_NODE_STRUCT_BEGIN(Ast_Module_String)
534         String module;
535         int dep_idx;
536     AST_NODE_STRUCT_END(Ast_Module_String)
537
538     #define AST_MODULE_STRING_ALLOC(node_type, node_location, module_str) ({ \
539         Ast_Module_String *____n = AST_NODE_ALLOC(Ast_Module_String, \
540         node_type, node_location); \
541         ____n->module = ast_module_string(module_str); \
542         AST_NODE_OF(____n); \
543     })
544
545     // For nodes without any contents.
546     AST_NODE_STRUCT_BEGIN(Ast_Empty)
547     AST_NODE_STRUCT_END(Ast_Empty)
548
549     #define AST_EMPTY_ALLOC(node_type, node_location) ({ \
550         Ast_Empty *____n = AST_NODE_ALLOC(Ast_Empty, \
551         node_type, node_location); \
552         AST_NODE_OF(____n); \
553     })
554
555     // For null constant node.
556     AST_NODE_STRUCT_BEGIN(Ast_Expr_Null)
557     AST_NODE_STRUCT_END(Ast_Expr_Null)
558
559     #define AST_EXPR_NULL_ALLOC(node_type, node_location) ({ \
560         Ast_Expr_Null *____n = AST_NODE_ALLOC(Ast_Expr_Null, \
561         node_type, node_location); \
562         AST_NODE_OF(____n); \
563     })
564
565     // Node for typedefs type iden = something.
566     AST_NODE_STRUCT_BEGIN(Ast_Type_Iden)
567         String iden;
568     AST_NODE_STRUCT_END(Ast_Type_Iden)
569
570     #define AST_TYPE_IDEN_ALLOC(node_type, node_location, identifier) ({ \

```

```

571     Ast_Type_Iden *___n = AST_NODE_ALLOC(Ast_Type_Iden, \
572         node_type, node_location); \
573     ___n->iden = string_duplicate(identifier); \
574     AST_NODE_OF(___n); \
575 })
576
577 // Nodes for complex types with one child. Only array currently.
578 // (type a = array of something)
579 AST_NODE_STRUCT_BEGIN(Ast_Type)
580     Ast_Node *type;
581 AST_NODE_STRUCT_END(Ast_Type)
582
583 #define AST_TYPE_ALLOC(node_type, node_location, complex_type) ({ \
584     Ast_Type *___n = AST_NODE_ALLOC(Ast_Type, \
585         node_type, node_location); \
586     ___n->type = complex_type; \
587     AST_NODE_OF(___n); \
588 })
589
590 AST_NODE_STRUCT_BEGIN(Ast_Stmt_List)
591     Vector *statements;
592     Uns num_rec_ctor_stmts;
593 AST_NODE_STRUCT_END(Ast_Stmt_List)
594
595 #define AST_STMT_LIST_ALLOC(node_type, node_location, vec_stmts) ({ \
596     Ast_Stmt_List *___n = AST_NODE_ALLOC(Ast_Stmt_List, \
597         node_type, node_location); \
598     ___n->statements = vec_stmts; \
599     ___n->num_rec_ctor_stmts = 0; \
600     AST_NODE_OF(___n); \
601 })
602
603 // Function definition node
604 // (func foo():something ... end foo) also works as:
605 // Extern function declaration node
606 // (extern (C) func foo():something; )
607 AST_NODE_STRUCT_BEGIN(Ast_Func_Def)
608     Ast_Node *iden;
609     Ast_Node *extern_type;
610     Ast_Node *return_type;
611     Vector *parameters;
612     Ast_Node *statements;
613 AST_NODE_STRUCT_END(Ast_Func_Def)
614
615 #define AST_FUNC_DEF_ALLOC(node_type, node_location, ext_type, identifier, \
616     type, vec_params, stmts) ({ \
617     Ast_Func_Def *___n = AST_NODE_ALLOC(Ast_Func_Def, \
618         node_type, node_location); \
619     ___n->iden = identifier; \
620     ___n->extern_type = ext_type; \
621     ___n->return_type = type; \
622     ___n->parameters = vec_params; \
623     ___n->statements = stmts; \
624     AST_NODE_OF(___n); \
625 })
626
627 typedef struct Symbol_Type_Struct Symbol_Type_Struct;
628
629 Ast_Expr_Type symbol_type_to_expr_type(Symbol_Type_Struct *st);
630
631 AST_NODE_ACCEPT_VISITOR_DECL(AST_EXPR_LOR);
632 AST_NODE_ACCEPT_VISITOR_DECL(AST_EXPR_LAND);
633 AST_NODE_ACCEPT_VISITOR_DECL(AST_EXPR_EQ);
634 AST_NODE_ACCEPT_VISITOR_DECL(AST_EXPR_NEQ);
635 AST_NODE_ACCEPT_VISITOR_DECL(AST_EXPR_GT);
636 AST_NODE_ACCEPT_VISITOR_DECL(AST_EXPR_LT);
637 AST_NODE_ACCEPT_VISITOR_DECL(AST_EXPR_GTEQ);
638 AST_NODE_ACCEPT_VISITOR_DECL(AST_EXPR_LTEQ);
639 AST_NODE_ACCEPT_VISITOR_DECL(AST_EXPR_PLUS);
640 AST_NODE_ACCEPT_VISITOR_DECL(AST_EXPR_MINUS);
641 AST_NODE_ACCEPT_VISITOR_DECL(AST_EXPR_MUL);
642 AST_NODE_ACCEPT_VISITOR_DECL(AST_EXPR_DIV);
643 AST_NODE_ACCEPT_VISITOR_DECL(AST_EXPR_CAST);
644 AST_NODE_ACCEPT_VISITOR_DECL(AST_EXPR_ARY_REF);

```

```

645 AST_NODE_ACCEPT_VISITOR_DECL(AST_EXPR_DOT_REF);
646 AST_NODE_ACCEPT_VISITOR_DECL(AST_EXPR_LNOT);
647 AST_NODE_ACCEPT_VISITOR_DECL(AST_EXPR_ABS);
648 AST_NODE_ACCEPT_VISITOR_DECL(AST_EXPR_FUNC_CALL);
649 AST_NODE_ACCEPT_VISITOR_DECL(AST_VARIABLE_IDEN);
650 AST_NODE_ACCEPT_VISITOR_DECL(AST_EXPR_INT);
651 AST_NODE_ACCEPT_VISITOR_DECL(AST_EXPR_BOOL);
652 AST_NODE_ACCEPT_VISITOR_DECL(AST_EXPR_NULL);
653 AST_NODE_ACCEPT_VISITOR_DECL(AST_SIMPLE_TYPE_INT);
654 AST_NODE_ACCEPT_VISITOR_DECL(AST_SIMPLE_TYPE_BOOL);
655 AST_NODE_ACCEPT_VISITOR_DECL(AST_TYPE_IDEN);
656 AST_NODE_ACCEPT_VISITOR_DECL(AST_VAR_DECL);
657 AST_NODE_ACCEPT_VISITOR_DECL(AST_TYPE_DEF);
658 AST_NODE_ACCEPT_VISITOR_DECL(AST_TYPE_ARY);
659 AST_NODE_ACCEPT_VISITOR_DECL(AST_TYPE_REC);
660 AST_NODE_ACCEPT_VISITOR_DECL(AST_STMT_LIST);
661 AST_NODE_ACCEPT_VISITOR_DECL(AST_FUNC_DEF);
662 AST_NODE_ACCEPT_VISITOR_DECL(AST_IF_STMT);
663 AST_NODE_ACCEPT_VISITOR_DECL(AST_IF_ELSE_STMT);
664 AST_NODE_ACCEPT_VISITOR_DECL(AST_ALLOC_REC);
665 AST_NODE_ACCEPT_VISITOR_DECL(AST_ALLOC_ARY);
666 AST_NODE_ACCEPT_VISITOR_DECL(AST_WHILE_STMT);
667 AST_NODE_ACCEPT_VISITOR_DECL(AST_RETURN_STMT);
668 AST_NODE_ACCEPT_VISITOR_DECL(AST_WRITE_STMT);
669 AST_NODE_ACCEPT_VISITOR_DECL(AST_ASSIGNMENT);
670 AST_NODE_ACCEPT_VISITOR_DECL(AST_FUNC_BODY);
671 AST_NODE_ACCEPT_VISITOR_DECL(AST_EXPR_CHAR);
672 AST_NODE_ACCEPT_VISITOR_DECL(AST_SIMPLE_TYPE_CHAR);
673 AST_NODE_ACCEPT_VISITOR_DECL(AST_EXPR_STRING);
674 AST_NODE_ACCEPT_VISITOR_DECL(AST_SIMPLE_TYPE_STRING);
675 AST_NODE_ACCEPT_VISITOR_DECL(AST_IMPORT_STRING);
676 AST_NODE_ACCEPT_VISITOR_DECL(AST_PACKAGE_STRING);
677 AST_NODE_ACCEPT_VISITOR_DECL(AST_REC_SELF_PTR);
678 AST_NODE_ACCEPT_VISITOR_DECL(AST_SIMPLE_TYPE_VOID);
679 AST_NODE_ACCEPT_VISITOR_DECL(AST_FIN_FUNC_DEF);
680 AST_NODE_ACCEPT_VISITOR_DECL(AST_DELETE);
681 AST_NODE_ACCEPT_VISITOR_DECL(AST_EXT_FUNC_DECL);
682 AST_NODE_ACCEPT_VISITOR_DECL(AST_FIN_STMT_LIST);
683 AST_NODE_ACCEPT_VISITOR_DECL(AST_REC_FUNC_DEF);
684 AST_NODE_ACCEPT_VISITOR_DECL(AST_EXPR_DIRECT_REF);
685 AST_NODE_ACCEPT_VISITOR_DECL(AST_ALLOC_REC_CALL);
686
687 #endif // AST_H

```

:

A.3.3 src/ast/ast_string.c

```

1  #include "ast.h"
2  #include <errno.h>
3  #include "ast_string.h"
4
5  int32_t ast_string_to_int32(Ast_Expr_Int *n, String str)
6  {
7      errno = 0;
8      int32_t res = string_base10_to_int32(str);
9      if (errno) {
10         if (errno == EINVAL) {
11             /* Probably cannot happen. */
12             report_error_location(&n->ast_node.location,
13                 S("unable to convert " QFY("%S") " to an integer.\n"),
14                 str);
15         } else if (errno == ERANGE) {
16             report_error_location(&n->ast_node.location,
17                 S("value " QFY("%S") " %s int\n"),
18                 str,
19                 res == INT32_MAX ? "overflows" : "underflows");
20         } else {
21             /* Probably cannot happen. */
22             fatal_error(S("unexpected error converting string "
23                 "to integer.\n"));

```

```

24     }
25     res = 0;
26 }
27 return res;
28 }
29
30 /* Warning. Make sure string is dynamic before calling this function. */
31 bool __ast_string_unescape(Ast_Node *n, String s)
32 {
33     bool ret = true;
34     char *cstr = s->str;
35     char *endstr = cstr;
36     for (; *endstr; endstr++, cstr++) {
37         if (*endstr == '\\') {
38             ++endstr;
39             switch (*endstr) {
40                 case 'n':
41                     *cstr = '\n';
42                     break;
43                 case 't':
44                     *cstr = '\t';
45                     break;
46                 case '\\':
47                     *cstr = '\\';
48                     break;
49                 case '"':
50                     *cstr = '"';
51                     break;
52                 case '\':
53                     *cstr = '\';
54                     break;
55                 default:
56                     ret = false;
57                     if (*endstr) {
58                         report_error_location(&n->location,
59                             S("Invalid escape character " QFY("%c") "\n"),
60                             *endstr);
61                     } else {
62                         // Should never happen.
63                         report_error_location(&n->location,
64                             S("Invalid escape character " QFY("(null)") "\n"));
65                         goto out;
66                     }
67                     break;
68             }
69         } else {
70             *cstr = *endstr;
71         }
72     }
73 out:
74     *cstr = *endstr;
75     return ret;
76 }
77
78 #define STRING_ESCAPE_CHAR(c) (('\\' << 8) + (Int)c)
79
80 #include <debug.h>
81
82 void ast_string_escape(String s, String_Builder *sb)
83 {
84     string_builder_assign(sb, S(""));
85     char *cstr = s->str;
86     while (*cstr) {
87         switch (*cstr) {
88             case '\n':
89                 string_builder_append_int16(sb, STRING_ESCAPE_CHAR('n'));
90                 break;
91             case '\t':
92                 string_builder_append_int16(sb, STRING_ESCAPE_CHAR('t'));
93                 break;
94             case '\\':
95                 string_builder_append_int16(sb, STRING_ESCAPE_CHAR('\\'));
96                 break;
97             case '"':

```



```

98         string_builder_append_int16(sb, STRING_ESCAPE_CHAR('"'));
99         break;
100     case '\\':
101         string_builder_append_int16(sb, STRING_ESCAPE_CHAR('\\'));
102         break;
103     default:
104         string_builder_append_char(sb, *cstr);
105         break;
106     }
107     ++cstr;
108 }
109 }

```

:

A.3.4 src/ast/ast_string.h

```

1  #ifndef AST_STRING_H
2  #define AST_STRING_H
3
4  #include <ast/ast.h>
5  #include <string_builder.h>
6
7  typedef struct String_Builder String_Builder;
8
9  int32_t ast_string_to_int32(Ast_Expr_Int *n, String str);
10
11  /* Warning. Make sure string is dynamic before calling this function. That is:
12   * s->dynamic == true. Use string_assign(s, s) to make a string (s) dynamic.
13   * Returns true if no error was reported. */
14  bool __ast_string_unescape(Ast_Node *n, String s);
15
16  void ast_string_escape(String s, String_Builder *sb);
17
18  String ast_module_string(Const_String initial);
19
20 #endif // AST_STRING_H

```

:

A.3.5 src/ast/ast_visitor.h

```

1  #ifndef AST_VISITOR_H
2  #define AST_VISITOR_H
3
4  #include "ast.h"
5
6  typedef struct Ast_Visitor Ast_Visitor;
7
8  #define __AST_VISITOR_FUNC(node_type) __visit_ ## node_type
9  #define AST_VISITOR_FUNC(node_type) __AST_VISITOR_FUNC(node_type)
10
11  #define AST_VISITOR_FUNC_DECL(node_type, visitor_param, node_param) \
12  static void AST_VISITOR_FUNC(node_type)(Ast_Visitor *visitor_param, \
13  Ast_Node *node_param)
14
15  #define AST_VISITOR_FUNC_UNUSED(node_type) \
16  AST_VISITOR_FUNC_DECL(node_type, v, n) { (void)v; (void)n; }
17
18  #define __AST_VISITOR_FUNC_BEGIN(node_type, \
19  visitor_type, visitor_param, \
20  node_struct_type, node_param) \
21  AST_VISITOR_FUNC_DECL(node_type, __ ## visitor_param, __ ## node_param) \
22  { \
23  visitor_type *visitor_param = CONTAINER_OF(__ ## visitor_param, \
24  visitor_type, AST_VISITOR_FIELD); \
25  node_struct_type *node_param = CONTAINER_OF(__ ## node_param, \

```

```

26     node_struct_type, AST_NODE_FIELD);
27
28 #define AST_VISITOR_FUNC_BEGIN(node_type,          \
29     visitor_type, visitor_param,                  \
30     node_struct_type, node_param)                \
31     ___AST_VISITOR_FUNC_BEGIN(node_type, visitor_type, visitor_param, \
32     node_struct_type, node_param)
33
34 #define AST_VISITOR_FUNC_END }
35
36 #define ASTVF_BEGIN AST_VISITOR_FUNC_BEGIN
37 #define ASTVF_END AST_VISITOR_FUNC_END
38 #define ASTVF_UNUSED AST_VISITOR_FUNC_UNUSED
39
40 #define AST_VISITOR_CONTAINER_OF(node, type) \
41     CONTAINER_OF(node, type, AST_VISITOR_FIELD)
42
43 typedef void (*Ast_Visitor_Method)(Ast_Visitor *, Ast_Node *);
44
45 struct Ast_Visitor {
46     Ast_Visitor_Method AST_VISITOR_FUNC(AST_EXPR_LOR);
47     Ast_Visitor_Method AST_VISITOR_FUNC(AST_EXPR_LAND);
48     Ast_Visitor_Method AST_VISITOR_FUNC(AST_EXPR_EQ);
49     Ast_Visitor_Method AST_VISITOR_FUNC(AST_EXPR_NEQ);
50     Ast_Visitor_Method AST_VISITOR_FUNC(AST_EXPR_GT);
51     Ast_Visitor_Method AST_VISITOR_FUNC(AST_EXPR_LT);
52     Ast_Visitor_Method AST_VISITOR_FUNC(AST_EXPR_GTEQ);
53     Ast_Visitor_Method AST_VISITOR_FUNC(AST_EXPR_LTEQ);
54     Ast_Visitor_Method AST_VISITOR_FUNC(AST_EXPR_PLUS);
55     Ast_Visitor_Method AST_VISITOR_FUNC(AST_EXPR_MINUS);
56     Ast_Visitor_Method AST_VISITOR_FUNC(AST_EXPR_MUL);
57     Ast_Visitor_Method AST_VISITOR_FUNC(AST_EXPR_DIV);
58     Ast_Visitor_Method AST_VISITOR_FUNC(AST_EXPR_CAST);
59     Ast_Visitor_Method AST_VISITOR_FUNC(AST_EXPR_LNOT);
60     Ast_Visitor_Method AST_VISITOR_FUNC(AST_EXPR_ABS);
61     Ast_Visitor_Method AST_VISITOR_FUNC(AST_EXPR_INT);
62     Ast_Visitor_Method AST_VISITOR_FUNC(AST_EXPR_BOOL);
63     Ast_Visitor_Method AST_VISITOR_FUNC(AST_EXPR_NULL);
64     Ast_Visitor_Method AST_VISITOR_FUNC(AST_VARIABLE_IDEN);
65     Ast_Visitor_Method AST_VISITOR_FUNC(AST_EXPR_ARY_REF);
66     Ast_Visitor_Method AST_VISITOR_FUNC(AST_EXPR_FUNC_CALL);
67     Ast_Visitor_Method AST_VISITOR_FUNC(AST_EXPR_DOT_REF);
68     Ast_Visitor_Method AST_VISITOR_FUNC(AST_SIMPLE_TYPE_INT);
69     Ast_Visitor_Method AST_VISITOR_FUNC(AST_SIMPLE_TYPE_BOOL);
70     Ast_Visitor_Method AST_VISITOR_FUNC(AST_TYPE_IDEN);
71     Ast_Visitor_Method AST_VISITOR_FUNC(AST_VAR_DECL);
72     Ast_Visitor_Method AST_VISITOR_FUNC(AST_TYPE_DEF);
73     Ast_Visitor_Method AST_VISITOR_FUNC(AST_TYPE_ARY);
74     Ast_Visitor_Method AST_VISITOR_FUNC(AST_TYPE_REC);
75     Ast_Visitor_Method AST_VISITOR_FUNC(AST_STMT_LIST);
76     Ast_Visitor_Method AST_VISITOR_FUNC(AST_FUNC_DEF);
77     Ast_Visitor_Method AST_VISITOR_FUNC(AST_IF_STMT);
78     Ast_Visitor_Method AST_VISITOR_FUNC(AST_IF_ELSE_STMT);
79     Ast_Visitor_Method AST_VISITOR_FUNC(AST_ALLOC_REC);
80     Ast_Visitor_Method AST_VISITOR_FUNC(AST_ALLOC_ARY);
81     Ast_Visitor_Method AST_VISITOR_FUNC(AST_WHILE_STMT);
82     Ast_Visitor_Method AST_VISITOR_FUNC(AST_RETURN_STMT);
83     Ast_Visitor_Method AST_VISITOR_FUNC(AST_WRITE_STMT);
84     Ast_Visitor_Method AST_VISITOR_FUNC(AST_ASSIGNMENT);
85     Ast_Visitor_Method AST_VISITOR_FUNC(AST_FUNC_BODY);
86     Ast_Visitor_Method AST_VISITOR_FUNC(AST_EXPR_CHAR);
87     Ast_Visitor_Method AST_VISITOR_FUNC(AST_SIMPLE_TYPE_CHAR);
88     Ast_Visitor_Method AST_VISITOR_FUNC(AST_EXPR_STRING);
89     Ast_Visitor_Method AST_VISITOR_FUNC(AST_IMPORT_STRING);
90     Ast_Visitor_Method AST_VISITOR_FUNC(AST_SIMPLE_TYPE_STRING);
91     Ast_Visitor_Method AST_VISITOR_FUNC(AST_PACKAGE_STRING);
92     Ast_Visitor_Method AST_VISITOR_FUNC(AST_REC_SELF_PTR);
93     Ast_Visitor_Method AST_VISITOR_FUNC(AST_SIMPLE_TYPE_VOID);
94     Ast_Visitor_Method AST_VISITOR_FUNC(AST_FIN_FUNC_DEF);
95     Ast_Visitor_Method AST_VISITOR_FUNC(AST_DELETE);
96     Ast_Visitor_Method AST_VISITOR_FUNC(AST_EXT_FUNC_DECL);
97     Ast_Visitor_Method AST_VISITOR_FUNC(AST_FIN_STMT_LIST);
98     Ast_Visitor_Method AST_VISITOR_FUNC(AST_REC_FUNC_DEF);
99     Ast_Visitor_Method AST_VISITOR_FUNC(AST_EXPR_DIRECT_REF);

```

```

100     Ast_Visitor_Method AST_VISITOR_FUNC(AST_ALLOC_REC_CALL);
101 };
102
103 #define AST_VISITOR_STATIC_INIT() { \
104     .AST_VISITOR_FUNC(AST_EXPR_LOR) = \
105     AST_VISITOR_FUNC(AST_EXPR_LOR), \
106     .AST_VISITOR_FUNC(AST_EXPR_LAND) = \
107     AST_VISITOR_FUNC(AST_EXPR_LAND), \
108     .AST_VISITOR_FUNC(AST_EXPR_EQ) = \
109     AST_VISITOR_FUNC(AST_EXPR_EQ), \
110     .AST_VISITOR_FUNC(AST_EXPR_NEQ) = \
111     AST_VISITOR_FUNC(AST_EXPR_NEQ), \
112     .AST_VISITOR_FUNC(AST_EXPR_GT) = \
113     AST_VISITOR_FUNC(AST_EXPR_GT), \
114     .AST_VISITOR_FUNC(AST_EXPR_LT) = \
115     AST_VISITOR_FUNC(AST_EXPR_LT), \
116     .AST_VISITOR_FUNC(AST_EXPR_GTEQ) = \
117     AST_VISITOR_FUNC(AST_EXPR_GTEQ), \
118     .AST_VISITOR_FUNC(AST_EXPR_LTEQ) = \
119     AST_VISITOR_FUNC(AST_EXPR_LTEQ), \
120     .AST_VISITOR_FUNC(AST_EXPR_PLUS) = \
121     AST_VISITOR_FUNC(AST_EXPR_PLUS), \
122     .AST_VISITOR_FUNC(AST_EXPR_MINUS) = \
123     AST_VISITOR_FUNC(AST_EXPR_MINUS), \
124     .AST_VISITOR_FUNC(AST_EXPR_MUL) = \
125     AST_VISITOR_FUNC(AST_EXPR_MUL), \
126     .AST_VISITOR_FUNC(AST_EXPR_DIV) = \
127     AST_VISITOR_FUNC(AST_EXPR_DIV), \
128     .AST_VISITOR_FUNC(AST_EXPR_CAST) = \
129     AST_VISITOR_FUNC(AST_EXPR_CAST), \
130     .AST_VISITOR_FUNC(AST_EXPR_LNOT) = \
131     AST_VISITOR_FUNC(AST_EXPR_LNOT), \
132     .AST_VISITOR_FUNC(AST_EXPR_ABS) = \
133     AST_VISITOR_FUNC(AST_EXPR_ABS), \
134     .AST_VISITOR_FUNC(AST_EXPR_INT) = \
135     AST_VISITOR_FUNC(AST_EXPR_INT), \
136     .AST_VISITOR_FUNC(AST_EXPR_BOOL) = \
137     AST_VISITOR_FUNC(AST_EXPR_BOOL), \
138     .AST_VISITOR_FUNC(AST_EXPR_NULL) = \
139     AST_VISITOR_FUNC(AST_EXPR_NULL), \
140     .AST_VISITOR_FUNC(AST_VARIABLE_IDEN) = \
141     AST_VISITOR_FUNC(AST_VARIABLE_IDEN), \
142     .AST_VISITOR_FUNC(AST_EXPR_ARY_REF) = \
143     AST_VISITOR_FUNC(AST_EXPR_ARY_REF), \
144     .AST_VISITOR_FUNC(AST_EXPR_FUNC_CALL) = \
145     AST_VISITOR_FUNC(AST_EXPR_FUNC_CALL), \
146     .AST_VISITOR_FUNC(AST_EXPR_DOT_REF) = \
147     AST_VISITOR_FUNC(AST_EXPR_DOT_REF), \
148     .AST_VISITOR_FUNC(AST_SIMPLE_TYPE_INT) = \
149     AST_VISITOR_FUNC(AST_SIMPLE_TYPE_INT), \
150     .AST_VISITOR_FUNC(AST_SIMPLE_TYPE_BOOL) = \
151     AST_VISITOR_FUNC(AST_SIMPLE_TYPE_BOOL), \
152     .AST_VISITOR_FUNC(AST_TYPE_IDEN) = \
153     AST_VISITOR_FUNC(AST_TYPE_IDEN), \
154     .AST_VISITOR_FUNC(AST_VAR_DECL) = \
155     AST_VISITOR_FUNC(AST_VAR_DECL), \
156     .AST_VISITOR_FUNC(AST_TYPE_DEF) = \
157     AST_VISITOR_FUNC(AST_TYPE_DEF), \
158     .AST_VISITOR_FUNC(AST_TYPE_ARY) = \
159     AST_VISITOR_FUNC(AST_TYPE_ARY), \
160     .AST_VISITOR_FUNC(AST_TYPE_REC) = \
161     AST_VISITOR_FUNC(AST_TYPE_REC), \
162     .AST_VISITOR_FUNC(AST_STMT_LIST) = \
163     AST_VISITOR_FUNC(AST_STMT_LIST), \
164     .AST_VISITOR_FUNC(AST_FUNC_DEF) = \
165     AST_VISITOR_FUNC(AST_FUNC_DEF), \
166     .AST_VISITOR_FUNC(AST_IF_STMT) = \
167     AST_VISITOR_FUNC(AST_IF_STMT), \
168     .AST_VISITOR_FUNC(AST_IF_ELSE_STMT) = \
169     AST_VISITOR_FUNC(AST_IF_ELSE_STMT), \
170     .AST_VISITOR_FUNC(AST_ALLOC_REC) = \
171     AST_VISITOR_FUNC(AST_ALLOC_REC), \
172     .AST_VISITOR_FUNC(AST_ALLOC_ARY) = \
173     AST_VISITOR_FUNC(AST_ALLOC_ARY), \

```

```

174 .AST_VISITOR_FUNC(AST_WHILE_STMT) = \
175     AST_VISITOR_FUNC(AST_WHILE_STMT), \
176 .AST_VISITOR_FUNC(AST_RETURN_STMT) = \
177     AST_VISITOR_FUNC(AST_RETURN_STMT), \
178 .AST_VISITOR_FUNC(AST_WRITE_STMT) = \
179     AST_VISITOR_FUNC(AST_WRITE_STMT), \
180 .AST_VISITOR_FUNC(AST_ASSIGNMENT) = \
181     AST_VISITOR_FUNC(AST_ASSIGNMENT), \
182 .AST_VISITOR_FUNC(AST_FUNC_BODY) = \
183     AST_VISITOR_FUNC(AST_FUNC_BODY), \
184 .AST_VISITOR_FUNC(AST_EXPR_CHAR) = \
185     AST_VISITOR_FUNC(AST_EXPR_CHAR), \
186 .AST_VISITOR_FUNC(AST_SIMPLE_TYPE_CHAR) = \
187     AST_VISITOR_FUNC(AST_SIMPLE_TYPE_CHAR), \
188 .AST_VISITOR_FUNC(AST_EXPR_STRING) = \
189     AST_VISITOR_FUNC(AST_EXPR_STRING), \
190 .AST_VISITOR_FUNC(AST_IMPORT_STRING) = \
191     AST_VISITOR_FUNC(AST_IMPORT_STRING), \
192 .AST_VISITOR_FUNC(AST_SIMPLE_TYPE_STRING) = \
193     AST_VISITOR_FUNC(AST_SIMPLE_TYPE_STRING), \
194 .AST_VISITOR_FUNC(AST_REC_SELF_PTR) = \
195     AST_VISITOR_FUNC(AST_REC_SELF_PTR), \
196 .AST_VISITOR_FUNC(AST_SIMPLE_TYPE_VOID) = \
197     AST_VISITOR_FUNC(AST_SIMPLE_TYPE_VOID), \
198 .AST_VISITOR_FUNC(AST_FIN_FUNC_DEF) = \
199     AST_VISITOR_FUNC(AST_FIN_FUNC_DEF), \
200 .AST_VISITOR_FUNC(AST_DELETE) = \
201     AST_VISITOR_FUNC(AST_DELETE), \
202 .AST_VISITOR_FUNC(AST_EXT_FUNC_DECL) = \
203     AST_VISITOR_FUNC(AST_EXT_FUNC_DECL), \
204 .AST_VISITOR_FUNC(AST_FIN_STMT_LIST) = \
205     AST_VISITOR_FUNC(AST_FIN_STMT_LIST), \
206 .AST_VISITOR_FUNC(AST_REC_FUNC_DEF) = \
207     AST_VISITOR_FUNC(AST_REC_FUNC_DEF), \
208 .AST_VISITOR_FUNC(AST_EXPR_DIRECT_REF) = \
209     AST_VISITOR_FUNC(AST_EXPR_DIRECT_REF), \
210 .AST_VISITOR_FUNC(AST_ALLOC_REC_CALL) = \
211     AST_VISITOR_FUNC(AST_ALLOC_REC_CALL), \
212 .AST_VISITOR_FUNC(AST_PACKAGE_STRING) = \
213     AST_VISITOR_FUNC(AST_PACKAGE_STRING) \
214 }
215
216 #define AST_VISITOR_INIT() ((Ast_Visitor)AST_VISITOR_STATIC_INIT())
217
218 #define AST_VISITOR_FIELD ast_visitor
219
220 #define AST_VISITOR_STRUCT_BEGIN(visitor_name) \
221     typedef struct visitor_name { \
222         Ast_Visitor AST_VISITOR_FIELD;
223 #define AST_VISITOR_STRUCT_END(visitor_name) } visitor_name;
224
225 #define AST_VISITOR_OF(container) (&(container)->AST_VISITOR_FIELD)
226
227 #endif // AST_VISITOR_H

```

:

A.3.6 src/ast/ast_visitor_print.c

```

1  #include "ast_visitor_print.h"
2  #include "ast_string.h"
3  #include <std_def.h>
4  #include <std_include.h>
5
6  AST_VISITOR_STRUCT_BEGIN(Ast_Visitor_Print)
7      Uns indentation;
8      FILE *output_file;
9  AST_VISITOR_STRUCT_END(Ast_Visitor_Print)
10
11 #define INDENT_COUNT 4
12

```

```

13 #define PRINT_OPEN(stream, fmt, indent, ...) \
14 file_print_message(stream, S("%*s<" fmt ">\n"), indent, "", ## __VA_ARGS__)
15
16 #define PRINT_CLOSE(stream, fmt, indent, ...) \
17 file_print_message(stream, S("%*s</" fmt ">\n"), indent, "", ## __VA_ARGS__)
18
19 #define PRINT_SINGLE(stream, fmt, indent, ...) \
20 file_print_message(stream, S("%*s<" fmt " /\n"), indent, "", ## __VA_ARGS__)
21
22 static void stmt_list_action(Ast_Visitor_Print *v, Ast_Stmt_List *n,
23     const char *type)
24 {
25     Ast_Node *stmt;
26     Vector *vec = n->statements;
27
28     PRINT_OPEN(v->output_file, "%s line=\"%U\" column=\"%U\"",
29         v->indentation,
30         type,
31         n->ast_node.location.line,
32         n->ast_node.location.column);
33
34     v->indentation += INDENT_COUNT;
35     VECTOR_FOR_EACH_ENTRY(vec, stmt)
36         stmt->accept_visitor(stmt, AST_VISITOR_OF(v));
37     v->indentation -= INDENT_COUNT;
38
39     PRINT_CLOSE(v->output_file, "%s", v->indentation, type);
40 }
41
42 static void ternary_action(Ast_Visitor_Print *v, Ast_Node_Ternary *n,
43     const char *name)
44 {
45     PRINT_OPEN(v->output_file, "%s line=\"%U\" column=\"%U\"",
46         v->indentation,
47         name,
48         n->ast_node.location.line,
49         n->ast_node.location.column);
50
51     v->indentation += INDENT_COUNT;
52     n->lhs->accept_visitor(n->lhs, AST_VISITOR_OF(v));
53     n->mid->accept_visitor(n->mid, AST_VISITOR_OF(v));
54     n->rhs->accept_visitor(n->rhs, AST_VISITOR_OF(v));
55     v->indentation -= INDENT_COUNT;
56
57     PRINT_CLOSE(v->output_file, "%s", v->indentation, name);
58 }
59
60 static void binary_action(Ast_Visitor_Print *v, Ast_Node_Binary *n,
61     const char *name)
62 {
63     PRINT_OPEN(v->output_file, "%s line=\"%U\" column=\"%U\"",
64         v->indentation,
65         name,
66         n->ast_node.location.line,
67         n->ast_node.location.column);
68
69     v->indentation += INDENT_COUNT;
70     n->lhs->accept_visitor(n->lhs, AST_VISITOR_OF(v));
71     n->rhs->accept_visitor(n->rhs, AST_VISITOR_OF(v));
72     v->indentation -= INDENT_COUNT;
73
74     PRINT_CLOSE(v->output_file, "%s", v->indentation, name);
75 }
76
77 static void binary_op_action(Ast_Visitor_Print *v, Ast_Node_Binary *n,
78     const char *name)
79 {
80     Ast_Expr_Binary *bin = AST_EXPR_BINARY_OF(n);
81     PRINT_OPEN(v->output_file, "%s type=\"%S\" line=\"%U\" column=\"%U\"",
82         v->indentation,
83         name,
84         ast_expr_binary_get_expr_str(bin),
85         n->ast_node.location.line,
86         n->ast_node.location.column);

```

```

87
88     v->indentation += INDENT_COUNT;
89     if (n->lhs) // Because of DIRECT_REF
90         n->lhs->accept_visitor(n->lhs, AST_VISITOR_OF(v));
91     n->rhs->accept_visitor(n->rhs, AST_VISITOR_OF(v));
92     v->indentation -= INDENT_COUNT;
93
94     PRINT_CLOSE(v->output_file, "%s", v->indentation, name);
95 }
96
97 static void unary_action(Ast_Visitor_Print *v, Ast_Node_Unary *n,
98     const char *name)
99 {
100     PRINT_OPEN(v->output_file, "%s line=\"%U\" column=\"%U\"",
101         v->indentation,
102         name,
103         n->ast_node.location.line,
104         n->ast_node.location.column);
105
106     v->indentation += INDENT_COUNT;
107     n->expr->accept_visitor(n->expr, AST_VISITOR_OF(v));
108     v->indentation -= INDENT_COUNT;
109
110     PRINT_CLOSE(v->output_file, "%s", v->indentation, name);
111 }
112
113 static void unary_op_action(Ast_Visitor_Print *v, Ast_Node_Unary *n,
114     const char *name)
115 {
116     Ast_Expr_Unary *una = AST_EXPR_UNARY_OF(n);
117     PRINT_OPEN(v->output_file, "%s type=\"%S\" line=\"%U\" column=\"%U\"",
118         v->indentation,
119         name,
120         ast_expr_unary_get_expr_str(una),
121         n->ast_node.location.line,
122         n->ast_node.location.column);
123
124     v->indentation += INDENT_COUNT;
125     n->expr->accept_visitor(n->expr, AST_VISITOR_OF(v));
126     v->indentation -= INDENT_COUNT;
127
128     PRINT_CLOSE(v->output_file, "%s", v->indentation, name);
129 }
130
131 static void simple_type_action(Ast_Visitor_Print *v, Ast_Empty *n,
132     const char *type)
133 {
134     PRINT_SINGLE(v->output_file, "simple_type type=\"%S\" "
135         "line=\"%U\" column=\"%U\"",
136         v->indentation,
137         type,
138         n->ast_node.location.line,
139         n->ast_node.location.column);
140 }
141
142 static void type_action(Ast_Visitor_Print *v, Ast_Type *n, const char *type)
143 {
144     PRINT_OPEN(v->output_file, "%s line=\"%U\" column=\"%U\"",
145         v->indentation,
146         type,
147         n->ast_node.location.line,
148         n->ast_node.location.column);
149
150     v->indentation += INDENT_COUNT;
151     n->type->accept_visitor(n->type, AST_VISITOR_OF(v));
152     v->indentation -= INDENT_COUNT;
153
154     PRINT_CLOSE(v->output_file, "%s", v->indentation, type);
155 }
156
157 ASTVF_BEGIN(AST_EXPR_LOR, Ast_Visitor_Print, v, Ast_Node_Binary, n)
158     binary_op_action(v, n, "expr_logic_or");
159 ASTVF_END
160

```

```

161 ASTVF_BEGIN(AST_EXPR_AND, Ast_Visitor_Print, v, Ast_Node_Binary, n)
162     binary_op_action(v, n, "expr_logic_and");
163 ASTVF_END
164
165 ASTVF_BEGIN(AST_EXPR_EQ, Ast_Visitor_Print, v, Ast_Node_Binary, n)
166     binary_op_action(v, n, "expr_eq");
167 ASTVF_END
168
169 ASTVF_BEGIN(AST_EXPR_NEQ, Ast_Visitor_Print, v, Ast_Node_Binary, n)
170     binary_op_action(v, n, "expr_neq");
171 ASTVF_END
172
173 ASTVF_BEGIN(AST_EXPR_GT, Ast_Visitor_Print, v, Ast_Node_Binary, n)
174     binary_op_action(v, n, "expr_gt");
175 ASTVF_END
176
177 ASTVF_BEGIN(AST_EXPR_LT, Ast_Visitor_Print, v, Ast_Node_Binary, n)
178     binary_op_action(v, n, "expr_lt");
179 ASTVF_END
180
181 ASTVF_BEGIN(AST_EXPR_GTEQ, Ast_Visitor_Print, v, Ast_Node_Binary, n)
182     binary_op_action(v, n, "expr_gteq");
183 ASTVF_END
184
185 ASTVF_BEGIN(AST_EXPR_LTEQ, Ast_Visitor_Print, v, Ast_Node_Binary, n)
186     binary_op_action(v, n, "expr_lteq");
187 ASTVF_END
188
189 ASTVF_BEGIN(AST_EXPR_PLUS, Ast_Visitor_Print, v, Ast_Node_Binary, n)
190     binary_op_action(v, n, "expr_plus");
191 ASTVF_END
192
193 ASTVF_BEGIN(AST_EXPR_MINUS, Ast_Visitor_Print, v, Ast_Node_Binary, n)
194     binary_op_action(v, n, "expr_minus");
195 ASTVF_END
196
197 ASTVF_BEGIN(AST_EXPR_MUL, Ast_Visitor_Print, v, Ast_Node_Binary, n)
198     binary_op_action(v, n, "expr_mul");
199 ASTVF_END
200
201 ASTVF_BEGIN(AST_EXPR_DIV, Ast_Visitor_Print, v, Ast_Node_Binary, n)
202     binary_op_action(v, n, "expr_div");
203 ASTVF_END
204
205 ASTVF_BEGIN(AST_EXPR_CAST, Ast_Visitor_Print, v, Ast_Node_Binary, n)
206     binary_op_action(v, n, "expr_type_cast");
207 ASTVF_END
208
209 ASTVF_BEGIN(AST_EXPR_LNOT, Ast_Visitor_Print, v, Ast_Node_Unary, n)
210     unary_op_action(v, n, "expr_logic_not");
211 ASTVF_END
212
213 ASTVF_BEGIN(AST_EXPR_ABS, Ast_Visitor_Print, v, Ast_Node_Unary, n)
214     unary_op_action(v, n, "expr_abs");
215 ASTVF_END
216
217 ASTVF_BEGIN(AST_EXPR_INT, Ast_Visitor_Print, v, Ast_Expr_Int, n)
218     PRINT_SINGLE(v->output_file, "int_const value=\"%U\" "
219         "line=\"%U\" column=\"%U\"",
220         v->indentation,
221         n->val,
222         n->ast_node.location.line,
223         n->ast_node.location.column);
224 ASTVF_END
225
226 ASTVF_BEGIN(AST_EXPR_BOOL, Ast_Visitor_Print, v, Ast_Expr_Bool, n)
227     PRINT_SINGLE(v->output_file, "bool_const value=\"%s\" "
228         "line=\"%U\" column=\"%U\"",
229         v->indentation,
230         n->val ? "true" : "false",
231         n->ast_node.location.line,
232         n->ast_node.location.column);
233 ASTVF_END
234

```

```

235 ASTVF_BEGIN(AST_EXPR_NULL, Ast_Visitor_Print, v, Ast_Expr_Null, n)
236     PRINT_SINGLE(v->output_file, "null_const line=\"%U\" column=\"%U\"",
237         v->indentation,
238         n->ast_node.location.line,
239         n->ast_node.location.column);
240 ASTVF_END
241
242 ASTVF_BEGIN(AST_VARIABLE_IDEN, Ast_Visitor_Print, v, Ast_Variable_Iden, n)
243     PRINT_SINGLE(v->output_file, "identifier id=\"%S\" type=\"%S\" "
244         "line=\"%U\" column=\"%U\"",
245         v->indentation,
246         n->iden,
247         ast_variable_iden_get_expr_str(n),
248         n->ast_node.location.line,
249         n->ast_node.location.column);
250 ASTVF_END
251
252 ASTVF_BEGIN(AST_REC_SELF_PTR, Ast_Visitor_Print, v, Ast_Empty, n)
253     PRINT_SINGLE(v->output_file, "record_self "
254         "line=\"%U\" column=\"%U\"",
255         v->indentation,
256         n->ast_node.location.line,
257         n->ast_node.location.column);
258 ASTVF_END
259
260 ASTVF_BEGIN(AST_EXPR_ARY_REF, Ast_Visitor_Print, v, Ast_Node_Binary, n)
261     binary_op_action(v, n, "array_ref");
262 ASTVF_END
263
264 ASTVF_BEGIN(AST_EXPR_FUNC_CALL, Ast_Visitor_Print, v, Ast_Expr_Func_Call, n)
265     PRINT_OPEN(v->output_file, "func_call line=\"%U\" column=\"%U\"",
266         v->indentation,
267         n->ast_node.location.line,
268         n->ast_node.location.column);
269
270     v->indentation += INDENT_COUNT;
271
272     n->identifier->accept_visitor(n->identifier, AST_VISITOR_OF(v));
273
274     Ast_Node *arg;
275     Vector *vargs = n->arguments;
276     VECTOR_FOR_EACH_ENTRY(vargs, arg)
277         arg->accept_visitor(arg, AST_VISITOR_OF(v));
278
279     v->indentation -= INDENT_COUNT;
280
281     PRINT_CLOSE(v->output_file, "func_call", v->indentation);
282 ASTVF_END
283
284 ASTVF_BEGIN(AST_EXPR_DOT_REF, Ast_Visitor_Print, v, Ast_Node_Binary, n)
285     binary_op_action(v, n, "record_ref");
286 ASTVF_END
287
288 ASTVF_BEGIN(AST_EXPR_DIRECT_REF, Ast_Visitor_Print, v, Ast_Node_Binary, n)
289     binary_op_action(v, n, "direct_ref");
290 ASTVF_END
291
292 ASTVF_BEGIN(AST_SIMPLE_TYPE_INT, Ast_Visitor_Print, v, Ast_Empty, n)
293     simple_type_action(v, n, "int");
294 ASTVF_END
295
296 ASTVF_BEGIN(AST_SIMPLE_TYPE_VOID, Ast_Visitor_Print, v, Ast_Empty, n)
297     simple_type_action(v, n, "void");
298 ASTVF_END
299
300 ASTVF_BEGIN(AST_SIMPLE_TYPE_BOOL, Ast_Visitor_Print, v, Ast_Empty, n)
301     simple_type_action(v, n, "bool");
302 ASTVF_END
303
304 ASTVF_BEGIN(AST_TYPE_IDEN, Ast_Visitor_Print, v, Ast_Type_Iden, n)
305     PRINT_SINGLE(v->output_file, "type_iden id=\"%S\" "
306         "line=\"%U\" column=\"%U\"",
307         v->indentation,
308         n->iden,

```



```

309         n->ast_node.location.line,
310         n->ast_node.location.column);
311 ASTVF_END
312
313 ASTVF_BEGIN(AST_VAR_DECL, Ast_Visitor_Print, v, Ast_Node_Binary, n)
314     binary_action(v, n, "var_decl");
315 ASTVF_END
316
317 ASTVF_BEGIN(AST_TYPE_DEF, Ast_Visitor_Print, v, Ast_Node_Binary, n)
318     binary_action(v, n, "type_def");
319 ASTVF_END
320
321 ASTVF_BEGIN(AST_TYPE_ARY, Ast_Visitor_Print, v, Ast_Type, n)
322     type_action(v, n, "array_type");
323 ASTVF_END
324
325 ASTVF_BEGIN(AST_TYPE_REC, Ast_Visitor_Print, v, Ast_Type_Rec, n)
326     PRINT_OPEN(v->output_file, "record_type line=\"%U\" column=\"%U\"",
327         v->indentation,
328         n->ast_node.location.line,
329         n->ast_node.location.column);
330
331     v->indentation += INDENT_COUNT;
332
333     Ast_Node *arg;
334     Vector *vargs = n->extend_list;
335     VECTOR_FOR_EACH_ENTRY(vargs, arg)
336         arg->accept_visitor(arg, AST_VISITOR_OF(v));
337
338     vargs = n->body;
339     VECTOR_FOR_EACH_ENTRY(vargs, arg)
340         arg->accept_visitor(arg, AST_VISITOR_OF(v));
341
342     v->indentation -= INDENT_COUNT;
343
344     PRINT_CLOSE(v->output_file, "record_type", v->indentation);
345 ASTVF_END
346
347 ASTVF_BEGIN(AST_STMT_LIST, Ast_Visitor_Print, v, Ast_Stmt_List, n)
348     stmt_list_action(v, n, "statements");
349 ASTVF_END
350
351 ASTVF_BEGIN(AST_FIN_STMT_LIST, Ast_Visitor_Print, v, Ast_Stmt_List, n)
352     stmt_list_action(v, n, "finalize_stmt");
353 ASTVF_END
354
355 static void func_print(Ast_Visitor_Print *v, Ast_Func_Def *n)
356 {
357     Ast_Node *p;
358     Vector *vec = n->parameters;
359
360     PRINT_OPEN(v->output_file, "function line=\"%U\" column=\"%U\"",
361         v->indentation,
362         n->ast_node.location.line,
363         n->ast_node.location.column);
364
365     v->indentation += INDENT_COUNT;
366
367     n->iden->accept_visitor(n->iden, AST_VISITOR_OF(v));
368     n->return_type->accept_visitor(n->return_type, AST_VISITOR_OF(v));
369     VECTOR_FOR_EACH_ENTRY(vec, p)
370         p->accept_visitor(p, AST_VISITOR_OF(v));
371     n->statements->accept_visitor(n->statements, AST_VISITOR_OF(v));
372
373     v->indentation -= INDENT_COUNT;
374
375     PRINT_CLOSE(v->output_file, "function", v->indentation);
376 }
377
378 ASTVF_BEGIN(AST_EXT_FUNC_DECL, Ast_Visitor_Print, v, Ast_Func_Def, n)
379     Ast_Node *p;
380     Vector *vec = n->parameters;
381
382     PRINT_OPEN(v->output_file, "extern_func line=\"%U\" column=\"%U\"",

```

```

383         v->indentation,
384         n->ast_node.location.line,
385         n->ast_node.location.column);
386
387     v->indentation += INDENT_COUNT;
388
389     n->extern_type->accept_visitor(n->extern_type, AST_VISITOR_OF(v));
390     n->iden->accept_visitor(n->iden, AST_VISITOR_OF(v));
391     n->return_type->accept_visitor(n->return_type, AST_VISITOR_OF(v));
392
393     VECTOR_FOR_EACH_ENTRY(vec, p)
394         p->accept_visitor(p, AST_VISITOR_OF(v));
395
396     v->indentation -= INDENT_COUNT;
397
398     PRINT_CLOSE(v->output_file, "extern_func", v->indentation);
399 ASTVF_END
400
401 ASTVF_BEGIN(AST_FIN_FUNC_DEF, Ast_Visitor_Print, v, Ast_Func_Def, n)
402     func_print(v, n);
403 ASTVF_END
404
405 ASTVF_BEGIN(AST_REC_FUNC_DEF, Ast_Visitor_Print, v, Ast_Func_Def, n)
406     func_print(v, n);
407 ASTVF_END
408
409 ASTVF_BEGIN(AST_FUNC_DEF, Ast_Visitor_Print, v, Ast_Func_Def, n)
410     func_print(v, n);
411 ASTVF_END
412
413 ASTVF_BEGIN(AST_IF_STMT, Ast_Visitor_Print, v, Ast_Node_Binary, n)
414     binary_action(v, n, "if_stmt");
415 ASTVF_END
416
417 ASTVF_BEGIN(AST_IF_ELSE_STMT, Ast_Visitor_Print, v, Ast_Node_Ternary, n)
418     ternary_action(v, n, "if_else_stmt");
419 ASTVF_END
420
421 ASTVF_BEGIN(AST_ALLOC_ARY, Ast_Visitor_Print, v, Ast_Node_Binary, n)
422     binary_action(v, n, "allocate_array");
423 ASTVF_END
424
425 ASTVF_BEGIN(AST_ALLOC_REC, Ast_Visitor_Print, v, Ast_Node_Unary, n)
426     unary_action(v, n, "allocate_rec");
427 ASTVF_END
428
429 ASTVF_BEGIN(AST_ALLOC_REC_CALL, Ast_Visitor_Print, v, Ast_Node_Binary, n)
430     binary_action(v, n, "allocate_rec");
431 ASTVF_END
432
433 ASTVF_BEGIN(AST_DELETE, Ast_Visitor_Print, v, Ast_Node_Unary, n)
434     unary_action(v, n, "delete");
435 ASTVF_END
436
437 ASTVF_BEGIN(AST_WHILE_STMT, Ast_Visitor_Print, v, Ast_Node_Binary, n)
438     binary_action(v, n, "while_stmt");
439 ASTVF_END
440
441 ASTVF_BEGIN(AST_RETURN_STMT, Ast_Visitor_Print, v, Ast_Node_Unary, n)
442     unary_action(v, n, "return_stmt");
443 ASTVF_END
444
445 ASTVF_BEGIN(AST_WRITE_STMT, Ast_Visitor_Print, v, Ast_Node_Unary, n)
446     unary_action(v, n, "write_stmt");
447 ASTVF_END
448
449 ASTVF_BEGIN(AST_ASSIGNMENT, Ast_Visitor_Print, v, Ast_Node_Binary, n)
450     binary_action(v, n, "assignment");
451 ASTVF_END
452
453 ASTVF_BEGIN(AST_FUNC_BODY, Ast_Visitor_Print, v, Ast_Stmt_List, n)
454     stmt_list_action(v, n, "func_body");
455 ASTVF_END
456

```

```

457 ASTVF_BEGIN(AST_EXPR_CHAR, Ast_Visitor_Print, v, Ast_Expr_Char, n)
458     PRINT_SINGLE(v->output_file, "char_const value=\"%s" PRIx32 "\" "
459                 "line=\"%U\" column=\"%U\"",
460                 v->indentation,
461                 n->val,
462                 n->ast_node.location.line,
463                 n->ast_node.location.column);
464 ASTVF_END
465
466 ASTVF_BEGIN(AST_EXPR_STRING, Ast_Visitor_Print, v, Ast_Expr_String, n)
467     String_Builder sb = STRING_BUILDER_INIT();
468     ast_string_escape(n->val, &sb);
469     PRINT_SINGLE(v->output_file, "string_const value=\"%S\" "
470                 "line=\"%U\" column=\"%U\"",
471                 v->indentation,
472                 string_builder_const_str(&sb),
473                 n->ast_node.location.line,
474                 n->ast_node.location.column);
475     string_builder_clear(&sb);
476 ASTVF_END
477
478 ASTVF_BEGIN(AST_IMPORT_STRING, Ast_Visitor_Print, v, Ast_Module_String, n)
479     PRINT_SINGLE(v->output_file, "import file=\"%S\" "
480                 "line=\"%U\" column=\"%U\"",
481                 v->indentation,
482                 n->module,
483                 n->ast_node.location.line,
484                 n->ast_node.location.column);
485 ASTVF_END
486
487 ASTVF_BEGIN(AST_PACKAGE_STRING, Ast_Visitor_Print, v, Ast_Module_String, n)
488     PRINT_SINGLE(v->output_file, "package directory=\"%S\" "
489                 "line=\"%U\" column=\"%U\"",
490                 v->indentation,
491                 n->module,
492                 n->ast_node.location.line,
493                 n->ast_node.location.column);
494 ASTVF_END
495
496 ASTVF_BEGIN(AST_SIMPLE_TYPE_CHAR, Ast_Visitor_Print, v, Ast_Empty, n)
497     simple_type_action(v, n, "char");
498 ASTVF_END
499
500 ASTVF_BEGIN(AST_SIMPLE_TYPE_STRING, Ast_Visitor_Print, v, Ast_Empty, n)
501     simple_type_action(v, n, "string");
502 ASTVF_END
503
504 static Ast_Visitor_Print print_visitor = {
505     .AST_VISITOR_FIELD = AST_VISITOR_STATIC_INIT(),
506     .indentation = 0,
507     .output_file = NULL
508 };
509
510 void ast_visitor_print_accept_visitor(Ast_Node *root,
511     Const_String file_name_prefix)
512 {
513     String file_name = string_from_format(S("%S.vitaly.parse-tree"),
514         file_name_prefix);
515     print_visitor.output_file = file_open(file_name, S("w"));
516
517     if (!print_visitor.output_file)
518         fatal_error(S("Unable to open file '%S' for parse tree dump"),
519             file_name);
520     string_destroy(file_name);
521
522     root->accept_visitor(root, AST_VISITOR_OF(&print_visitor));
523
524     file_close(print_visitor.output_file);
525 }

```

:

A.3.7 src/ast/ast_visitor_print.h

```

1  #ifndef AST_VISITOR_PRINT_H
2  #define AST_VISITOR_PRINT_H
3
4  #include "ast_visitor.h"
5
6  void ast_visitor_print_accept_visitor(Ast_Node *root,
7      Const_String file_name_prefix);
8
9  #endif // AST_VISITOR_PRINT_H

```

:

A.3.8 src/ast/ast_visitor_print_graph.c

```

1  #include "ast_visitor_print_graph.h"
2  #include "ast_string.h"
3  #include <std_defines.h>
4  #include <std_include.h>
5  #include <dot_printer.h>
6
7  #define ASTVF_G_END \
8      dot_printer_pop_current_id(v->printer); \
9      ASTVF_END
10
11  AST_VISITOR_STRUCT_BEGIN(Ast_Visitor_Print_Graph)
12      Dot_Printer *printer;
13  AST_VISITOR_STRUCT_END(Ast_Visitor_Print_Graph)
14
15  static void stmt_list_action(Ast_Visitor_Print_Graph *v, Ast_Stmt_List *n,
16      const char *type)
17  {
18      Ast_Node *stmt;
19      Vector *vec = n->statements;
20
21      String tmp_str = string_from_format(S("%s\\nlocation: %U:%U"),
22          type,
23          n->ast_node.location.line,
24          n->ast_node.location.column);
25      dot_printer_push_insert(v->printer, n, tmp_str);
26      string_destroy(tmp_str);
27
28      VECTOR_FOR_EACH_ENTRY(vec, stmt){
29          stmt->accept_visitor(stmt, AST_VISITOR_OF(v));
30      }
31  }
32
33  static void ternary_action(Ast_Visitor_Print_Graph *v, Ast_Node_Ternary *n,
34      const char *name)
35  {
36      String tmp_str = string_from_format(S("%s\\nlocation: %U:%U"),
37          name,
38          n->ast_node.location.line,
39          n->ast_node.location.column);
40      dot_printer_push_insert(v->printer, n, tmp_str);
41      string_destroy(tmp_str);
42
43      n->lhs->accept_visitor(n->lhs, AST_VISITOR_OF(v));
44      n->mid->accept_visitor(n->mid, AST_VISITOR_OF(v));
45      n->rhs->accept_visitor(n->rhs, AST_VISITOR_OF(v));
46  }
47
48  static void binary_action(Ast_Visitor_Print_Graph *v, Ast_Node_Binary *n,
49      const char *name)
50  {
51      String tmp_str = string_from_format(S("%s\\nlocation: %U:%U"),
52          name,
53          n->ast_node.location.line,
54          n->ast_node.location.column);

```

```

55     dot_printer_push_insert(v->printer, n,tmp_str);
56     string_destroy(tmp_str);
57
58     if (n->lhs) // because of DIRECT_REF
59         n->lhs->accept_visitor(n->lhs, AST_VISITOR_OF(v));
60     n->rhs->accept_visitor(n->rhs, AST_VISITOR_OF(v));
61 }
62
63 static void binary_op_action(Ast_Visitor_Print_Graph *v, Ast_Node_Binary *n,
64                             const char *name)
65 {
66     Ast_Expr_Binary *bin = AST_EXPR_BINARY_OF(n);
67     String tmp_str = string_from_format(
68         S("%s\\ntype=\\\\"%S\\\\"\\nlocation: %U:%U"),
69         name,
70         ast_expr_binary_get_expr_str(bin),
71         n->ast_node.location.line,
72         n->ast_node.location.column);
73     dot_printer_push_insert(v->printer, n,tmp_str);
74     string_destroy(tmp_str);
75
76     n->lhs->accept_visitor(n->lhs, AST_VISITOR_OF(v));
77     n->rhs->accept_visitor(n->rhs, AST_VISITOR_OF(v));
78 }
79
80 static void unary_action(Ast_Visitor_Print_Graph *v, Ast_Node_Unary *n,
81                          const char *name)
82 {
83     String tmp_str = string_from_format(S("%s\\nlocation: %U:%U"),
84         name,
85         n->ast_node.location.line,
86         n->ast_node.location.column);
87     dot_printer_push_insert(v->printer, n,tmp_str);
88     string_destroy(tmp_str);
89
90     n->expr->accept_visitor(n->expr, AST_VISITOR_OF(v));
91 }
92
93 static void unary_op_action(Ast_Visitor_Print_Graph *v, Ast_Node_Unary *n,
94                             const char *name)
95 {
96     Ast_Expr_Unary *una = AST_EXPR_UNARY_OF(n);
97     String tmp_str = string_from_format(
98         S("%s\\ntype=\\\\"%S\\\\"\\nlocation: %U:%U"),
99         name,
100         ast_expr_unary_get_expr_str(una),
101         n->ast_node.location.line,
102         n->ast_node.location.column);
103     dot_printer_push_insert(v->printer, n,tmp_str);
104     string_destroy(tmp_str);
105
106     n->expr->accept_visitor(n->expr, AST_VISITOR_OF(v));
107 }
108
109 static void simple_type_action(Ast_Visitor_Print_Graph *v, Ast_Empty *n,
110                               const char *type)
111 {
112     String tmp_str = string_from_format(
113         S("simple_type\\ntype=\\\\"%s\\\\"\\nlocation: %U:%U"),
114         type,
115         n->ast_node.location.line,
116         n->ast_node.location.column);
117     dot_printer_push_insert(v->printer, n,tmp_str);
118     string_destroy(tmp_str);
119 }
120
121 static void type_action(Ast_Visitor_Print_Graph *v, Ast_Type *n,
122                        const char *type)
123 {
124     String tmp_str = string_from_format(S("%s location: %U:%U"),
125         type,
126         n->ast_node.location.line,
127         n->ast_node.location.column);
128     dot_printer_push_insert(v->printer, n,tmp_str);

```

```

129     string_destroy(tmp_str);
130
131     n->type->accept_visitor(n->type, AST_VISITOR_OF(v));
132 }
133
134 ASTVF_BEGIN(AST_EXPR_LOR, Ast_Visitor_Print_Graph, v, Ast_Node_Binary, n)
135     binary_op_action(v, n, "expr_logic_or");
136 ASTVF_G_END
137
138 ASTVF_BEGIN(AST_EXPR_LAND, Ast_Visitor_Print_Graph, v, Ast_Node_Binary, n)
139     binary_op_action(v, n, "expr_logic_and");
140 ASTVF_G_END
141
142 ASTVF_BEGIN(AST_EXPR_EQ, Ast_Visitor_Print_Graph, v, Ast_Node_Binary, n)
143     binary_op_action(v, n, "expr_eq");
144 ASTVF_G_END
145
146 ASTVF_BEGIN(AST_EXPR_NEQ, Ast_Visitor_Print_Graph, v, Ast_Node_Binary, n)
147     binary_op_action(v, n, "expr_neq");
148 ASTVF_G_END
149
150 ASTVF_BEGIN(AST_EXPR_GT, Ast_Visitor_Print_Graph, v, Ast_Node_Binary, n)
151     binary_op_action(v, n, "expr_gt");
152 ASTVF_G_END
153
154 ASTVF_BEGIN(AST_EXPR_LT, Ast_Visitor_Print_Graph, v, Ast_Node_Binary, n)
155     binary_op_action(v, n, "expr_lt");
156 ASTVF_G_END
157
158 ASTVF_BEGIN(AST_EXPR_GTEQ, Ast_Visitor_Print_Graph, v, Ast_Node_Binary, n)
159     binary_op_action(v, n, "expr_gteq");
160 ASTVF_G_END
161
162 ASTVF_BEGIN(AST_EXPR_LTEQ, Ast_Visitor_Print_Graph, v, Ast_Node_Binary, n)
163     binary_op_action(v, n, "expr_lteq");
164 ASTVF_G_END
165
166 ASTVF_BEGIN(AST_EXPR_PLUS, Ast_Visitor_Print_Graph, v, Ast_Node_Binary, n)
167     binary_op_action(v, n, "expr_plus");
168 ASTVF_G_END
169
170 ASTVF_BEGIN(AST_EXPR_MINUS, Ast_Visitor_Print_Graph, v, Ast_Node_Binary, n)
171     binary_op_action(v, n, "expr_minus");
172 ASTVF_G_END
173
174 ASTVF_BEGIN(AST_EXPR_MUL, Ast_Visitor_Print_Graph, v, Ast_Node_Binary, n)
175     binary_op_action(v, n, "expr_mul");
176 ASTVF_G_END
177
178 ASTVF_BEGIN(AST_EXPR_DIV, Ast_Visitor_Print_Graph, v, Ast_Node_Binary, n)
179     binary_op_action(v, n, "expr_div");
180 ASTVF_G_END
181
182 ASTVF_BEGIN(AST_EXPR_CAST, Ast_Visitor_Print_Graph, v, Ast_Node_Binary, n)
183     binary_op_action(v, n, "expr_type_cast");
184 ASTVF_G_END
185
186 ASTVF_BEGIN(AST_EXPR_LNOT, Ast_Visitor_Print_Graph, v, Ast_Node_Unary, n)
187     unary_op_action(v, n, "expr_logic_not");
188 ASTVF_G_END
189
190 ASTVF_BEGIN(AST_EXPR_ABS, Ast_Visitor_Print_Graph, v, Ast_Node_Unary, n)
191     unary_op_action(v, n, "expr_abs");
192 ASTVF_G_END
193
194 ASTVF_BEGIN(AST_EXPR_INT, Ast_Visitor_Print_Graph, v, Ast_Expr_Int, n)
195     String tmp_str = string_from_format(S("int_const\\nvalue=\\\\"%U\\\\""),
196     "\\nlocation: %U:%U"),
197     n->val,
198     n->ast_node.location.line,
199     n->ast_node.location.column);
200     dot_printer_push_insert(v->printer, n, tmp_str);
201     string_destroy(tmp_str);
202 ASTVF_G_END

```

```

203
204 ASTVF_BEGIN(AST_EXPR_BOOL, Ast_Visitor_Print_Graph, v, Ast_Expr_Bool, n)
205     String tmp_str = string_from_format(S("bool_const\\nvalue=\\\\"%s\\\\""),
206         "\\nlocation: %U:%U"),
207     n->val ? "true" : "false",
208     n->ast_node.location.line,
209     n->ast_node.location.column);
210     dot_printer_push_insert(v->printer, n, tmp_str);
211     string_destroy(tmp_str);
212 ASTVF_G_END
213
214 ASTVF_BEGIN(AST_EXPR_NULL, Ast_Visitor_Print_Graph, v, Ast_Expr_Null, n)
215     String tmp_str = string_from_format(S("null_const"
216         "\\nlocation: %U:%U"),
217     n->ast_node.location.line,
218     n->ast_node.location.column);
219     dot_printer_push_insert(v->printer, n, tmp_str);
220     string_destroy(tmp_str);
221 ASTVF_G_END
222
223 ASTVF_BEGIN(AST_VARIABLE_IDEN, Ast_Visitor_Print_Graph, v,
224     Ast_Variable_Iden, n)
225     String tmp_str = string_from_format(S("identifier\\nid=\\\\"%S\\\\"\\n"
226         "type=\\\\"%S\\\\"\\nlocation: %U:%U"),
227     n->iden,
228     ast_variable_iden_get_expr_str(n),
229     n->ast_node.location.line,
230     n->ast_node.location.column);
231
232     dot_printer_push_insert(v->printer, n, tmp_str);
233     string_destroy(tmp_str);
234 ASTVF_G_END
235
236 ASTVF_BEGIN(AST_REC_SELF_PTR, Ast_Visitor_Print_Graph, v, Ast_Empty, n)
237     String tmp_str = string_from_format(S("record_self\\nlocation: %U:%U"),
238     n->ast_node.location.line,
239     n->ast_node.location.column);
240
241     dot_printer_push_insert(v->printer, n, tmp_str);
242     string_destroy(tmp_str);
243 ASTVF_END
244
245 ASTVF_BEGIN(AST_EXPR_ARY_REF, Ast_Visitor_Print_Graph, v,
246     Ast_Node_Binary, n)
247     binary_action(v, n, "array_ref");
248 ASTVF_G_END
249
250 ASTVF_BEGIN(AST_EXPR_FUNC_CALL, Ast_Visitor_Print_Graph, v,
251     Ast_Expr_Func_Call, n)
252     String tmp_str = string_from_format(S("func_call\\nlocation: %U:%U"),
253     n->ast_node.location.line,
254     n->ast_node.location.column);
255     dot_printer_push_insert(v->printer, n, tmp_str);
256     string_destroy(tmp_str);
257
258     n->identifier->accept_visitor(n->identifier, AST_VISITOR_OF(v));
259
260     Ast_Node *arg;
261
262     Vector *vargs = n->arguments;
263     VECTOR_FOR_EACH_ENTRY(vargs, arg)
264         arg->accept_visitor(arg, AST_VISITOR_OF(v));
265 ASTVF_G_END
266
267 ASTVF_BEGIN(AST_EXPR_DOT_REF, Ast_Visitor_Print_Graph, v,
268     Ast_Node_Binary, n)
269     binary_action(v, n, "record_ref");
270 ASTVF_G_END
271
272 ASTVF_BEGIN(AST_EXPR_DIRECT_REF, Ast_Visitor_Print_Graph, v,
273     Ast_Node_Binary, n)
274     binary_action(v, n, "direct_ref");
275 ASTVF_G_END
276

```

```

277 ASTVF_BEGIN(AST_SIMPLE_TYPE_INT, Ast_Visitor_Print_Graph, v,
278     Ast_Empty, n)
279     simple_type_action(v, n, "int");
280 ASTVF_G_END
281
282 ASTVF_BEGIN(AST_SIMPLE_TYPE_VOID, Ast_Visitor_Print_Graph, v,
283     Ast_Empty, n)
284     simple_type_action(v, n, "void");
285 ASTVF_G_END
286
287 ASTVF_BEGIN(AST_SIMPLE_TYPE_BOOL, Ast_Visitor_Print_Graph, v,
288     Ast_Empty, n)
289     simple_type_action(v, n, "bool");
290 ASTVF_G_END
291
292 ASTVF_BEGIN(AST_TYPE_IDEN, Ast_Visitor_Print_Graph, v, Ast_Type_Iden, n)
293     String tmp_str = string_from_format(
294         S("type_iden\\nid=\\\\"%S\\\\"\\nlocation: %U:%U"),
295         n->iden,
296         n->ast_node.location.line,
297         n->ast_node.location.column);
298     dot_printer_push_insert(v->printer, n,tmp_str);
299     string_destroy(tmp_str);
300 ASTVF_G_END
301
302 ASTVF_BEGIN(AST_VAR_DECL, Ast_Visitor_Print_Graph, v, Ast_Node_Binary, n)
303     binary_action(v, n, "var_decl");
304 ASTVF_G_END
305
306 ASTVF_BEGIN(AST_TYPE_DEF, Ast_Visitor_Print_Graph, v, Ast_Node_Binary, n)
307     binary_action(v, n, "type_def");
308 ASTVF_G_END
309
310 ASTVF_BEGIN(AST_TYPE_ARY, Ast_Visitor_Print_Graph, v, Ast_Type, n)
311     type_action(v, n, "array_type");
312 ASTVF_G_END
313
314 ASTVF_BEGIN(AST_TYPE_REC, Ast_Visitor_Print_Graph, v, Ast_Type_Rec, n)
315     String tmp_str = string_from_format(S("record_type\\nlocation: %U:%U"),
316         n->ast_node.location.line,
317         n->ast_node.location.column);
318     dot_printer_push_insert(v->printer, n,tmp_str);
319     string_destroy(tmp_str);
320
321     Ast_Node *arg;
322     Vector *vargs = n->extend_list;
323     VECTOR_FOR_EACH_ENTRY(vargs, arg)
324         arg->accept_visitor(arg, AST_VISITOR_OF(v));
325
326     vargs = n->body;
327     VECTOR_FOR_EACH_ENTRY(vargs, arg)
328         arg->accept_visitor(arg, AST_VISITOR_OF(v));
329
330 ASTVF_G_END
331
332 ASTVF_BEGIN(AST_STMT_LIST, Ast_Visitor_Print_Graph, v, Ast_Stmt_List, n)
333     stmt_list_action(v, n, "statements");
334 ASTVF_G_END
335
336 ASTVF_BEGIN(AST_FIN_STMT_LIST, Ast_Visitor_Print_Graph, v, Ast_Stmt_List, n)
337     stmt_list_action(v, n, "finalize_stmt");
338 ASTVF_G_END
339
340 static void func_print(Ast_Visitor_Print_Graph *v, Ast_Func_Def *n)
341 {
342     Ast_Node *p;
343     Vector *vec = n->parameters;
344
345     String tmp_str = string_from_format(S("function\\nlocation: %U:%U"),
346         n->ast_node.location.line,
347         n->ast_node.location.column);
348     dot_printer_push_insert(v->printer, n,tmp_str);
349     string_destroy(tmp_str);
350

```



```

351     n->iden->accept_visitor(n->iden, AST_VISITOR_OF(v));
352     n->return_type->accept_visitor(n->return_type, AST_VISITOR_OF(v));
353     VECTOR_FOR_EACH_ENTRY(vec, p)
354         p->accept_visitor(p, AST_VISITOR_OF(v));
355
356     n->statements->accept_visitor(n->statements, AST_VISITOR_OF(v));
357 }
358
359 ASTVF_BEGIN(AST_EXT_FUNC_DECL, Ast_Visitor_Print_Graph, v, Ast_Func_Def, n)
360     Ast_Node *p;
361     Vector *vec = n->parameters;
362
363     String tmp_str = string_from_format(S("extern_func\\nlocation: %U:%U"),
364         n->ast_node.location.line,
365         n->ast_node.location.column);
366     dot_printer_push_insert(v->printer, n, tmp_str);
367     string_destroy(tmp_str);
368
369     n->extern_type->accept_visitor(n->extern_type, AST_VISITOR_OF(v));
370     n->iden->accept_visitor(n->iden, AST_VISITOR_OF(v));
371     n->return_type->accept_visitor(n->return_type, AST_VISITOR_OF(v));
372
373     VECTOR_FOR_EACH_ENTRY(vec, p)
374         p->accept_visitor(p, AST_VISITOR_OF(v));
375 ASTVF_G_END
376
377 ASTVF_BEGIN(AST_FIN_FUNC_DEF, Ast_Visitor_Print_Graph, v, Ast_Func_Def, n)
378     func_print(v, n);
379 ASTVF_G_END
380
381 ASTVF_BEGIN(AST_REC_FUNC_DEF, Ast_Visitor_Print_Graph, v, Ast_Func_Def, n)
382     func_print(v, n);
383 ASTVF_G_END
384
385 ASTVF_BEGIN(AST_FUNC_DEF, Ast_Visitor_Print_Graph, v, Ast_Func_Def, n)
386     func_print(v, n);
387 ASTVF_G_END
388
389 ASTVF_BEGIN(AST_IF_STMT, Ast_Visitor_Print_Graph, v, Ast_Node_Binary, n)
390     binary_action(v, n, "if_stmt");
391 ASTVF_G_END
392
393 ASTVF_BEGIN(AST_IF_ELSE_STMT, Ast_Visitor_Print_Graph, v,
394     Ast_Node_Ternary, n)
395     ternary_action(v, n, "if_else_stmt");
396 ASTVF_G_END
397
398 ASTVF_BEGIN(AST_ALLOC_ARY, Ast_Visitor_Print_Graph, v, Ast_Node_Binary, n)
399     binary_action(v, n, "allocate_array");
400 ASTVF_G_END
401
402 ASTVF_BEGIN(AST_ALLOC_REC, Ast_Visitor_Print_Graph, v, Ast_Node_Unary, n)
403     unary_action(v, n, "allocate_rec");
404 ASTVF_G_END
405
406 ASTVF_BEGIN(AST_ALLOC_REC_CALL, Ast_Visitor_Print_Graph, v,
407     Ast_Node_Binary, n)
408     binary_action(v, n, "allocate_rec");
409 ASTVF_G_END
410
411 ASTVF_BEGIN(AST_DELETE, Ast_Visitor_Print_Graph, v, Ast_Node_Unary, n)
412     unary_action(v, n, "delete");
413 ASTVF_G_END
414
415 ASTVF_BEGIN(AST_WHILE_STMT, Ast_Visitor_Print_Graph, v, Ast_Node_Binary, n)
416     binary_action(v, n, "while_stmt");
417 ASTVF_G_END
418
419 ASTVF_BEGIN(AST_RETURN_STMT, Ast_Visitor_Print_Graph, v, Ast_Node_Unary, n)
420     unary_action(v, n, "return_stmt");
421 ASTVF_G_END
422
423 ASTVF_BEGIN(AST_WRITE_STMT, Ast_Visitor_Print_Graph, v, Ast_Node_Unary, n)
424     unary_action(v, n, "write_stmt");

```

```

425 ASTVF_G_END
426
427 ASTVF_BEGIN(AST_ASSIGNMENT, Ast_Visitor_Print_Graph, v, Ast_Node_Binary, n)
428     binary_action(v, n, "assignment");
429 ASTVF_G_END
430
431 ASTVF_BEGIN(AST_FUNC_BODY, Ast_Visitor_Print_Graph, v, Ast_Stmt_List, n)
432     stmt_list_action(v, n, "func_body");
433 ASTVF_G_END
434
435 ASTVF_BEGIN(AST_EXPR_CHAR, Ast_Visitor_Print_Graph, v, Ast_Expr_Char, n)
436     String tmp_str = string_from_format(
437         S("char_const\\nvalue=\\\\" PRIx32 "\\\" "
438         "\\nlocation: %U:%U"),
439         n->val,
440         n->ast_node.location.line,
441         n->ast_node.location.column);
442     dot_printer_push_insert(v->printer, n,tmp_str);
443     string_destroy(tmp_str);
444 ASTVF_G_END
445
446 ASTVF_BEGIN(AST_EXPR_STRING, Ast_Visitor_Print_Graph, v, Ast_Expr_String, n)
447     String_Builder sb = STRING_BUILDER_INIT();
448     ast_string_escape(n->val, &sb);
449     String tmp_str = string_from_format(S("string_const\\nvalue=\\\\" "%S\\\" "
450     "\\nlocation: %U:%U"),
451     string_builder_const_str(&sb),
452     n->ast_node.location.line,
453     n->ast_node.location.column);
454     string_builder_clear(&sb);
455     dot_printer_push_insert(v->printer, n,tmp_str);
456     string_destroy(tmp_str);
457 ASTVF_G_END
458
459 ASTVF_BEGIN(AST_IMPORT_STRING, Ast_Visitor_Print_Graph, v,
460     Ast_Module_String, n)
461     String tmp_str = string_from_format(S("import\\nfile=\\\\" "%S\\\" "
462     "\\nlocation: %U:%U"),
463     n->module,
464     n->ast_node.location.line,
465     n->ast_node.location.column);
466     dot_printer_push_insert(v->printer, n,tmp_str);
467     string_destroy(tmp_str);
468 ASTVF_G_END
469
470 ASTVF_BEGIN(AST_PACKAGE_STRING, Ast_Visitor_Print_Graph, v,
471     Ast_Module_String, n)
472     String tmp_str = string_from_format(S("package \\ndirectory=\\\\" "%S\\\" "
473     "\\nlocation: %U:%U"),
474     n->module,
475     n->ast_node.location.line,
476     n->ast_node.location.column);
477     dot_printer_push_insert(v->printer, n,tmp_str);
478 ASTVF_G_END
479
480 ASTVF_BEGIN(AST_SIMPLE_TYPE_CHAR, Ast_Visitor_Print_Graph, v,
481     Ast_Empty, n)
482     simple_type_action(v, n, "char");
483 ASTVF_G_END
484
485 ASTVF_BEGIN(AST_SIMPLE_TYPE_STRING, Ast_Visitor_Print_Graph, v,
486     Ast_Empty, n)
487     simple_type_action(v, n, "string");
488 ASTVF_G_END
489
490 static Ast_Visitor_Print_Graph print_visitor = {
491     .AST_VISITOR_FIELD = AST_VISITOR_STATIC_INIT(),
492     .printer = NULL,
493 };
494
495 void ast_visitor_print_graph_accept_visitor(Ast_Node *root,
496     Const_String file_name_prefix)
497 {
498     print_visitor.printer = dot_printer_init(file_name_prefix,

```

```

499         S("vitaly.parse-tree"), S("LR"));
500
501     root->accept_visitor(root, AST_VISITOR_OF(&print_visitor));
502
503     dot_printer_fin_com_des(print_visitor.printer, S("pdf"));
504 }

```

:

A.3.9 src/ast/ast_visitor_print_graph.h

```

1  #ifndef AST_VISITOR_PRINT_GRAPH_H
2  #define AST_VISITOR_PRINT_GRAPH_H
3
4  #include "ast_visitor.h"
5
6  void ast_visitor_print_graph_accept_visitor(Ast_Node *root,
7      Const_String file_name_prefix);
8
9  #endif // AST_VISITOR_PRINT_GRAPH_H

```

:

A.3.10 src/ast/ast_visitor_symbol_table.c

```

1  #include "ast_visitor_symbol_table.h"
2  #include "symbol_table.h"
3  #include <errno.h>
4  #include <io.h>
5  #include <unistd.h>
6  #include <debug.h>
7  #if 0
8  #include <stdlib.h>
9  #endif
10
11 #undef DEBUG_TYPE
12 #define DEBUG_TYPE symbol-table
13
14 #define CSTR_REC_PREFIX    "R"
15 #define CSTR_FUNC_PREFIX  "F"
16 #define CSTR_TYPE_PREFIX  "T"
17 #define CSTR_VAR_PREFIX   "V"
18 #define CSTR_STMT_LIST_PREFIX "."
19
20 typedef enum Prev_Visited_Type {
21     PREV_VISITED_TYPE_FUNC,
22     PREV_VISITED_TYPE_VAR,
23     PREV_VISITED_TYPE_TYPE_DEF,
24     PREV_VISITED_TYPE_IMPORT
25 } Prev_Visited_Type;
26
27 AST_VISITOR_STRUCT_BEGIN(Ast_Visitor_Symbol_Table)
28     Ast *ast;
29     Ast_Variable_Iden *prev_variable_iden;
30     Symbol_Table *sym_table;
31     Symbol_Table_Node *current_node;
32     String current_unique_name;
33     Symbol_Type_Struct *prev_type;
34     Symbol_Type_Struct *curr_rec_struct;
35     String prev_iden;
36     Prev_Visited_Type prev_visited;
37     Symbol_Type prev_symbol_type;
38     Uns stmt_list_nest;
39     Uns array_nest;
40     Int next_unnamed_type_idx;
41     bool next_var_is_param;
42     bool next_type_in_func_signature;

```

```

43     bool next_type_is_numbered;
44     AST_VISITOR_STRUCT_END(Ast_Visitor_Symbol_Table)
45
46     static void unary_action(Ast_Visitor_Symbol_Table *v, Ast_Node_Unary *n)
47     {
48         ast_node_set_symbol_table_node(&n->ast_node, v->current_node);
49         n->expr->accept_visitor(n->expr, AST_VISITOR_OF(v));
50     }
51
52     static void binary_action(Ast_Visitor_Symbol_Table *v, Ast_Node_Binary *n)
53     {
54         ast_node_set_symbol_table_node(&n->ast_node, v->current_node);
55         n->lhs->accept_visitor(n->lhs, AST_VISITOR_OF(v));
56         n->rhs->accept_visitor(n->rhs, AST_VISITOR_OF(v));
57     }
58
59     static void binary_stmt_action(Ast_Visitor_Symbol_Table *v, Ast_Node_Binary *n)
60     {
61         ast_node_set_symbol_table_node(&n->ast_node, v->current_node);
62         n->lhs->accept_visitor(n->lhs, AST_VISITOR_OF(v));
63         n->rhs->accept_visitor(n->rhs, AST_VISITOR_OF(v));
64     }
65
66     ASTVF_BEGIN(AST_EXPR_LOR, Ast_Visitor_Symbol_Table, v, Ast_Node_Binary, n)
67         binary_action(v, n);
68     ASTVF_END
69
70     ASTVF_BEGIN(AST_EXPR_LAND, Ast_Visitor_Symbol_Table, v, Ast_Node_Binary, n)
71         binary_action(v, n);
72     ASTVF_END
73
74     ASTVF_BEGIN(AST_EXPR_EQ, Ast_Visitor_Symbol_Table, v, Ast_Node_Binary, n)
75         binary_action(v, n);
76     ASTVF_END
77
78     ASTVF_BEGIN(AST_EXPR_NEQ, Ast_Visitor_Symbol_Table, v, Ast_Node_Binary, n)
79         binary_action(v, n);
80     ASTVF_END
81
82     ASTVF_BEGIN(AST_EXPR_GT, Ast_Visitor_Symbol_Table, v, Ast_Node_Binary, n)
83         binary_action(v, n);
84     ASTVF_END
85
86     ASTVF_BEGIN(AST_EXPR_LT, Ast_Visitor_Symbol_Table, v, Ast_Node_Binary, n)
87         binary_action(v, n);
88     ASTVF_END
89
90     ASTVF_BEGIN(AST_EXPR_GTEQ, Ast_Visitor_Symbol_Table, v, Ast_Node_Binary, n)
91         binary_action(v, n);
92     ASTVF_END
93
94     ASTVF_BEGIN(AST_EXPR_LTEQ, Ast_Visitor_Symbol_Table, v, Ast_Node_Binary, n)
95         binary_action(v, n);
96     ASTVF_END
97
98     ASTVF_BEGIN(AST_EXPR_PLUS, Ast_Visitor_Symbol_Table, v, Ast_Node_Binary, n)
99         binary_action(v, n);
100    ASTVF_END
101
102    ASTVF_BEGIN(AST_EXPR_MINUS, Ast_Visitor_Symbol_Table, v, Ast_Node_Binary, n)
103        binary_action(v, n);
104    ASTVF_END
105
106    ASTVF_BEGIN(AST_EXPR_MUL, Ast_Visitor_Symbol_Table, v, Ast_Node_Binary, n)
107        binary_action(v, n);
108    ASTVF_END
109
110    ASTVF_BEGIN(AST_EXPR_DIV, Ast_Visitor_Symbol_Table, v, Ast_Node_Binary, n)
111        binary_action(v, n);
112    ASTVF_END
113
114    ASTVF_BEGIN(AST_EXPR_CAST, Ast_Visitor_Symbol_Table, v, Ast_Node_Binary, n)
115        ast_node_set_symbol_table_node(&n->ast_node, v->current_node);
116

```

```

117     n->lhs->accept_visitor(n->lhs, AST_VISITOR_OF(v));
118
119     symbol_table_insert_location(v->sym_table,
120         v->current_node,
121         v->current_unique_name,
122         v->prev_type,
123         ast_node_get_file_location(AST_NODE_OF(n)));
124
125     n->rhs->accept_visitor(n->rhs, AST_VISITOR_OF(v));
126 ASTVF_END
127
128 ASTVF_BEGIN(AST_EXPR_LNOT, Ast_Visitor_Symbol_Table, v, Ast_Node_Unary, n)
129     unary_action(v, n);
130 ASTVF_END
131
132 ASTVF_BEGIN(AST_EXPR_ABS, Ast_Visitor_Symbol_Table, v, Ast_Node_Unary, n)
133     unary_action(v, n);
134 ASTVF_END
135
136 ASTVF_BEGIN(AST_EXPR_INT, Ast_Visitor_Symbol_Table, v, Ast_Expr_Int, n)
137     ast_node_set_symbol_table_node(&n->ast_node, v->current_node);
138 ASTVF_END
139
140 ASTVF_BEGIN(AST_EXPR_BOOL, Ast_Visitor_Symbol_Table, v, Ast_Expr_Bool, n)
141     ast_node_set_symbol_table_node(&n->ast_node, v->current_node);
142 ASTVF_END
143
144 ASTVF_BEGIN(AST_EXPR_NULL, Ast_Visitor_Symbol_Table, v, Ast_Expr_Null, n)
145     ast_node_set_symbol_table_node(&n->ast_node, v->current_node);
146 ASTVF_END
147
148 ASTVF_BEGIN(AST_VARIABLE_IDEN, Ast_Visitor_Symbol_Table,
149     v, Ast_Variable_Iden, n)
150     ast_node_set_symbol_table_node(&n->ast_node, v->current_node);
151     v->prev_iden = n->iden;
152     v->prev_variable_iden = n;
153 ASTVF_END
154
155 ASTVF_BEGIN(AST_EXPR_ARY_REF, Ast_Visitor_Symbol_Table, v, Ast_Node_Binary, n)
156     binary_action(v, n);
157 ASTVF_END
158
159 ASTVF_BEGIN(AST_EXPR_FUNC_CALL, Ast_Visitor_Symbol_Table, v,
160     Ast_Expr_Func_Call, n)
161     Ast_Node *expr;
162     Vector *vec;
163     ast_node_set_symbol_table_node(&n->ast_node, v->current_node);
164     n->identifier->accept_visitor(n->identifier, AST_VISITOR_OF(v));
165     vec = n->arguments;
166     VECTOR_FOR_EACH_ENTRY(vec, expr)
167         expr->accept_visitor(expr, AST_VISITOR_OF(v));
168 ASTVF_END
169
170 ASTVF_BEGIN(AST_EXPR_DOT_REF, Ast_Visitor_Symbol_Table, v, Ast_Node_Binary, n)
171     binary_action(v, n);
172 ASTVF_END
173
174 ASTVF_BEGIN(AST_EXPR_DIRECT_REF, Ast_Visitor_Symbol_Table, v,
175     Ast_Node_Binary, n)
176     ast_node_set_symbol_table_node(&n->ast_node, v->current_node);
177     if (n->lhs) {
178         n->lhs->accept_visitor(n->lhs, AST_VISITOR_OF(v));
179         symbol_table_insert_location(v->sym_table,
180             v->current_node,
181             v->current_unique_name,
182             v->prev_type,
183             ast_node_get_file_location(AST_NODE_OF(n)));
184     }
185     n->rhs->accept_visitor(n->rhs, AST_VISITOR_OF(v));
186 ASTVF_END
187
188 ASTVF_BEGIN(AST_SIMPLE_TYPE_INT, Ast_Visitor_Symbol_Table, v,
189     Ast_Empty, n)
190     v->prev_symbol_type = SYMBOL_TYPE_INT;

```

```

191     if (v->next_type_in_func_signature)
192         return;
193     ast_node_set_symbol_table_node(&n->ast_node, v->current_node);
194     v->prev_type = symbol_type_int_alloc(v->sym_table);
195 ASTVF_END
196
197 ASTVF_BEGIN(AST_SIMPLE_TYPE_VOID, Ast_Visitor_Symbol_Table, v,
198     Ast_Empty, n)
199     v->prev_symbol_type = SYMBOL_TYPE_VOID;
200     if (v->next_type_in_func_signature)
201         return;
202     ast_node_set_symbol_table_node(&n->ast_node, v->current_node);
203     v->prev_type = symbol_type_void_alloc(v->sym_table);
204 ASTVF_END
205
206 ASTVF_BEGIN(AST_SIMPLE_TYPE_BOOL, Ast_Visitor_Symbol_Table, v,
207     Ast_Empty, n)
208     v->prev_symbol_type = SYMBOL_TYPE_BOOL;
209     if (v->next_type_in_func_signature)
210         return;
211     ast_node_set_symbol_table_node(&n->ast_node, v->current_node);
212     v->prev_type = symbol_type_bool_alloc(v->sym_table);
213 ASTVF_END
214
215 ASTVF_BEGIN(AST_TYPE_IDEN, Ast_Visitor_Symbol_Table, v, Ast_Type_Iden, n)
216     Symbol_Type_Iden *tiden;
217
218     v->prev_symbol_type = SYMBOL_TYPE_IDEN;
219     v->prev_iden = n->iden;
220     if (v->next_type_in_func_signature)
221         return;
222
223     ast_node_set_symbol_table_node(&n->ast_node, v->current_node);
224     v->prev_type = symbol_type_iden_alloc(v->sym_table, n->iden);
225
226     tiden = SYMBOL_TYPE_STRUCT_CONTAINER(v->prev_type, Symbol_Type_Iden);
227     tiden->sym_node = v->current_node;
228     tiden->loc = ast_node_get_file_location(&n->ast_node);
229 ASTVF_END
230
231 ASTVF_BEGIN(AST_VAR_DECL, Ast_Visitor_Symbol_Table, v, Ast_Node_Binary, n)
232     Symbol *duplicate;
233     String identifier;
234     Symbol_Table_Node *current_node;
235     String prev_unique_name;
236
237     if (v->next_type_in_func_signature) {
238         n->rhs->accept_visitor(n->rhs, AST_VISITOR_OF(v));
239         return;
240     }
241
242     ast_node_set_symbol_table_node(&n->ast_node, v->current_node);
243
244     n->lhs->accept_visitor(n->lhs, AST_VISITOR_OF(v));
245     identifier = v->prev_iden;
246
247     Uns hash = string_hash_code(identifier);
248
249     current_node = v->current_node;
250     if (v->next_var_is_param)
251         v->current_node = v->current_node->parent;
252
253     prev_unique_name = v->current_unique_name;
254     v->current_unique_name = string_from_format(S(CSTR_VAR_PREFIX "%S.%S"),
255         prev_unique_name, identifier);
256     n->rhs->accept_visitor(n->rhs, AST_VISITOR_OF(v));
257     v->current_node = current_node;
258
259     duplicate = symbol_table_node_get(v->current_node, identifier, hash,
260         SYMBOL_PROPERTY_VAR);
261
262     v->prev_iden = identifier;
263     if (!duplicate) {
264         String var_name = v->current_unique_name;

```

```

265     ____symbol_table_node_insert(
266         v->current_node,
267         identifier,
268         var_name,
269         hash,
270         v->prev_type,
271         SYMBOL_PROPERTY_VAR,
272         ast_node_get_file_location(n->lhs));
273 } else {
274     duplicate->resolved_type = symbol_type_unknown_alloc(v->sym_table);
275     v->prev_type = duplicate->resolved_type;
276     DLOG("Duplicate variable %S\n", identifier);
277     report_error_location(ast_node_get_file_location(n->lhs), S(
278         "duplicate declaration of variable " QFY("%S")
279         ", previous declaration was in:\n\t%F\n"),
280         identifier, duplicate->location);
281 }
282 string_destroy(v->current_unique_name);
283 v->prev_visited = PREV_VISITED_TYPE_VAR;
284 v->current_unique_name = prev_unique_name;
285 ASTVF_END
286
287 ASTVF_BEGIN(AST_TYPE_DEF, Ast_Visitor_Symbol_Table, v, Ast_Node_Binary, n)
288     String identifier;
289     String prev_unique_name;
290
291     ast_node_set_symbol_table_node(&n->ast_node, v->current_node);
292
293     n->lhs->accept_visitor(n->lhs, AST_VISITOR_OF(v));
294     identifier = v->prev_iden;
295
296     Uns hash = string_hash_code(identifier);
297
298     prev_unique_name = v->current_unique_name;
299     v->current_unique_name = string_from_format(S(CSTR_TYPE_PREFIX "%S.%S"),
300         prev_unique_name, identifier);
301     n->rhs->accept_visitor(n->rhs, AST_VISITOR_OF(v));
302
303     if (v->prev_symbol_type == SYMBOL_TYPE_REC) {
304         Symbol_Type_Rec *rec = SYMBOL_TYPE_STRUCT_CONTAINER(
305             v->prev_type, Symbol_Type_Rec);
306         symbol_type_rec_set_name(rec, identifier);
307     } else if (v->prev_symbol_type == SYMBOL_TYPE_ARY) {
308         Symbol_Type_Ary *ary = SYMBOL_TYPE_STRUCT_CONTAINER(
309             v->prev_type, Symbol_Type_Ary);
310         symbol_type_ary_set_name(ary, identifier);
311     }
312
313     String var_name = v->current_unique_name;
314     ____symbol_table_node_insert(v->current_node, identifier,
315         var_name, hash, v->prev_type,
316         SYMBOL_PROPERTY_TYPE_DEF, ast_node_get_file_location(n->lhs));
317
318     string_destroy(v->current_unique_name);
319     v->prev_iden = identifier;
320     v->prev_visited = PREV_VISITED_TYPE_TYPE_DEF;
321     v->current_unique_name = prev_unique_name;
322 ASTVF_END
323
324 ASTVF_BEGIN(AST_TYPE_ARY, Ast_Visitor_Symbol_Table, v, Ast_Type, n)
325     if (v->next_type_in_func_signature) {
326         ++v->array_nest;
327         n->type->accept_visitor(n->type, AST_VISITOR_OF(v));
328         return;
329     }
330     ast_node_set_symbol_table_node(&n->ast_node, v->current_node);
331     n->type->accept_visitor(n->type, AST_VISITOR_OF(v));
332     v->prev_type = symbol_type_ary_alloc(v->sym_table, v->prev_type,
333         v->current_node->scope_id);
334     v->prev_symbol_type = SYMBOL_TYPE_ARY;
335 ASTVF_END
336
337 ASTVF_BEGIN(AST_TYPE_REC, Ast_Visitor_Symbol_Table, v, Ast_Type_Rec, n)
338     Ast_Node *tmp_node;

```

```

339     Symbol_Type_Struct *prev_rec;
340     Vector *node_vec, *trec_vec1;
341     bool saved_next_type_is_numbered;
342
343     if (v->next_type_in_func_signature) {
344         v->prev_symbol_type = SYMBOL_TYPE_REC;
345         return;
346     }
347
348     ast_node_set_symbol_table_node(&n->ast_node, v->current_node);
349
350     Symbol_Type_Rec *trec;
351     String prev_unique_name = v->current_unique_name;
352     if (v->next_type_is_numbered)
353         v->current_unique_name = string_from_format(S(CSTR_REC_PREFIX "%U%S"),
354             v->next_unnamed_type_idx++,
355             v->current_unique_name);
356     else
357         v->current_unique_name = string_from_format(S(CSTR_REC_PREFIX "%S"),
358             v->current_unique_name);
359
360     Symbol_Type_Struct *type = symbol_type_rec_alloc(v->sym_table,
361         v->current_node->scope_id, v->current_unique_name);
362     trec = SYMBOL_TYPE_STRUCT_CONTAINER(type, Symbol_Type_Rec);
363
364     Int prev_unnamed_type_idx = v->next_unnamed_type_idx;
365     v->next_unnamed_type_idx = 0;
366     saved_next_type_is_numbered = v->next_type_is_numbered;
367     v->next_type_is_numbered = true;
368
369     node_vec = n->extend_list;
370     trec_vec1 = &trec->extended_types;
371     VECTOR_FOR_EACH_ENTRY(node_vec, tmp_node) {
372         tmp_node->accept_visitor(tmp_node, AST_VISITOR_OF(v));
373         vector_append(trec_vec1, v->prev_type);
374     }
375     v->next_type_is_numbered = saved_next_type_is_numbered;
376     v->next_unnamed_type_idx = prev_unnamed_type_idx;
377
378     v->current_node = symbol_table_node_alloc_insert(v->current_node,
379         v->sym_table, SYMBOL_TABLE_NODE_REC, trec);
380     trec->rec_sym_node = v->current_node;
381
382     node_vec = n->body;
383     prev_rec = v->curr_rec_struct;
384     v->curr_rec_struct = type;
385
386     VECTOR_FOR_EACH_ENTRY(node_vec, tmp_node) {
387         tmp_node->accept_visitor(tmp_node, AST_VISITOR_OF(v));
388         if (v->prev_visited == PREV_VISITED_TYPE_FUNC)
389             symbol_type_rec_append_func_identifier(trec, v->prev_iden);
390         else if (v->prev_visited == PREV_VISITED_TYPE_VAR)
391             symbol_type_rec_append_var_identifier(trec, v->prev_iden);
392     }
393     v->curr_rec_struct = prev_rec;
394
395     n->body_node = v->current_node;
396     v->current_node = v->current_node->parent;
397     string_destroy(v->current_unique_name);
398     v->current_unique_name = prev_unique_name;
399     v->prev_symbol_type = SYMBOL_TYPE_REC;
400     v->prev_type = type;
401     ASTVF_END
402
403     static void stmt_list_action(Ast_Visitor_Symbol_Table *v, Ast_Stmt_List *n)
404     {
405         Vector *statements;
406         Ast_Node *stmt;
407
408         ast_node_set_symbol_table_node(&n->ast_node, v->current_node);
409
410         Symbol_Table_Node_Type ntype;
411         if (!v->current_node || v->current_node->type == SYMBOL_TABLE_NODE_GLOBAL)
412             ntype = SYMBOL_TABLE_NODE_GLOBAL;

```



```

413     else
414         ntype = SYMBOL_TABLE_NODE_INTERMEDIATE;
415         v->current_node = symbol_table_node_alloc_insert(v->current_node,
416             v->sym_table, ntype, NULL);
417
418         String prev_unique_name = v->current_unique_name;
419         if (!v->stmt_list_nest) {
420             v->current_unique_name = string_from_format(
421                 S(CSTR_STMT_LIST_PREFIX "%S"),
422                 prev_unique_name);
423         } else {
424             v->current_unique_name = string_from_format(
425                 S(CSTR_STMT_LIST_PREFIX "%U%S"),
426                 v->current_node->scope_id, prev_unique_name);
427         }
428
429         ++v->stmt_list_nest;
430         statements = n->statements;
431         VECTOR_FOR_EACH_ENTRY(statements, stmt)
432             stmt->accept_visitor(stmt, AST_VISITOR_OF(v));
433         --v->stmt_list_nest;
434
435         string_destroy(v->current_unique_name);
436         v->current_unique_name = prev_unique_name;
437         v->current_node = v->current_node->parent;
438     }
439
440     ASTVF_BEGIN(AST_STMT_LIST, Ast_Visitor_Symbol_Table, v, Ast_Stmt_List, n)
441         stmt_list_action(v, n);
442     ASTVF_END
443
444     ASTVF_BEGIN(AST_FIN_STMT_LIST, Ast_Visitor_Symbol_Table, v, Ast_Stmt_List, n)
445         stmt_list_action(v, n);
446     ASTVF_END
447
448     static void func_def_append_type_str(Ast_Visitor_Symbol_Table *v,
449         Symbol_Func_Map *func_map, String_Builder *sb)
450     {
451         Const_String rec_str;
452         if (v->array_nest) {
453             rec_str = S("a%D");
454             if (v->prev_symbol_type != SYMBOL_TYPE_REC) {
455                 do
456                     string_builder_append_char(sb, 'a');
457                 while (--v->array_nest);
458             } else {
459                 v->array_nest = 0;
460             }
461         } else {
462             rec_str = S("r%D");
463         }
464         switch (v->prev_symbol_type) {
465             case SYMBOL_TYPE_INT:
466                 string_builder_append_char(sb, 'i');
467                 break;
468             case SYMBOL_TYPE_BOOL:
469                 string_builder_append_char(sb, 'b');
470                 break;
471             case SYMBOL_TYPE_CHAR:
472                 string_builder_append_char(sb, 'c');
473                 break;
474             case SYMBOL_TYPE_STRING:
475                 string_builder_append_char(sb, 's');
476                 break;
477             case SYMBOL_TYPE_REC:;
478                 String str = string_from_format(rec_str,
479                     func_map ? func_map->unnamed_type_count++ : 0);
480                 string_builder_append(sb, str);
481                 string_destroy(str);
482                 break;
483             case SYMBOL_TYPE_IDEN:
484                 string_builder_append_char(sb, '$');
485                 string_builder_append(sb, v->prev_iden);
486                 string_builder_append_char(sb, '$');

```

```

487         break;
488     default:
489         assert(false);
490     }
491 }
492
493 static void func_symbol_table(Ast_Visitor_Symbol_Table *v, Ast_Func_Def *n)
494 {
495     Vector *param_node_vec;
496     Ast_Node *param;
497     Symbol_Type_Func *func_type;
498     String identifier;
499     String prev_unique_name;
500     Uns hash_code;
501     bool saved_next_var_isc;
502     bool saved_next_type_in_sig;
503     bool saved_next_type_is_numbered;
504     bool is_extern_c;
505     bool destroy_unique_name;
506     Ast_Variable_Iden *func_iden_node;
507
508     ast_node_set_symbol_table_node(&n->ast_node, v->current_node);
509
510     destroy_unique_name = false;
511     is_extern_c = false;
512
513     if (n->extern_type) {
514         n->extern_type->accept_visitor(n->extern_type, AST_VISITOR_OF(v));
515         if (string_compare_nocase(v->prev_iden, S("c"))) {
516             report_error_location(ast_node_get_file_location(n->extern_type),
517                                 S("unrecognized extern identifier " QFY("%S") "\n"),
518                                 v->prev_iden);
519         } else {
520             is_extern_c = true;
521         }
522     }
523
524     n->iden->accept_visitor(n->iden, AST_VISITOR_OF(v));
525     identifier = v->prev_iden;
526     func_iden_node = v->prev_variable_iden;
527
528     hash_code = string_hash_code(identifier);
529     Symbol_Func_Map *func_map = symbol_table_node_get_func_map(v->current_node,
530                                                                identifier, hash_code);
531
532     Int initial_unnamed_type_count;
533     if (func_map)
534         initial_unnamed_type_count = func_map->unnamed_type_count;
535     else
536         initial_unnamed_type_count = 0;
537
538     func_type = SYMBOL_TYPE_STRUCT_CONTAINER(
539         symbol_type_func_alloc(v->sym_table, is_extern_c),
540         Symbol_Type_Func);
541
542     prev_unique_name = v->current_unique_name;
543
544     String_Builder unique_name = STRING_BUILDER_INIT();
545     assert(!v->array_nest);
546
547     saved_next_type_is_numbered = v->next_type_is_numbered;
548     saved_next_type_in_sig = v->next_type_in_func_signature;
549     v->next_type_in_func_signature = true;
550     v->next_type_is_numbered = true;
551     param_node_vec = n->parameters;
552     VECTOR_FOR_EACH_ENTRY(param_node_vec, param) {
553         param->accept_visitor(param, AST_VISITOR_OF(v));
554         func_def_append_type_str(v, func_map, &unique_name);
555     }
556     v->next_type_in_func_signature = saved_next_type_in_sig;
557     v->next_type_is_numbered = saved_next_type_is_numbered;
558
559     Symbol_Table_Node *body_node = symbol_table_node_alloc_insert(
560         v->current_node, v->sym_table, SYMBOL_TABLE_NODE_FUNC, NULL);

```

```

561     v->current_node = body_node;
562
563     string_builder_append(&unique_name, prev_unique_name);
564     string_builder_append_char(&unique_name, '.');
565     string_builder_append(&unique_name, identifier);
566
567     v->current_unique_name = string_from_format(S(CSTR_FUNC_PREFIX "%S"),
568         string_builder_const_str(&unique_name));
569     string_builder_clear(&unique_name);
570
571
572     Int prev_unnamed_type_idx = v->next_unnamed_type_idx;
573     saved_next_type_is_numbered = v->next_type_is_numbered;
574
575     saved_next_var_isp = v->next_var_is_param;
576     v->next_type_is_numbered = true;
577     v->next_var_is_param = true;
578     v->next_unnamed_type_idx = initial_unnamed_type_count;
579     VECTOR_FOR_EACH_ENTRY(param_node_vec, param) {
580         param->accept_visitor(param, AST_VISITOR_OF(v));
581         symbol_type_func_append_param_identifier(func_type, v->prev_iden);
582     }
583     v->next_var_is_param = saved_next_var_isp;
584
585     v->current_node = body_node->parent;
586
587     Int ret_was_unique;
588     (void)ret_was_unique;
589     DEBUGT(def, ret_was_unique = v->next_unnamed_type_idx);
590
591     n->return_type->accept_visitor(n->return_type, AST_VISITOR_OF(v));
592     func_type->return_type = v->prev_type;
593     v->current_node = body_node;
594     v->next_type_is_numbered = saved_next_type_is_numbered;
595
596     DEBUGT(def,
597         ret_was_unique = v->next_unnamed_type_idx - ret_was_unique;
598     );
599
600     if (n->statements)
601         n->statements->accept_visitor(n->statements, AST_VISITOR_OF(v));
602
603     func_type->body_sym_node = v->current_node;
604     v->current_node = v->current_node->parent;
605
606     File_Location *loc = ast_node_get_file_location(AST_NODE_OF(n));
607
608     String var_name;
609     if (v->current_node->type == SYMBOL_TABLE_NODE_GLOBAL &&
610         v->stmt_list_nest == 1 &&
611         !string_compare(identifier, MAIN_FUNC_STR)) {
612         destroy_unique_name = true;
613         var_name = string_duplicate(MAIN_FUNC_STR);
614     } else if (is_extern_c) {
615         var_name = string_duplicate(identifier);
616         destroy_unique_name = true;
617     } else {
618         var_name = v->current_unique_name;
619     }
620     v->prev_type = SYMBOL_TYPE_STRUCT_OF_CONTAINER(func_type);
621     __symbol_table_node_insert(v->current_node, identifier, var_name,
622         hash_code, v->prev_type, SYMBOL_PROPERTY_FUNC, loc);
623
624     if (!func_map) {
625         func_map = symbol_table_node_get_func_map(v->current_node, identifier,
626             hash_code);
627         assert(func_map);
628     } else {
629         assert(func_map->unnamed_type_count == v->next_unnamed_type_idx -
630             ret_was_unique);
631     }
632     func_map->unnamed_type_count = v->next_unnamed_type_idx;
633     if (is_extern_c)
634         func_map->has_extern_c = true;

```

```

635
636     string_destroy(func_iden_node->iden);
637     func_iden_node->iden = var_name;
638
639     if (destroy_unique_name)
640         string_destroy(v->current_unique_name);
641
642     v->next_unnamed_type_idx = prev_unnamed_type_idx;
643     v->prev_iden = var_name;
644     v->prev_visited = PREV_VISITED_TYPE_FUNC;
645     v->current_unique_name = prev_unique_name;
646 }
647
648 ASTVF_BEGIN(AST_EXT_FUNC_DECL, Ast_Visitor_Symbol_Table, v, Ast_Func_Def, n)
649     func_symbol_table(v, n);
650 ASTVF_END
651
652 ASTVF_BEGIN(AST_FIN_FUNC_DEF, Ast_Visitor_Symbol_Table, v, Ast_Func_Def, n)
653     /* Errors regarding finalize function is handled in type checker
654      * so we don't do much different here. */
655     if (v->current_node->type == SYMBOL_TABLE_NODE_REC)
656         v->current_node->has_finalize_func = true;
657     func_symbol_table(v, n);
658 ASTVF_END
659
660 ASTVF_BEGIN(AST_REC_FUNC_DEF, Ast_Visitor_Symbol_Table, v, Ast_Func_Def, n)
661     /* Errors regarding record function is handled in type checker
662      * so we don't do much different here. */
663     if (v->current_node->type == SYMBOL_TABLE_NODE_REC)
664         v->current_node->has_record_func = true;
665     func_symbol_table(v, n);
666 ASTVF_END
667
668 ASTVF_BEGIN(AST_FUNC_DEF, Ast_Visitor_Symbol_Table, v, Ast_Func_Def, n)
669     func_symbol_table(v, n);
670 ASTVF_END
671
672 ASTVF_BEGIN(AST_IF_STMT, Ast_Visitor_Symbol_Table, v, Ast_Node_Binary, n)
673     binary_stmt_action(v, n);
674 ASTVF_END
675
676 ASTVF_BEGIN(AST_IF_ELSE_STMT, Ast_Visitor_Symbol_Table, v, Ast_Node_Ternary, n)
677     ast_node_set_symbol_table_node(&n->ast_node, v->current_node);
678     n->lhs->accept_visitor(n->lhs, AST_VISITOR_OF(v));
679     n->mid->accept_visitor(n->mid, AST_VISITOR_OF(v));
680     n->rhs->accept_visitor(n->rhs, AST_VISITOR_OF(v));
681 ASTVF_END
682
683 ASTVF_BEGIN(AST_ALLOC_ARY, Ast_Visitor_Symbol_Table, v, Ast_Node_Binary, n)
684     binary_action(v, n);
685 ASTVF_END
686
687 ASTVF_BEGIN(AST_ALLOC_REC, Ast_Visitor_Symbol_Table, v, Ast_Node_Unary, n)
688     unary_action(v, n);
689 ASTVF_END
690
691 ASTVF_BEGIN(AST_ALLOC_REC_CALL, Ast_Visitor_Symbol_Table, v,
692     Ast_Node_Binary, n)
693     binary_action(v, n);
694 ASTVF_END
695
696 ASTVF_BEGIN(AST_DELETE, Ast_Visitor_Symbol_Table, v, Ast_Node_Unary, n)
697     unary_action(v, n);
698 ASTVF_END
699
700 ASTVF_BEGIN(AST_WHILE_STMT, Ast_Visitor_Symbol_Table, v, Ast_Node_Binary, n)
701     binary_stmt_action(v, n);
702 ASTVF_END
703
704 ASTVF_BEGIN(AST_RETURN_STMT, Ast_Visitor_Symbol_Table, v, Ast_Node_Unary, n)
705     unary_action(v, n);
706 ASTVF_END
707
708 ASTVF_BEGIN(AST_WRITE_STMT, Ast_Visitor_Symbol_Table, v, Ast_Node_Unary, n)

```

```

709     unary_action(v, n);
710 ASTVF_END
711
712 ASTVF_BEGIN(AST_ASSIGNMENT, Ast_Visitor_Symbol_Table, v, Ast_Node_Binary, n)
713     binary_action(v, n);
714 ASTVF_END
715
716 ASTVF_BEGIN(AST_EXPR_CHAR, Ast_Visitor_Symbol_Table, v, Ast_Expr_Char, n)
717     ast_node_set_symbol_table_node(&n->ast_node, v->current_node);
718 ASTVF_END
719
720 ASTVF_BEGIN(AST_EXPR_STRING, Ast_Visitor_Symbol_Table, v, Ast_Expr_String, n)
721     ast_node_set_symbol_table_node(&n->ast_node, v->current_node);
722 ASTVF_END
723
724 ASTVF_BEGIN(AST_IMPORT_STRING, Ast_Visitor_Symbol_Table, v,
725     Ast_Module_String, n)
726     Symbol_Table_Node *node = v->current_node;
727     ast_node_set_symbol_table_node(AST_NODE_OF(n), node);
728     Const_String package = ast_get_package(v->ast);
729     String dependency;
730     DLOG("lookup real import name for '%S' ...", n->module);
731     if (package) {
732         String vit = string_from_format(S("%S.vit"), n->module);
733         String viti = string_from_format(S("%S.viti"), n->module);
734         char *cwd;
735         if (!(cwd = getcwd(NULL, 0)))
736             fatal_error(S("unable to get current working directory [%m]\n"));
737
738         if (chdir(string_to_cstr(ast_get_dirname(v->ast))))
739             fatal_error(S("unable to change working directory [%m]\n"));
740
741         if (file_access_read(vit) || file_access_read(viti)) {
742             dependency = string_dir_concat(package, n->module);
743             //vector_append(&node->import_dependencies,
744             //    string_dir_concat(package, n->module));
745         } else {
746             dependency = string_duplicate(n->module);
747             //vector_append(&node->import_dependencies,
748             //    string_duplicate(n->module));
749         }
750
751         if (chdir(cwd))
752             fatal_error(S("unable to reset working directory [%m]\n"));
753
754         free_mem(cwd);
755         string_destroy(vit);
756         string_destroy(viti);
757     } else {
758         dependency = string_duplicate(n->module);
759         //vector_append(&node->import_dependencies,
760         //    string_duplicate(n->module));
761     }
762
763     DLOG("result = '%S'\n", dependency);
764     //DLOG("result = '%S'\n", vector_peek_last(
765     //    &node->import_dependencies));
766
767     // n->dep_idx = vector_size(&v->current_node->import_dependencies_loc);
768     // vector_append(&node->import_dependencies_loc,
769     //    ast_node_get_file_location(AST_NODE_OF(n)));
770     String str;
771     int idx = 0;
772     VECTOR_FOR_EACH_ENTRY(&node->import_dependencies, str) {
773         if (!string_compare(dependency, str)) {
774             string_destroy(dependency);
775             goto out;
776         }
777     }
778     idx++;
779 }
780 vector_append(&node->import_dependencies, dependency);
781 vector_append(&node->import_dependencies_loc,
782     ast_node_get_file_location(AST_NODE_OF(n)));

```

```

783 out:
784     n->dep_idx = idx;
785     v->prev_visited = PREV_VISITED_TYPE_IMPORT;
786 ASTVF_END
787
788 ASTVF_BEGIN(AST_PACKAGE_STRING, Ast_Visitor_Symbol_Table, v,
789     Ast_Module_String, n)
790     ast_node_set_symbol_table_node(AST_NODE_OF(n), v->current_node);
791
792     char *abs_path = realpath(string_to_cstr(ast_get_dirname(v->ast)), NULL);
793
794     if (!string_ends_with(S(abs_path), n->module)) {
795         String tmp_module = string_duplicate(n->module);
796         string_replace_all(tmp_module, '/', '.');
797         report_error_location(ast_node_get_file_location(AST_NODE_OF(n)),
798             S("expected file with package name " QFY("%S")
799               " inside directory structure " QFY("%S") "\n"),
800             tmp_module, n->module);
801         string_destroy(tmp_module);
802     } else {
803         String prev_unique = v->current_unique_name;
804         v->current_unique_name = string_from_format(S(".%S%S"),
805             n->module, prev_unique);
806         string_replace_all(v->current_unique_name, '/', '.');
807         string_destroy(prev_unique);
808         v->ast->package = n->module;
809     }
810     free_mem(abs_path);
811 ASTVF_END
812
813 ASTVF_BEGIN(AST_SIMPLE_TYPE_CHAR, Ast_Visitor_Symbol_Table,
814     v, Ast_Empty, n)
815     v->prev_symbol_type = SYMBOL_TYPE_CHAR;
816     if (v->next_type_in_func_signature)
817         return;
818     ast_node_set_symbol_table_node(&n->ast_node, v->current_node);
819     v->prev_type = symbol_type_char_alloc(v->sym_table);
820 ASTVF_END
821
822 ASTVF_BEGIN(AST_SIMPLE_TYPE_STRING, Ast_Visitor_Symbol_Table,
823     v, Ast_Empty, n)
824     v->prev_symbol_type = SYMBOL_TYPE_STRING;
825     if (v->next_type_in_func_signature)
826         return;
827     ast_node_set_symbol_table_node(&n->ast_node, v->current_node);
828     v->prev_type = symbol_type_string_alloc(v->sym_table);
829 ASTVF_END
830
831 ASTVF_BEGIN(AST_FUNC_BODY, Ast_Visitor_Symbol_Table, v, Ast_Stmt_List, n)
832     Vector *stmt_list;
833     ast_node_set_symbol_table_node(&n->ast_node, v->current_node);
834
835     Ast_Node *stmt;
836     stmt_list = n->statements;
837     VECTOR_FOR_EACH_ENTRY(stmt_list, stmt)
838         stmt->accept_visitor(stmt, AST_VISITOR_OF(v));
839 ASTVF_END
840
841 ASTVF_BEGIN(AST_REC_SELF_PTR, Ast_Visitor_Symbol_Table, v, Ast_Empty, n)
842     ast_node_set_symbol_table_node(&n->ast_node, v->current_node);
843     if (v->curr_rec_struct) {
844         symbol_table_insert_location(v->sym_table,
845             v->current_node,
846             v->current_unique_name,
847             v->curr_rec_struct,
848             ast_node_get_file_location(AST_NODE_OF(n)));
849     }
850 ASTVF_END
851
852 static Ast_Visitor_Symbol_Table sym_table_visitor = {
853     .AST_VISITOR_FIELD = AST_VISITOR_STATIC_INIT()
854 };
855
856 void __symbol_table_resolve(Symbol_Table *t);

```

```

857
858 Symbol_Table *ast_visitor_symbol_table_gen(Ast *ast)
859 {
860     if (!ast_is_valid(ast))
861         return NULL;
862
863     Ast_Node *root = ast_get_root(ast);
864
865     sym_table_visitor.sym_table = symbol_table_alloc(ast_get_file_name(ast));
866     sym_table_visitor.ast = ast;
867
868     String basename = string_basename(ast_get_file_name(ast));
869
870     assert(string_ends_with(basename, SOURCE_SUFFIX_STR) ||
871            string_ends_with(basename, INTERFACE_SUFFIX_STR));
872
873     sym_table_visitor.current_unique_name = string_to_module_name(basename);
874     string_destroy(basename);
875
876     root->accept_visitor(root, AST_VISITOR_OF(&sym_table_visitor));
877
878     string_destroy(sym_table_visitor.current_unique_name);
879
880     return sym_table_visitor.sym_table;
881 }

```

:

A.3.11 src/ast/ast_visitor_symbol_table.h

```

1  #ifndef AST_VISITOR_SYMBOL_TABLE_H
2  #define AST_VISITOR_SYMBOL_TABLE_H
3
4  #include "ast_visitor.h"
5
6  typedef struct Symbol_Table Symbol_Table;
7
8  /* Returns NULL is the Ast is invalid. */
9  Symbol_Table *ast_visitor_symbol_table_gen(Ast *ast);
10
11 #endif // AST_VISITOR_SYMBOL_TABLE_H

```

:

A.3.12 src/ast/symbol_table.c

```

1  #include "symbol_table.h"
2  #include <string_builder.h>
3  #include <pointer_hash.h>
4  #include <main.h>
5  #include <import_handler.h>
6
7  #undef DEBUG_TYPE
8  #define DEBUG_TYPE symbol-table
9
10 static inline bool symbol_type_names_equal(String lhs_name, Uns lhs_scope,
11 String rhs_name, Uns rhs_scope)
12 {
13     if (lhs_scope != rhs_scope)
14         return false;
15     return !string_compare(lhs_name, rhs_name);
16 }
17
18 static inline bool symbol_types_equal(Symbol_Type_Struct *lhs,
19 Symbol_Type_Struct *rhs)
20 {
21     return lhs->methods->same_type(lhs, rhs);

```

```

22 }
23
24 static inline bool symbol_type_vectors_same_type(Vector *lhs, Vector *rhs)
25 {
26     Uns size = vector_size(lhs);
27     if (size != vector_size(rhs))
28         return false;
29     while (size--) {
30         if (!symbol_types_equal(vector_get(lhs, size),
31                                 vector_get(rhs, size)))
32             return false;
33     }
34     return true;
35 }
36
37 static inline bool symbol_type_vector_idens_identical(Vector *lhs,
38 Vector *rhs)
39 {
40     Uns size = vector_size(lhs);
41     if (size != vector_size(rhs))
42         return false;
43     while (size--) {
44         Const_String lhs_str = STRING_AFTER_LAST(vector_get(lhs, size), '.');
45         Const_String rhs_str = STRING_AFTER_LAST(vector_get(rhs, size), '.');
46 #if 0
47         Const_String lhs_str = vector_get(lhs, size);
48         Const_String rhs_str = vector_get(rhs, size);
49 #endif
50         if (string_compare(lhs_str, rhs_str))
51             return false;
52     }
53     return true;
54 }
55
56 bool __symbol_var_is_in_scope(File_Location *var_loc,
57 Symbol *lookup_sym, Symbol_Table_Node *child_sym_node)
58 {
59     Symbol_Table_Node_Type nt = symbol_get_symbol_table_node_type(lookup_sym);
60     if (nt == SYMBOL_TABLE_NODE_REC)
61         return true;
62     if (nt == SYMBOL_TABLE_NODE_IMPORT)
63         return true;
64     if (file_location_cmp_lncol(var_loc, lookup_sym->location) >= 0)
65         return true;
66
67     Symbol_Table_Node *look_sym_func = lookup_sym->sym_node;
68     while (look_sym_func->type == SYMBOL_TABLE_NODE_INTERMEDIATE)
69         look_sym_func = look_sym_func->parent;
70     Symbol_Table_Node *child_func = child_sym_node;
71     while (child_func->type == SYMBOL_TABLE_NODE_INTERMEDIATE)
72         child_func = child_func->parent;
73
74 #ifdef NEST_FUNCS_ACCESS_LATER_DECL
75     if (child_func->scope_id != look_sym_func->scope_id) {
76         assert(child_func->scope_id > look_sym_func->scope_id);
77         return true;
78     }
79 #endif
80     return false;
81 }
82
83
84 static inline Type_Def_Symbol *symbol_table_node_lookup_type_def(
85 Symbol_Table_Node *sym_node, String iden, Uns hash_code)
86 {
87     Symbol *sym = __symbol_table_node_lookup(sym_node, iden, hash_code,
88 SYMBOL_PROPERTY_TYPE_DEF);
89     if (sym)
90         return TYPE_DEF_SYMBOL_OF_SYMBOL(sym);
91     return NULL;
92 }
93
94 static inline Symbol_Type_Struct *__symbol_resolve(Symbol *sym,
95 Symbol_Table *t)

```



```

96 {
97     return sym->resolved_type->methods->resolve(sym->resolved_type, t);
98 }
99
100 static bool symbol_type_rec_structs_equal(Symbol_Type_Rec *rhs,
101     Symbol_Type_Rec *lhs)
102 {
103     bool ret = false;
104     #if 0
105     if (!vector_is_empty(&rhs->func_identifiers) ||
106         !vector_is_empty(&lhs->func_identifiers)) {
107         ret = rhs->rec_sym_node->scope_id == lhs->rec_sym_node->scope_id;
108         goto out;
109     }
110     #endif
111
112     if (!symbol_type_vector_idens_identical(&lhs->var_identifiers,
113         &rhs->var_identifiers))
114         goto out;
115
116     if (!symbol_type_vector_idens_identical(&lhs->func_identifiers,
117         &rhs->func_identifiers))
118         goto out;
119
120     if (!symbol_type_vectors_same_type(&lhs->var_types,
121         &rhs->var_types))
122         goto out;
123
124     if (!symbol_type_vectors_same_type(&lhs->func_types,
125         &rhs->func_types))
126         goto out;
127
128     if (!symbol_type_vectors_same_type(&lhs->extended_types,
129         &rhs->extended_types))
130         goto out;
131
132     ret = true;
133
134 out:
135     return ret;
136 }
137
138 static inline bool symbol_type_arys_equal(Symbol_Type_Ary *lhs_ary,
139     Symbol_Type_Ary *rhs_ary)
140 {
141     for (;;) {
142         Symbol_Type_Struct *lhs_tmp = lhs_ary->ary_type;
143         Symbol_Type_Struct *rhs_tmp = rhs_ary->ary_type;
144         Symbol_Type tmp_t = lhs_tmp->methods->get_type();
145
146         if (tmp_t == SYMBOL_TYPE_UNKNOWN)
147             return true;
148         Symbol_Type rhs_t = rhs_tmp->methods->get_type();
149         if (rhs_t == SYMBOL_TYPE_UNKNOWN)
150             return true;
151
152         if (rhs_t != tmp_t)
153             return false;
154
155         if (tmp_t != SYMBOL_TYPE_ARY) {
156             if (tmp_t == SYMBOL_TYPE_REC) {
157                 Symbol_Type_Rec *lhs_rec, *rhs_rec;
158                 lhs_rec = SYMBOL_TYPE_STRUCT_CONTAINER(lhs_tmp,
159                     Symbol_Type_Rec);
160                 rhs_rec = SYMBOL_TYPE_STRUCT_CONTAINER(rhs_tmp,
161                     Symbol_Type_Rec);
162                 if (!lhs_rec->rec_name || !rhs_rec->rec_name)
163                     return false;
164                 else
165                     return lhs_tmp->methods->same_type(lhs_tmp, rhs_tmp);
166             } else {
167                 return lhs_tmp->methods->same_type(lhs_tmp, rhs_tmp);
168             }
169         }

```

```

170     } else {
171         lhs_ary = SYMBOL_TYPE_STRUCT_CONTAINER(lhs_tmp,
172         Symbol_Type_Ary);
173         rhs_ary = SYMBOL_TYPE_STRUCT_CONTAINER(rhs_tmp,
174         Symbol_Type_Ary);
175     }
176 }
177 }
178
179 static inline bool symbol_type_rec_is_imported(Symbol_Type_Rec *rec)
180 {
181     assert(rec->rec_sym_node);
182     assert(rec->rec_sym_node->parent);
183     return rec->rec_sym_node->parent->type == SYMBOL_TABLE_NODE_IMPORT;
184 }
185
186 bool __type_def_symbol_types_equal(Symbol_Type_Struct *lhs,
187     Symbol_Type_Struct *rhs)
188 {
189     Symbol_Type lhs_type = lhs->methods->get_type();
190     Symbol_Type rhs_type = rhs->methods->get_type();
191
192     if (lhs_type == rhs_type) {
193         switch (lhs_type) {
194             case SYMBOL_TYPE_ARRAY:
195                 do {
196                     Symbol_Type_Ary *lhs_ary = SYMBOL_TYPE_STRUCT_CONTAINER(lhs,
197                     Symbol_Type_Ary);
198                     Symbol_Type_Ary *rhs_ary = SYMBOL_TYPE_STRUCT_CONTAINER(rhs,
199                     Symbol_Type_Ary);
200                     lhs = lhs_ary->ary_type;
201                     rhs = rhs_ary->ary_type;
202                     lhs_type = lhs->methods->get_type();
203                     rhs_type = rhs->methods->get_type();
204
205                 } while (lhs_type == SYMBOL_TYPE_ARRAY &&
206                     rhs_type == SYMBOL_TYPE_ARRAY);
207                 if (lhs_type != SYMBOL_TYPE_REC ||
208                     rhs_type != SYMBOL_TYPE_REC)
209                     break;
210                 /* Fall through */
211             case SYMBOL_TYPE_REC:
212                 Symbol_Type_Rec *lhs_rec = SYMBOL_TYPE_STRUCT_CONTAINER(lhs,
213                 Symbol_Type_Rec);
214                 Symbol_Type_Rec *rhs_rec = SYMBOL_TYPE_STRUCT_CONTAINER(rhs,
215                 Symbol_Type_Rec);
216
217                 if (vector_is_empty(&lhs_rec->func_identifiers) &&
218                     vector_is_empty(&rhs_rec->func_identifiers))
219                     break;
220
221                 if ((!symbol_type_rec_is_imported(lhs_rec) &&
222                     !symbol_type_rec_is_imported(rhs_rec))) {
223                     return lhs_rec->rec_sym_node->scope_id ==
224                     rhs_rec->rec_sym_node->scope_id;
225                 } else {
226                     return !string_compare(lhs_rec->unique_name,
227                     rhs_rec->unique_name);
228                 }
229
230             default:
231                 break;
232         }
233     }
234
235     return lhs->methods->same_type(lhs, rhs);
236 }
237
238 static inline Symbol_Type_Struct *symbol_type_get_rec_cycle(Symbol *symbol,
239     Symbol_Table *t)
240 {
241     Symbol_Type_Struct *ret = NULL;
242
243     if (symbol->resolved_type->methods->get_type() == SYMBOL_TYPE_REC) {

```

```

244     Symbol_Type_Rec *rec = SYMBOL_TYPE_STRUCT_CONTAINER(
245         symbol->resolved_type, Symbol_Type_Rec);
246
247     assert(t->rec_cycle_type != SYMBOL_REC_CYCLE_MARK_NONE &&
248         rec->cycle_mark != SYMBOL_REC_CYCLE_MARK_NONE);
249
250     /* We have detected a record cycle.
251      * Find out whether the cycle is allowed. */
252     if (t->rec_cycle_type == SYMBOL_REC_CYCLE_ALLOWED ||
253         rec->cycle_mark == SYMBOL_REC_CYCLE_ALLOWED) {
254         /* Mark that this results in a cycle which
255          * is allowed and return. */
256         assert(rec->rec_name);
257         ret = symbol_type_cycle_alloc(t, rec->rec_name);
258     }
259 }
260
261 return ret;
262 }
263
264 static inline Symbol_Type_Struct *symbol_type_get_ary_cycle(Symbol *symbol,
265     Symbol_Table *t)
266 {
267     Symbol_Type_Ary *curr, *ary;
268     Symbol_Type_Struct *ret = NULL;
269     if (symbol->resolved_type->methods->get_type() ==
270         SYMBOL_TYPE_ARY) {
271
272         ary = SYMBOL_TYPE_STRUCT_CONTAINER(symbol->resolved_type,
273             Symbol_Type_Ary);
274         curr = ary;
275         do {
276             curr = SYMBOL_TYPE_STRUCT_CONTAINER(curr->ary_type,
277                 Symbol_Type_Ary);
278         } while (curr->sym_struct.methods->get_type() == SYMBOL_TYPE_ARY);
279
280         if (curr->sym_struct.methods->get_type() == SYMBOL_TYPE_REC)
281             ret = symbol_type_cycle_alloc(t, ary->ary_name);
282     }
283
284     return ret;
285 }
286
287 static inline void report_conflicting_type_definitions(Type_Def_Symbol *sym)
288 {
289     Double_List *list;
290     Double_List_Node *dbnode;
291     Type_Def_Symbol *tmp_sym;
292
293     if (!is_error_reported_here(sym->symbol.location)) {
294         if (sym->symbol.sym_node->type != SYMBOL_TABLE_NODE_IMPORT) {
295             String str = string_from_format(
296                 S("conflicting type definitions for type " QFY("%1$S")
297                     " which is also defined in:\n"),
298                 STRING_AFTER_LAST(sym->symbol.identifier, '.'));
299
300             list = &sym->dbnode;
301             DOUBLE_LIST_FOR_EACH(list, dbnode) {
302                 tmp_sym = TYPE_DEF_SYMBOL_OF_DBNODE(dbnode);
303                 string_append_format(str, S("\t%f\n"),
304                     tmp_sym->symbol.location);
305             }
306
307             report_error_location(sym->symbol.location, str);
308             string_destroy(str);
309         } else {
310             String str = string_from_format(
311                 S("conflicting imported type definitions for type "
312                     QFY("%S") " which is imported from:\n\t%S\n"),
313                 sym->symbol.identifier, sym->symbol.unique_name);
314
315             list = &sym->dbnode;
316             DOUBLE_LIST_FOR_EACH(list, dbnode) {
317                 tmp_sym = TYPE_DEF_SYMBOL_OF_DBNODE(dbnode);

```

```

318         string_append_format(str, S("\t%S\n"),
319                             tmp_sym->symbol.unique_name);
320     }
321
322     report_error_location(sym->symbol.location, str);
323     string_destroy(str);
324 }
325 }
326 }
327
328 static Symbol_Type_Struct *___type_def_symbol_resolve(Type_Def_Symbol *symbol,
329 Symbol_Table *t)
330 {
331     Symbol_Type_Struct *ret;
332     Symbol_Type_Struct *tmp_type;
333     Type_Def_Symbol *next_sym = symbol;
334     Double_List *dblist;
335     Double_List_Node *dbnode;
336
337     bool report_cycle = symbol->report_cycle;
338     symbol->report_cycle = false;
339
340     if (symbol->cycle_marked) {
341         ret = symbol_type_get_rec_cycle(&symbol->symbol, t);
342         if (ret)
343             goto out;
344         ret = symbol_type_get_ary_cycle(&symbol->symbol, t);
345         if (ret)
346             goto out;
347
348         DOUBLE_LIST_FOR_EACH(&symbol->dbnode, dbnode) {
349             next_sym = TYPE_DEF_SYMBOL_OF_DBNODE(dbnode);
350             if (!next_sym->cycle_marked) {
351                 goto no_cycle;
352             } else {
353                 ret = symbol_type_get_rec_cycle(&next_sym->symbol, t);
354                 if (ret)
355                     goto out;
356                 ret = symbol_type_get_ary_cycle(&next_sym->symbol, t);
357                 if (ret)
358                     goto out;
359             }
360         }
361
362         /* Cycle in type definitions is detected.
363          * Return NULL to notify about this. */
364         goto out;
365     }
366
367 no_cycle:
368     next_sym->cycle_marked = true;
369     ret = next_sym->symbol.resolved_type->methods->resolve(
370         next_sym->symbol.resolved_type, t);
371     next_sym->cycle_marked = false;
372
373     dblist = &next_sym->dbnode;
374     /* Loop through type symbols with identifier ==
375      * symbol->symbol.identifier == next_sym->symbol.identifier. */
376     DOUBLE_LIST_FOR_EACH(dblist, dbnode) {
377         next_sym = TYPE_DEF_SYMBOL_OF_DBNODE(dbnode);
378
379         if (next_sym->cycle_marked) {
380             if (!ret) {
381                 if ((ret = symbol_type_get_rec_cycle(&next_sym->symbol, t)))
382                     break;
383                 else
384                     continue;
385             } else {
386                 break;
387             }
388         }
389
390         next_sym->cycle_marked = true;
391         tmp_type = next_sym->symbol.resolved_type->methods->resolve(

```

```

392         next_sym->symbol.resolved_type, t);
393     next_sym->cycle_marked = false;
394
395     if (!ret) {
396         ret = tmp_type;
397     } else if (t->last_resolve_pass &&
398               tmp_type && !__type_def_symbol_types_equal(tmp_type, ret)) {
399         ret = symbol_type_unknown_alloc(t);
400         report_conflicting_type_definitios(next_sym);
401     }
402
403 }
404
405 if (!ret && report_cycle) {
406     /* We were not able to resolve the symbol + we have found a cycle
407      * in the type definitions. We notify the user about this here. */
408
409     if (!is_error_reported_here(symbol->symbol.location)) {
410         if (symbol_get_symbol_table_node_type(&symbol->symbol) !=
411             SYMBOL_TABLE_NODE_IMPORT)
412             report_error_location(symbol->symbol.location,
413                                  S("cycle in type definitions invloving " QFY("%S") "\n"),
414                                  STRING_AFTER_DOT(symbol->symbol.identifier));
415         else {
416             report_error_location(symbol->symbol.location,
417                                  S("cycle in imported type definitions invloving "
418                                     QFY("%S") "\n"),
419                                  STRING_AFTER_DOT(symbol->symbol.identifier));
420         }
421     }
422     ret = symbol_type_unknown_alloc(t);
423 }
424
425 out:
426     return ret;
427 }
428
429 static inline Symbol_Type_Struct *symbol_resolve(Symbol *sym, Symbol_Table *t)
430 {
431     if (t->current_resolve_property == SYMBOL_PROPERTY_TYPE_DEF)
432         return __type_def_symbol_resolve(TYPE_DEF_SYMBOL_OF_SYMBOL(sym), t);
433     return __symbol_resolve(sym, t);
434 }
435
436 static inline Symbol_Type_Struct *symbol_type_struct_alloc_finalize(
437     Symbol_Type_Struct *s, Symbol_Type_Struct_Methods *methods,
438     Symbol_Table *t)
439 {
440     s->methods = methods;
441     double_list_append(&t->all_symbol_types, &s->dbnode);
442     return s;
443 }
444
445 static inline void symbol_type_struct_destroy_common(
446     Symbol_Type_Struct *s UNUSED)
447 {
448 }
449
450 static bool symbol_rec_ambiguous_ref_compare(String search,
451     Hash_Map_Slot *map_slot)
452 {
453     Symbol_Type_Rec_Ambiguous_Ref *map_ref =
454         SYMBOL_REC_AMBIGUOUS_REF_OF(map_slot);
455     return !string_compare(map_ref->field_name, search);
456 }
457
458 static inline Symbol_Type_Rec_Ambiguous_Ref *symbol_rec_ambiguous_ref_alloc(
459     Const_String field_name)
460 {
461     Symbol_Type_Rec_Ambiguous_Ref *r =
462         ALLOC_NEW(Symbol_Type_Rec_Ambiguous_Ref);
463     r->sym_locations = VECTOR_INIT_SIZE(PTR_SIZE);
464     r->field_name = field_name;
465     return r;

```

```

466 }
467
468 static inline void symbol_rec_ambiguous_ref_add_loc(
469     Symbol_Type_Rec_Ambiguous_Ref *r, File_Location *loc)
470 {
471     Vector *v = &r->sym_locations;
472     File_Location *tmp_loc;
473     VECTOR_FOR_EACH_ENTRY(v, tmp_loc) {
474         if (!file_location_cmp(loc, tmp_loc))
475             return;
476     }
477     vector_append(v, loc);
478 }
479
480 static void symbol_rec_ambiguous_ref_destroy(Hash_Map_Slot *slot)
481 {
482     Symbol_Type_Rec_Ambiguous_Ref *r = SYMBOL_REC_AMBIGUOUS_REF_OF(slot);
483     vector_clear(&r->sym_locations);
484     free_mem(r);
485 }
486
487 /*****
488
489 static Symbol_Type symbol_type_void_get_type()
490 {
491     return SYMBOL_TYPE_VOID;
492 }
493
494 static Symbol_Type_Struct *symbol_type_void_resolve(Symbol_Type_Struct *self,
495     Symbol_Table *t UNUSED)
496 {
497     return self;
498 }
499
500 static bool symbol_type_void_same_type(Symbol_Type_Struct *self UNUSED,
501     Symbol_Type_Struct *oth)
502 {
503     Symbol_Type oth_t = oth->methods->get_type();
504     assert(oth_t != SYMBOL_TYPE_IDEN);
505     return oth_t == SYMBOL_TYPE_VOID || oth_t == SYMBOL_TYPE_UNKNOWN;
506 }
507
508 static void symbol_type_void_append_str(Symbol_Type_Struct *self UNUSED,
509     String_Builder *sb)
510 {
511     string_builder_append(sb, S("void"));
512 }
513
514 static void symbol_type_void_destroy(Symbol_Type_Struct *self)
515 {
516     symbol_type_struct_destroy_common(self);
517     free_mem(SYMBOL_TYPE_STRUCT_CONTAINER(self, Symbol_Type_Void));
518 }
519
520 static Symbol_Type_Struct_Methods symbol_type_void_methods = {
521     .get_type = symbol_type_void_get_type,
522     .resolve = symbol_type_void_resolve,
523     .append_str = symbol_type_void_append_str,
524     .destroy = symbol_type_void_destroy,
525     .same_type = symbol_type_void_same_type
526 };
527
528 Symbol_Type_Struct *symbol_type_void_alloc(Symbol_Table *t)
529 {
530     if (!t->symbol_type_struct_void) {
531         Symbol_Type_Void *s = ALLOC_NEW(Symbol_Type_Void);
532         t->symbol_type_struct_void = symbol_type_struct_alloc_finalize(
533             SYMBOL_TYPE_STRUCT_OF_CONTAINER(s),
534             &symbol_type_void_methods, t);
535     }
536     return t->symbol_type_struct_void;
537 }
538
539 *****/

```

```

540
541 /*****
542
543 static Symbol_Type symbol_type_int_get_type()
544 {
545     return SYMBOL_TYPE_INT;
546 }
547
548 static Symbol_Type_Struct *symbol_type_int_resolve(Symbol_Type_Struct *self,
549     Symbol_Table *t UNUSED)
550 {
551     return self;
552 }
553
554 static bool symbol_type_int_same_type(Symbol_Type_Struct *self UNUSED,
555     Symbol_Type_Struct *oth)
556 {
557     Symbol_Type oth_t = oth->methods->get_type();
558     assert(oth_t != SYMBOL_TYPE_IDEN);
559     return oth_t == SYMBOL_TYPE_INT || oth_t == SYMBOL_TYPE_UNKNOWN;
560 }
561
562 static void symbol_type_int_append_str(Symbol_Type_Struct *self UNUSED,
563     String_Builder *sb)
564 {
565     string_builder_append(sb, S("int"));
566 }
567
568 static void symbol_type_int_destroy(Symbol_Type_Struct *self)
569 {
570     symbol_type_struct_destroy_common(self);
571     free_mem(SYMBOL_TYPE_STRUCT_CONTAINER(self, Symbol_Type_Int));
572 }
573
574 static Symbol_Type_Struct_Methods symbol_type_int_methods = {
575     .get_type = symbol_type_int_get_type,
576     .resolve = symbol_type_int_resolve,
577     .append_str = symbol_type_int_append_str,
578     .destroy = symbol_type_int_destroy,
579     .same_type = symbol_type_int_same_type
580 };
581
582 Symbol_Type_Struct *symbol_type_int_alloc(Symbol_Table *t)
583 {
584     if (!t->symbol_type_struct_int) {
585         Symbol_Type_Int *s = ALLOC_NEW(Symbol_Type_Int);
586         t->symbol_type_struct_int = symbol_type_struct_alloc_finalize(
587             SYMBOL_TYPE_STRUCT_OF_CONTAINER(s),
588             &symbol_type_int_methods, t);
589     }
590     return t->symbol_type_struct_int;
591 }
592
593 /*****
594
595 static Symbol_Type symbol_type_iden_get_type()
596 {
597     return SYMBOL_TYPE_IDEN;
598 }
599
600 static Symbol_Type_Struct *symbol_type_iden_resolve(Symbol_Type_Struct *self,
601     Symbol_Table *t)
602 {
603     Symbol *saved_sym;
604     Symbol_Type_Struct *ret;
605     Symbol_Type_Iden *tiden = SYMBOL_TYPE_STRUCT_CONTAINER(self,
606         Symbol_Type_Iden);
607
608     Symbol *lookup = __symbol_table_node_lookup(tiden->sym_node, tiden->iden,
609         string_hash_code(tiden->iden), SYMBOL_PROPERTY_TYPE_DEF);
610     if (lookup) {
611         saved_sym = t->current_symbol;
612         t->current_symbol = lookup;
613         ret = symbol_resolve(lookup, t);

```

```

614         t->current_symbol = saved_sym;
615     } else {
616
617     #if 0
618         if (import_handler_is_merge_table(t))
619             return self;
620     #endif
621
622         ret = symbol_type_unknown_alloc(t);
623         if (!is_error_reported_here(tiden->loc)) {
624             report_error_location(tiden->loc, S("unable to resolve type "
625                 QFY("%S") "\n"),
626                 STRING_AFTER_DOT(tiden->iden));
627         }
628     }
629
630     return ret;
631 }
632
633 static bool symbol_type_iden_same_type(Symbol_Type_Struct *self UNUSED,
634     Symbol_Type_Struct *oth UNUSED)
635 {
636     DEBUGT(def,
637         fatal_error(S("Symbol table trying to compare identifier "
638             "type with other type\n"));
639     );
640     return false;
641 }
642
643 static void symbol_type_iden_append_str(Symbol_Type_Struct *self,
644     String_Builder *sb)
645 {
646     Symbol_Type_Iden *tiden =
647         SYMBOL_TYPE_STRUCT_CONTAINER(self, Symbol_Type_Iden);
648     String s = string_from_format(S("identifier (%S)", tiden->iden);
649     string_builder_append(sb, s);
650     string_destroy(s);
651 }
652
653 static void symbol_type_iden_destroy(Symbol_Type_Struct *self)
654 {
655     symbol_type_struct_destroy_common(self);
656     Symbol_Type_Iden *tiden = SYMBOL_TYPE_STRUCT_CONTAINER(self,
657         Symbol_Type_Iden);
658     string_destroy(tiden->iden);
659     free_mem(tiden);
660 }
661
662 static Symbol_Type_Struct_Methods symbol_type_iden_methods = {
663     .get_type = symbol_type_iden_get_type,
664     .resolve = symbol_type_iden_resolve,
665     .append_str = symbol_type_iden_append_str,
666     .destroy = symbol_type_iden_destroy,
667     .same_type = symbol_type_iden_same_type
668 };
669
670 Symbol_Type_Struct *symbol_type_iden_alloc(Symbol_Table *t, String iden)
671 {
672     Symbol_Type_Iden *s = ALLOC_NEW(Symbol_Type_Iden);
673     s->iden = string_duplicate(iden);
674     return symbol_type_struct_alloc_finalize(
675         SYMBOL_TYPE_STRUCT_OF_CONTAINER(s),
676         &symbol_type_iden_methods, t);
677 }
678
679 void symbol_type_iden_set_iden(Symbol_Type_Iden *self, String str)
680 {
681     string_destroy(self->iden);
682     self->iden = str;
683 }
684
685 /*****
686
687 static Symbol_Type symbol_type_rec_get_type()

```



```

688 {
689     return SYMBOL_TYPE_REC;
690 }
691
692 static inline Symbol_Type_Struct *symbol_rec_get_extended_type(
693     Symbol_Type_Struct *extended_type, Symbol_Table *t)
694 {
695     Type_Def_Symbol *sym;
696     Symbol_Type_Cycle *cycle;
697     if (t->last_resolve_pass || t->current_resolve_property !=
698         SYMBOL_PROPERTY_TYPE_DEF) {
699         switch (extended_type->methods->get_type()) {
700             case SYMBOL_TYPE_REC:
701                 break;
702             case SYMBOL_TYPE_CYCLE:
703                 cycle = SYMBOL_TYPE_STRUCT_CONTAINER(extended_type,
704                     Symbol_Type_Cycle);
705                 sym = symbol_table_node_lookup_type_def(cycle->sym_node,
706                     cycle->name, string_hash_code(cycle->name));
707                 assert(sym);
708                 if (sym->symbol.resolved_type->methods->get_type() ==
709                     SYMBOL_TYPE_REC) {
710                     extended_type = sym->symbol.resolved_type;
711                     break;
712                 }
713                 /* Fall through. */
714             case SYMBOL_TYPE_UNKNOWN:
715                 break;
716             default:
717                 if (!is_error_reported_here(t->current_symbol->location))
718                     report_error_location(t->current_symbol->location,
719                         S("record " QFY("%S") " is extending "
720                             "non-record type\n"),
721                         t->current_symbol->identifier);
722                 extended_type = symbol_type_unknown_alloc(t);
723                 break;
724         }
725     }
726     return extended_type;
727 }
728
729 static inline const char *get_current_record_type(Symbol_Table *t)
730 {
731     if (t->current_symbol->resolved_type->methods->get_type() ==
732         SYMBOL_TYPE_ARY)
733         return "array record";
734     return "record";
735 }
736
737 static inline void symbol_rec_append_remaps(Hash_Map *src,
738     Hash_Map *dest)
739 {
740     Hash_Map_Slot *src_slot;
741     HASH_MAP_FOR_EACH(src, src_slot) {
742         Symbol_Func_Remap *tmp = SYMBOL_FUNC_REMAP_OF(src_slot);
743         Uns hash = hash_map_slot_get_hash_code(&tmp->hash_slot);
744
745         if (!hash_map_contains(dest, tmp->initial_iden, hash)) {
746             Symbol_Func_Remap *new_map = symbol_func_remap_alloc(
747                 tmp->initial_iden, tmp->new_iden);
748             hash_map_insert(dest, &new_map->hash_slot, hash);
749         }
750     }
751 }
752
753 static inline void symbol_rec_override_map(
754     Symbol_Table_Node *tmp_node,
755     Symbol_Func_Map *curr_idens,
756     Symbol *super_sym,
757     Symbol_Table_Node *curr_node)
758 {
759     Symbol *curr_sym;
760     Vector *curr_overloaded = &curr_idens->overload_idens;
761

```

```

762     DLOG("Vector is: %p\n", curr_overloaded);
763
764     VECTOR_FOR_EACH_ENTRY(curr_overloaded, curr_sym) {
765         if (curr_sym->resolved_type->methods->same_type(
766             curr_sym->resolved_type,
767             super_sym->resolved_type)) {
768             Uns hash = string_hash_code(super_sym->identifier);
769
770             DLOG("insert overloaded: %S to symbol %p\n",
771                 super_sym->identifier, curr_sym);
772
773             if (!hash_map_contains(&curr_node->func_remaps,
774                 super_sym->identifier, hash)) {
775                 Symbol_Func_Remap *remap = symbol_func_remap_alloc(
776                     super_sym->identifier, curr_sym->identifier);
777                 hash_map_insert(&curr_node->func_remaps,
778                     &remap->hash_slot, hash);
779             }
780             return;
781         }
782     }
783
784     symbol_table_node_insert(
785         tmp_node,
786         curr_idens->func_iden,
787         super_sym->unique_name,
788         super_sym->resolved_type,
789         SYMBOL_PROPERTY_FUNC,
790         super_sym->location);
791 }
792
793 static inline void symbol_rec_accumulate_super_funcs(
794     Symbol_Table_Node *tmp_node,
795     Symbol_Type_Rec *curr_rec,
796     Symbol_Type_Rec *curr_super,
797     Symbol_Table *t UNUSED)
798 {
799     Hash_Map *super_map;
800     Hash_Map *curr_map;
801     Hash_Map_Slot *super_slot;
802     Hash_Map_Slot *curr_slot;
803     Symbol *super_sym;
804
805     super_map = &curr_super->rec_sym_node->func_iden_map;
806     curr_map = &curr_rec->rec_sym_node->func_iden_map;
807
808     HASH_MAP_FOR_EACH(super_map, super_slot) {
809         Symbol_Func_Map *super_idens = SYMBOL_FUNC_MAP_OF(super_slot);
810
811         if (symbol_func_maps_ctor(super_idens))
812             continue;
813
814         DLOG("super idens: %S\n", super_idens->func_iden);
815         curr_slot = hash_map_get(curr_map, super_idens->func_iden,
816             hash_map_slot_get_hash_code(&super_idens->hash_slot));
817         if (curr_slot) {
818             Symbol_Func_Map *curr_idens = SYMBOL_FUNC_MAP_OF(curr_slot);
819             DLOG("super size: %U\n", vector_size(&super_idens->overload_idens));
820             VECTOR_FOR_EACH_ENTRY(&super_idens->overload_idens, super_sym) {
821                 DLOG("super sym: %S\n", super_sym->identifier);
822                 symbol_rec_override_map(tmp_node, curr_idens,
823                     super_sym, curr_rec->rec_sym_node);
824             }
825         } else {
826             VECTOR_FOR_EACH_ENTRY(&super_idens->overload_idens, super_sym) {
827                 symbol_table_node_insert(
828                     tmp_node,
829                     super_idens->func_iden,
830                     super_sym->unique_name,
831                     super_sym->resolved_type,
832                     SYMBOL_PROPERTY_FUNC,
833                     super_sym->location);
834             }
835         }
836     }

```

```

836     symbol_rec_append_remaps (&super_idens->sym_node->func_remaps,
837                             &curr_rec->rec_sym_node->func_remaps);
838 }
839 }
840
841 static inline void symbol_rec_accumulate_super_vars(
842     Symbol_Table_Node *tmp_node,
843     Symbol_Type_Rec *curr_rec,
844     Symbol_Type_Rec *curr_super,
845     Symbol_Table *t)
846 {
847     Hash_Map *super_map;
848     Hash_Map_Slot *super_slot;
849     Symbol *base_sym, *tmp_sym, *super_sym;
850
851     super_map = &curr_super->rec_sym_node->symbol_maps[SYMBOL_PROPERTY_VAR];
852     HASH_MAP_FOR_EACH(super_map, super_slot) {
853         super_sym = SYMBOL_OF_SLOT(super_slot);
854         tmp_sym = symbol_table_node_get(tmp_node,
855                                         super_sym->identifier,
856                                         hash_map_slot_get_hash_code(&super_sym->hash_slot),
857                                         SYMBOL_PROPERTY_VAR);
858
859         if (!tmp_sym) {
860             symbol_table_node_insert(
861                 tmp_node,
862                 super_sym->identifier,
863                 super_sym->unique_name,
864                 super_sym->resolved_type,
865                 SYMBOL_PROPERTY_VAR,
866                 super_sym->location);
867         } else {
868             base_sym = symbol_table_node_get(
869                 curr_rec->rec_sym_node,
870                 super_sym->identifier,
871                 hash_map_slot_get_hash_code(&super_sym->hash_slot),
872                 SYMBOL_PROPERTY_VAR);
873
874             if (!base_sym) {
875                 Symbol_Type_Rec_Ambiguous_Ref *r =
876                     symbol_type_rec_get_ambiguous_ref(curr_rec,
877                                                         super_sym->identifier);
878
879                 if (r) {
880                     symbol_rec_ambiguous_ref_add_loc(r, super_sym->location);
881                 } else {
882                     r = symbol_rec_ambiguous_ref_alloc(super_sym->identifier);
883                     symbol_rec_ambiguous_ref_add_loc(r, tmp_sym->location);
884                     symbol_rec_ambiguous_ref_add_loc(r, super_sym->location);
885
886                     symbol_type_rec_add_ambiguous_ref(curr_rec, r);
887
888                     symbol_table_node_insert(
889                         curr_rec->rec_sym_node,
890                         super_sym->identifier,
891                         super_sym->unique_name,
892                         symbol_type_unknown_alloc(t),
893                         SYMBOL_PROPERTY_VAR,
894                         t->current_symbol->location);
895                 }
896             }
897         }
898     }
899 }
900
901 static inline void symbol_type_rec_append_super_vars(Symbol_Type_Rec *rec,
902                                                     Symbol_Table_Node *tmp_node)
903 {
904     Hash_Map *map;
905     Hash_Map_Slot *slot;
906     Symbol *sym;
907     Symbol_Table_Node *rec_node;
908
909     rec_node = rec->rec_sym_node;

```

```

910     map = &tmp_node->symbol_maps[SYMBOL_PROPERTY_VAR];
911     HASH_MAP_FOR_EACH(map, slot) {
912         sym = SYMBOL_OF_SLOT(slot);
913         if (!symbol_table_node_get(rec_node, sym->identifier,
914             hash_map_slot_get_hash_code(&sym->hash_slot),
915             SYMBOL_PROPERTY_VAR)) {
916             symbol_table_node_insert(rec_node,
917                 sym->identifier,
918                 sym->unique_name,
919                 sym->resolved_type,
920                 SYMBOL_PROPERTY_VAR,
921                 sym->location);
922         }
923     }
924 }
925 }
926
927 static inline void symbol_type_rec_append_super_funcs(Symbol_Type_Rec *rec,
928     Symbol_Table_Node *tmp_node)
929 {
930     Hash_Map_Slot *slot;
931     Symbol_Table_Node *rec_node = rec->rec_sym_node;
932     Hash_Map *map = &tmp_node->func_iden_map;
933
934     HASH_MAP_FOR_EACH(map, slot) {
935         Symbol_Func_Map *func_map = SYMBOL_FUNC_MAP_OF(slot);
936         Symbol *fsym;
937         VECTOR_FOR_EACH_ENTRY(&func_map->overload_idens, fsym) {
938             symbol_table_node_insert(rec_node,
939                 func_map->func_iden,
940                 fsym->identifier,
941                 fsym->resolved_type,
942                 SYMBOL_PROPERTY_FUNC,
943                 fsym->location);
944         }
945     }
946 }
947
948 bool __symbol_type_rec_ambiguous_cast(Symbol_Type_Rec *cast,
949     Symbol_Type_Rec *rec)
950 {
951     Vector *extend_vec;
952     Symbol_Type_Struct *super_type;
953     Uns count = 0;
954
955     extend_vec = &rec->extended_types;
956     VECTOR_FOR_EACH_ENTRY(extend_vec, super_type) {
957         if (symbol_type_rec_assignment_compatible(
958             SYMBOL_TYPE_STRUCT_OF_CONTAINER(cast), super_type)) {
959             if (++count > 1)
960                 return true;
961         }
962     }
963     return false;
964 }
965
966 #if 0
967 static bool symbol_type_rec_ambiguous_extend(Symbol_Type_Struct *cast_st,
968     Symbol_Type_Struct *rec_st)
969 {
970     Vector *extend_vec;
971     Symbol_Type_Struct *super_type;
972     Symbol_Type_Rec *super_rec;
973     Uns count = 0;
974
975     if (cast_st->methods->same_type(cast_st, rec_st))
976         return true;
977
978     Symbol_Type_Rec *cast = SYMBOL_TYPE_STRUCT_CONTAINER(cast_st,
979         Symbol_Type_Rec);
980     Symbol_Type_Rec *rec = SYMBOL_TYPE_STRUCT_CONTAINER(rec_st,
981         Symbol_Type_Rec);
982
983

```

```

984     extend_vec = &rec->extended_types;
985     VECTOR_FOR_EACH_ENTRY(extend_vec, super_type) {
986         super_rec = SYMBOL_TYPE_STRUCT_CONTAINER(super_type, Symbol_Type_Rec);
987         if (symbol_type_rec_assignment_compatible(
988             SYMBOL_TYPE_STRUCT_OF_CONTAINER(cast),
989             SYMBOL_TYPE_STRUCT_OF_CONTAINER(super_rec)) &&
990             symbol_type_rec_assignment_compatible(
991                 SYMBOL_TYPE_STRUCT_OF_CONTAINER(super_rec),
992                 SYMBOL_TYPE_STRUCT_OF_CONTAINER(cast))) {
993             if (++count > 1)
994                 return true;
995         }
996     }
997     return false;
998 }
999 #endif
1000
1001 static void symbol_table_node_destroy(Double_List_Node *dnnode);
1002
1003 static inline Symbol_Type_Struct *symbol_type_finalize(Symbol_Type_Struct *s,
1004     Symbol_Table *t);
1005
1006 static void symbol_table_node_merge_remap(Symbol_Table_Node *src,
1007     Symbol_Table_Node *dest);
1008
1009 static void report_missing_finalize(File_Location *loc, const char *curr_type,
1010     Const_String curr_rec, Const_String base_reg_name)
1011 {
1012     if (!cmdopts.warn_no_finalize)
1013         return;
1014
1015     if (base_reg_name) {
1016         report_warning_location(loc, S("record " QFY("%S")
1017             " inherited by %s " QFY("%S")
1018             " does not have a " QFY("finalize") " function\n"),
1019             STRING_AFTER_DOT(base_reg_name),
1020             curr_type,
1021             STRING_AFTER_DOT(curr_rec));
1022     } else {
1023         report_warning_location(loc, S("record inherited by %s " QFY("%S")
1024             " is not defining " QFY("finalize") " function\n"),
1025             curr_type,
1026             STRING_AFTER_DOT(curr_rec));
1027     }
1028 }
1029
1030 static void symbol_type_rec_insert_void_func(Symbol_Type_Rec *rec,
1031     Const_String func_name, String unique_name,
1032     Symbol_Table *t, bool is_concrete)
1033 {
1034     Symbol_Type_Struct *fstruct = symbol_type_func_alloc(t, false);
1035     Symbol_Type_Func *func = SYMBOL_TYPE_STRUCT_CONTAINER(fstruct,
1036         Symbol_Type_Func);
1037     func->return_type = symbol_type_void_alloc(t);
1038     func->body_sym_node = symbol_table_node_alloc_insert(rec->rec_sym_node,
1039         t, SYMBOL_TABLE_NODE_FUNC, NULL);
1040     func->is_resolved = true;
1041     func->is_concrete_func = is_concrete;
1042
1043     symbol_table_node_insert(rec->rec_sym_node, (String)func_name,
1044         unique_name, fstruct, SYMBOL_PROPERTY_FUNC,
1045         &t->>null_location);
1046
1047     vector_append(&rec->func_identifiers, unique_name);
1048     vector_append(&rec->func_types, fstruct);
1049 }
1050
1051 static void symbol_type_rec_insert_finalize(Symbol_Type_Rec *rec,
1052     Symbol_Table *t)
1053 {
1054     String del_name = string_from_format(S("%S.finalize"),
1055         rec->unique_name);
1056     rec->missing_finalize_name = del_name;
1057 }

```

```

1058
1059     symbol_type_rec_insert_void_func(rec, S("finalize"), del_name, t, true);
1060
1061     rec->rec_sym_node->has_finalize_func = true;
1062 }
1063
1064 static void symbol_type_rec_insert_record_func(Symbol_Type_Rec *rec,
1065     Symbol_Table *t, bool is_concrete)
1066 {
1067     String name = string_from_format(S("%S.record"), rec->unique_name);
1068     if (is_concrete)
1069         rec->missing_record_func_name = name;
1070
1071     symbol_type_rec_insert_void_func(rec, S("record"), name, t, is_concrete);
1072     rec->rec_sym_node->has_record_func = is_concrete;
1073 }
1074
1075 static bool symbol_type_rec_has_finalize(Symbol_Type_Rec *rec, Symbol_Table *t)
1076 {
1077     Vector *extend_vec;
1078     Symbol_Type_Struct *super_type;
1079     Symbol_Type_Rec *super_rec;
1080
1081     bool has_finalize = rec->rec_sym_node->has_finalize_func;
1082     if (!has_finalize) {
1083         extend_vec = &rec->extended_types;
1084         VECTOR_FOR_EACH_ENTRY(extend_vec, super_type) {
1085             super_rec = SYMBOL_TYPE_STRUCT_CONTAINER(super_type,
1086                 Symbol_Type_Rec);
1087             if (symbol_type_rec_has_finalize(super_rec, t)) {
1088                 /* Then we insert a default finalize function. */
1089                 has_finalize = true;
1090                 symbol_type_rec_insert_finalize(rec, t);
1091                 break;
1092             }
1093         }
1094     }
1095
1096     return has_finalize;
1097 }
1098
1099 static bool symbol_type_rec_has_record_func(Symbol_Type_Rec *rec,
1100     Symbol_Table *t)
1101 {
1102     Vector *extend_vec;
1103     Symbol_Type_Struct *super_type;
1104     Symbol_Type_Rec *super_rec;
1105
1106     bool has = rec->rec_sym_node->has_record_func;
1107     if (!has) {
1108         extend_vec = &rec->extended_types;
1109         VECTOR_FOR_EACH_ENTRY(extend_vec, super_type) {
1110             super_rec = SYMBOL_TYPE_STRUCT_CONTAINER(super_type,
1111                 Symbol_Type_Rec);
1112             if (symbol_type_rec_has_record_func(super_rec, t)) {
1113                 /* Then we insert a default 'record' function. */
1114                 has = true;
1115                 symbol_type_rec_insert_record_func(rec, t, true);
1116                 break;
1117             }
1118         }
1119     }
1120
1121     return has;
1122 }
1123
1124 static Symbol_Type_Struct *symbol_type_rec_insert_base_types(
1125     Symbol_Type_Rec *rec, Symbol_Table *t)
1126 {
1127     if (rec->super_fields_appended)
1128         return SYMBOL_TYPE_STRUCT_OF_CONTAINER(rec);
1129
1130     rec->super_fields_appended = true;
1131

```

```

1132     for (Uns i = 0; i < vector_size(&rec->extended_types); i++) {
1133         Symbol_Type_Struct *base_struct = vector_get(&rec->extended_types, i);
1134         base_struct = symbol_type_finalize(base_struct, t);
1135         vector_set(&rec->extended_types, i, base_struct);
1136     }
1137     for (Uns i = 0; i < vector_size(&rec->func_types); i++) {
1138         Symbol_Type_Struct *func_struct = vector_get(&rec->func_types, i);
1139         func_struct = symbol_type_finalize(func_struct, t);
1140         vector_set(&rec->func_types, i, func_struct);
1141     }
1142     for (Uns i = 0; i < vector_size(&rec->var_types); i++) {
1143         Symbol_Type_Struct *var_struct = vector_get(&rec->var_types, i);
1144         var_struct = symbol_type_finalize(var_struct, t);
1145         vector_set(&rec->var_types, i, var_struct);
1146     }
1147
1148     Vector *extend_vec;
1149     Symbol_Type_Struct *super_type;
1150     Symbol_Type_Rec *super_rec;
1151
1152     Vector parsed_recs = VECTOR_INIT_SIZE(4);
1153     Symbol_Type_Struct *ret = SYMBOL_TYPE_STRUCT_OF_CONTAINER(rec);
1154
1155     Symbol_Table_Node *tmp_node = __symbol_table_node_alloc(t,
1156         SYMBOL_TABLE_NODE_REC, rec);
1157
1158     /* Insert default 'finalize' function if it's needed. */
1159     symbol_type_rec_has_finalize(rec, t);
1160
1161     /* Insert default function 'record' if it's missing. */
1162     symbol_type_rec_has_record_func(rec, t);
1163
1164     extend_vec = &rec->extended_types;
1165     VECTOR_FOR_EACH_ENTRY(extend_vec, super_type) {
1166         super_rec = SYMBOL_TYPE_STRUCT_OF_CONTAINER(super_type, Symbol_Type_Rec);
1167
1168         bool super_has_finalize = symbol_type_rec_has_finalize(super_rec, t);
1169         if (symbol_type_rec_ambiguous_cast(super_type,
1170             SYMBOL_TYPE_STRUCT_OF_CONTAINER(rec))) {
1171
1172             bool exists = false;
1173             Symbol_Type_Rec *parsed_rec;
1174             VECTOR_FOR_EACH_ENTRY(&parsed_recs, parsed_rec) {
1175                 if (parsed_rec == super_rec) {
1176                     exists = true;
1177                     break;
1178                 }
1179             }
1180
1181             if (!exists) {
1182                 const char *curr_type = get_current_record_type(t);
1183                 bool is_merge_table = import_handler_is_merge_table(t);
1184                 if (super_rec->rec_name && !is_merge_table) {
1185                     report_error_location(t->current_symbol->location,
1186                         S("%s " QFY("%S")
1187                             " is extending record " QFY("%S")
1188                             " which is inaccessible due to ambiguity\n"),
1189                         curr_type,
1190                         STRING_AFTER_DOT(t->current_symbol->identifier),
1191                         STRING_AFTER_DOT(super_rec->rec_name));
1192                 } else if (!is_merge_table) {
1193                     report_error_location(t->current_symbol->location,
1194                         S("%s " QFY("%S")
1195                             " is extending a record "
1196                             " which is inaccessible due to ambiguity\n"),
1197                         curr_type,
1198                         STRING_AFTER_DOT(t->current_symbol->identifier));
1199                 }
1200             }
1201         } else if (!super_has_finalize) {
1202             const char *curr_type = get_current_record_type(t);
1203             report_missing_finalize(t->current_symbol->location, curr_type,
1204                 t->current_symbol->identifier, super_rec->rec_name);
1205         }

```

```

1206
1207     if (symbol_type_rec_assignment_compatible(ret,
1208         SYMBOL_TYPE_STRUCT_OF_CONTAINER(super_rec)) {
1209         /* This actually should never happen. */
1210         const char *curr_type = get_current_record_type(t);
1211         Const_String msg;
1212         msg = S("%s " QFY("%S") " is ambiguously extending the "
1213             "same record twice\n");
1214         report_error_location(t->current_symbol->location,
1215             msg, curr_type,
1216             STRING_AFTER_DOT(t->current_symbol->identifier));
1217         ret = symbol_type_unknown_alloc(t);
1218         goto out;
1219     }
1220
1221     symbol_rec_accumulate_super_vars(tmp_node, rec, super_rec, t);
1222     symbol_rec_accumulate_super_funcs(tmp_node, rec, super_rec, t);
1223
1224     vector_append(&parsed_recs, super_type);
1225 }
1226
1227 symbol_type_rec_append_super_vars(rec, tmp_node);
1228 symbol_type_rec_append_super_funcs(rec, tmp_node);
1229
1230 out:
1231 vector_clear(&parsed_recs);
1232 symbol_table_node_destroy(&tmp_node->dbnode);
1233
1234 Symbol_Table_Node *rec_node = rec->rec_sym_node;
1235 Symbol_Func_Map *fmap = symbol_table_node_get_func_map(rec_node,
1236     (String)S("record"), string_hash_code(S("record")));
1237 if (!fmap)
1238     symbol_type_rec_insert_record_func(rec, t, false);
1239
1240 return ret;
1241 }
1242
1243 static Symbol_Type_Struct *symbol_type_func_finalize(Symbol_Type_Func *f,
1244     Symbol_Table *t);
1245
1246 static inline Symbol_Type_Struct *symbol_type_finalize(Symbol_Type_Struct *s,
1247     Symbol_Table *t)
1248 {
1249     Symbol_Type_Ary *ary;
1250     Symbol_Type_Rec *rec;
1251     Symbol_Type_Func *func;
1252     Symbol_Type_Cycle *cycle;
1253     switch (s->methods->get_type()) {
1254     case SYMBOL_TYPE_REC:
1255         rec = SYMBOL_TYPE_STRUCT_CONTAINER(s, Symbol_Type_Rec);
1256         return symbol_type_rec_insert_base_types(rec, t);
1257
1258     case SYMBOL_TYPE_ARY:
1259         ary = SYMBOL_TYPE_STRUCT_CONTAINER(s, Symbol_Type_Ary);
1260         ary->ary_type = symbol_type_finalize(ary->ary_type, t);
1261         return s;
1262
1263     case SYMBOL_TYPE_FUNC:
1264         func = SYMBOL_TYPE_STRUCT_CONTAINER(s, Symbol_Type_Func);
1265         return symbol_type_func_finalize(func, t);
1266
1267     case SYMBOL_TYPE_CYCLE:
1268         cycle = SYMBOL_TYPE_STRUCT_CONTAINER(s, Symbol_Type_Cycle);
1269         return symbol_table_node_lookup(cycle->sym_node, cycle->name,
1270             SYMBOL_PROPERTY_TYPE_DEF)->resolved_type;
1271
1272     default:
1273         return s;
1274     }
1275 }
1276
1277 static Symbol_Type_Struct *symbol_type_func_finalize(Symbol_Type_Func *f,
1278     Symbol_Table *t)
1279 {

```



```

1280     f->return_type = symbol_type_finalize(f->return_type, t);
1281     for (Uns i = 0; i < vector_size(&f->param_types); i++) {
1282         Symbol_Type_Struct *p = vector_get(&f->param_types, i);
1283         p = symbol_type_finalize(p, t);
1284         vector_set(&f->param_types, i, p);
1285     }
1286     return SYMBOL_TYPE_STRUCT_OF_CONTAINER(f);
1287 }
1288
1289 static Symbol_Type_Struct *symbol_type_rec_resolve(Symbol_Type_Struct *self,
1290     Symbol_Table *t)
1291 {
1292     Vector *tmp_vec;
1293     String tmp_str;
1294     Symbol *saved_sym, *sym;
1295     Uns i;
1296     Symbol_Type_Struct *tmp_type;
1297     Symbol_Type_Rec *trec = SYMBOL_TYPE_STRUCT_CONTAINER(self,
1298         Symbol_Type_Rec);
1299
1300     /* Save cycle type such that we can reset it before returning. */
1301     Symbol_Rec_Cycle_Type prev_table_cycle_type = t->rec_cycle_type;
1302     Symbol_Rec_Cycle_Type prev_rec_cycle_type = trec->cycle_mark;
1303
1304     /* Cycle is not allowed in the extended records. */
1305     trec->cycle_mark = t->rec_cycle_type = SYMBOL_REC_CYCLE_NOT_ALLOWED;
1306     i = 0;
1307     tmp_vec = &trec->extended_types;
1308     VECTOR_FOR_EACH_ENTRY(tmp_vec, tmp_type) {
1309         tmp_type = tmp_type->methods->resolve(tmp_type, t);
1310         DEBUGT(def,
1311             if (t->last_resolve_pass)
1312                 assert(tmp_type)
1313         );
1314         if (tmp_type) {
1315             tmp_type = symbol_rec_get_extended_type(tmp_type, t);
1316             if (tmp_type->methods->get_type() == SYMBOL_TYPE_UNKNOWN) {
1317                 self = symbol_type_unknown_alloc(t);
1318                 // goto out;
1319             }
1320             vector_set(tmp_vec, i++, tmp_type);
1321         } else {
1322             self = NULL;
1323             goto out;
1324         }
1325     }
1326
1327     if (self->methods->get_type() == SYMBOL_TYPE_UNKNOWN)
1328         goto out;
1329
1330     if (trec->body_resolved && !t->last_resolve_pass)
1331         goto out;
1332     trec->body_resolved = true;
1333
1334     if (trec->body_resolved_last)
1335         goto out;
1336
1337     if (t->last_resolve_pass)
1338         trec->body_resolved_last = true;
1339
1340     /* Cycle allowed in the functions and variables inside records. */
1341     trec->cycle_mark = t->rec_cycle_type = SYMBOL_REC_CYCLE_ALLOWED;
1342
1343     saved_sym = t->current_symbol;
1344     i = 0;
1345     tmp_vec = &trec->func_identifiers;
1346     VECTOR_FOR_EACH_ENTRY(tmp_vec, tmp_str) {
1347         DLOG("lookup func: %S\n", tmp_str);
1348         sym = symbol_table_node_get(trec->rec_sym_node,
1349             tmp_str, string_hash_code(tmp_str), SYMBOL_PROPERTY_FUNC);
1350         assert(sym);
1351         t->current_symbol = sym;
1352         tmp_type = sym->resolved_type->methods->resolve(
1353             sym->resolved_type, t);

```

```

1354     DEBUGT(def, if (trec->body_resolved_last) assert(tmp_type));
1355
1356     if (tmp_type)
1357         sym->resolved_type = tmp_type;
1358
1359     if (trec->body_resolved_last) {
1360         if (tmp_type->methods->get_type() == SYMBOL_TYPE_CYCLE) {
1361             Symbol_Type_Cycle *cycle = SYMBOL_TYPE_STRUCT_CONTAINER(
1362                 tmp_type, Symbol_Type_Cycle);
1363             sym->resolved_type = symbol_table_node_lookup(
1364                 cycle->sym_node, cycle->name,
1365                 SYMBOL_PROPERTY_TYPE_DEF->resolved_type;
1366             }
1367             vector_append(&trec->func_types, sym->resolved_type);
1368         }
1369     }
1370
1371     i = 0;
1372     tmp_vec = &trec->var_identifiers;
1373     VECTOR_FOR_EACH_ENTRY(tmp_vec, tmp_str) {
1374         DLOG("lookup var: %S\n", tmp_str);
1375         sym = symbol_table_node_get(trec->rec_sym_node,
1376             tmp_str, string_hash_code(tmp_str), SYMBOL_PROPERTY_VAR);
1377         assert(sym);
1378         t->current_symbol = sym;
1379         tmp_type = sym->resolved_type->methods->resolve(
1380             sym->resolved_type, t);
1381         DEBUGT(def, if (trec->body_resolved_last) assert(tmp_type));
1382         if (tmp_type)
1383             sym->resolved_type = tmp_type;
1384         if (trec->body_resolved_last) {
1385             if (tmp_type->methods->get_type() == SYMBOL_TYPE_CYCLE) {
1386                 Symbol_Type_Cycle *cycle = SYMBOL_TYPE_STRUCT_CONTAINER(
1387                     tmp_type, Symbol_Type_Cycle);
1388                 sym->resolved_type = symbol_table_node_lookup(
1389                     cycle->sym_node, cycle->name,
1390                     SYMBOL_PROPERTY_TYPE_DEF->resolved_type;
1391                 }
1392                 vector_append(&trec->var_types, sym->resolved_type);
1393             }
1394         }
1395
1396         t->current_symbol = saved_sym;
1397
1398     out:
1399     /* Restore the cycle type to whatever it was before this
1400      * function was called. */
1401     t->rec_cycle_type = prev_table_cycle_type;
1402     trec->cycle_mark = prev_rec_cycle_type;
1403     return self;
1404 }
1405
1406 static bool symbol_type_rec_same_type(Symbol_Type_Struct *self,
1407     Symbol_Type_Struct *oth)
1408 {
1409     Hash_Map *rec_comp_map;
1410     Symbol *oth_sym;
1411     Symbol_Type_Rec *oth_rec, *self_rec;
1412     Symbol_Type oth_t = oth->methods->get_type();
1413     bool ret = true;
1414
1415     assert(oth_t != SYMBOL_TYPE_IDEN);
1416
1417     if (oth == self || oth_t == SYMBOL_TYPE_UNKNOWN)
1418         goto out;
1419
1420     self_rec = SYMBOL_TYPE_STRUCT_CONTAINER(self, Symbol_Type_Rec);
1421
1422     if (oth_t == SYMBOL_TYPE_CYCLE) {
1423         Symbol_Type_Cycle *oth_cy =
1424             SYMBOL_TYPE_STRUCT_CONTAINER(oth, Symbol_Type_Cycle);
1425         oth_sym = symbol_table_node_lookup(oth_cy->sym_node,
1426             oth_cy->name, SYMBOL_PROPERTY_TYPE_DEF);
1427         ret = symbol_type_rec_same_type(self, oth_sym->resolved_type);

```

```

1428     goto out;
1429 }
1430
1431 if (oth_t != SYMBOL_TYPE_REC) {
1432     ret = false;
1433     goto out;
1434 }
1435 oth_rec = SYMBOL_TYPE_STRUCT_CONTAINER(oth, Symbol_Type_Rec);
1436
1437 rec_comp_map = self_rec->rec_sym_node->record_comparisons;
1438
1439 Symbol_Rec_Comp_Struct *comp = symbol_rec_comp_struct_alloc(
1440     self_rec, oth_rec);
1441 Symbol_Rec_Comp_Struct *map_comp = symbol_table_get_comp_result(
1442     rec_comp_map, comp);
1443
1444 if (map_comp) {
1445     symbol_rec_comp_struct_destroy(&comp->hash_slot);
1446     if (symbol_rec_comp_struct_get(map_comp) == REC_COMP_DIFFERENT)
1447         ret = false;
1448     else
1449         ret = true;
1450     goto out;
1451 }
1452 symbol_table_insert_comp_result(rec_comp_map, comp);
1453
1454 ret = symbol_type_rec_structs_equal(self_rec, oth_rec);
1455
1456 if (ret)
1457     symbol_rec_comp_struct_set(comp, REC_COMP_SAME);
1458 else
1459     symbol_rec_comp_struct_set(comp, REC_COMP_DIFFERENT);
1460
1461 out:
1462     return ret;
1463 }
1464
1465 static void symbol_type_rec_append_str(Symbol_Type_Struct *self,
1466     String_Builder *sb)
1467 {
1468     Uns vec_size, idx;
1469     Vector *tmp_vec;
1470     Symbol_Type_Struct *tmp_type;
1471     Symbol_Type_Rec *trec =
1472         SYMBOL_TYPE_STRUCT_CONTAINER(self, Symbol_Type_Rec);
1473
1474     string_builder_append(sb, S("record["));
1475     if (trec->rec_name)
1476         string_builder_append(sb, trec->rec_name);
1477
1478     if (trec->append_str_cycle_marked) {
1479         string_builder_append(sb, S("]"));
1480         goto out;
1481     }
1482
1483     trec->append_str_cycle_marked = true;
1484
1485     string_builder_append(sb, S("] "));
1486     string_builder_append(sb, S("of"));
1487
1488     tmp_vec = &trec->extended_types;
1489     vec_size = vector_size(tmp_vec);
1490     if (!vec_size)
1491         goto skip_extend;
1492
1493     string_builder_append(sb, S(" "));
1494     for (idx = 0; idx < vec_size - 1; idx++) {
1495         tmp_type = vector_get(tmp_vec, idx);
1496         tmp_type->methods->append_str(tmp_type, sb);
1497         string_builder_append(sb, S(", "));
1498     }
1499     tmp_type = vector_get(tmp_vec, idx);
1500     tmp_type->methods->append_str(tmp_type, sb);
1501

```

```

1502 skip_extend:
1503     string_builder_append(sb, S(" { "));
1504
1505     tmp_vec = &trec->func_types;
1506     vec_size = vector_size(tmp_vec);
1507     if (!vec_size)
1508         goto skip_funcs;
1509
1510     for (idx = 0; idx < vec_size - 1; idx++) {
1511         tmp_type = vector_get(tmp_vec, idx);
1512         tmp_type->methods->append_str(tmp_type, sb);
1513         string_builder_append(sb, S(", "));
1514     }
1515     tmp_type = vector_get(tmp_vec, idx);
1516     tmp_type->methods->append_str(tmp_type, sb);
1517
1518 skip_funcs:
1519     if (!vector_is_empty(tmp_vec))
1520         string_builder_append(sb, S(", "));
1521
1522     tmp_vec = &trec->var_types;
1523     vec_size = vector_size(tmp_vec);
1524     if (!vec_size)
1525         goto skip_vars;
1526
1527     for (idx = 0; idx < vec_size - 1; idx++) {
1528         tmp_type = vector_get(tmp_vec, idx);
1529         tmp_type->methods->append_str(tmp_type, sb);
1530         string_builder_append(sb, S(", "));
1531     }
1532     tmp_type = vector_get(tmp_vec, idx);
1533     tmp_type->methods->append_str(tmp_type, sb);
1534     string_builder_append(sb, S(" "));
1535
1536     trec->append_str_cycle_marked = false;
1537
1538 skip_vars:
1539     string_builder_append(sb, S("}"));
1540
1541 out:;
1542 }
1543
1544 void symbol_type_rec_append_func_identifier(Symbol_Type_Rec *self,
1545     Const_String iden)
1546 {
1547     DLOG("append: %S\n", iden);
1548     vector_append(&self->func_identifiers, string_duplicate(iden));
1549 }
1550
1551 void symbol_type_rec_append_var_identifier(Symbol_Type_Rec *self,
1552     Const_String iden)
1553 {
1554     vector_append(&self->var_identifiers, string_duplicate(iden));
1555 }
1556
1557 void symbol_type_rec_set_name(Symbol_Type_Rec *self, Const_String name)
1558 {
1559     self->rec_name = string_duplicate(name);
1560 }
1561
1562 void symbol_type_rec_set_unique_name(Symbol_Type_Rec *self,
1563     Const_String unique_name)
1564 {
1565     self->unique_name = string_duplicate(unique_name);
1566 }
1567
1568 static void symbol_type_rec_destroy(Symbol_Type_Struct *self)
1569 {
1570     Symbol_Type_Rec *trec = SYMBOL_TYPE_STRUCT_CONTAINER(self,
1571         Symbol_Type_Rec);
1572     symbol_type_struct_destroy_common(self);
1573
1574     if (trec->rec_name)
1575         string_destroy(trec->rec_name);

```

```

1576     vector_clear(&trec->extended_types);
1577     vector_for_each_destroy(&trec->func_identifiers,
1578         (Vector_Destructor) string_destroy);
1579     vector_clear(&trec->func_types);
1580     vector_for_each_destroy(&trec->var_identifiers,
1581         (Vector_Destructor) string_destroy);
1582     vector_clear(&trec->var_types);
1583
1584     string_destroy(trec->unique_name);
1585
1586     hash_map_for_each_destroy(&trec->ambiguous_refs,
1587         symbol_rec_ambiguous_ref_destroy);
1588
1589     free_mem(trec);
1590 }
1591
1592
1593 static Symbol_Type_Struct_Methods symbol_type_rec_methods = {
1594     .get_type = symbol_type_rec_get_type,
1595     .resolve = symbol_type_rec_resolve,
1596     .append_str = symbol_type_rec_append_str,
1597     .destroy = symbol_type_rec_destroy,
1598     .same_type = symbol_type_rec_same_type
1599 };
1600
1601 Symbol_Type_Struct *symbol_type_rec_alloc(Symbol_Table *t, Uns scope_id,
1602     Const_String unique_name)
1603 {
1604     Symbol_Type_Rec *s = ALLOC_NEW(Symbol_Type_Rec);
1605     s->extended_types = VECTOR_INIT_SIZE(PTR_SIZE);
1606     s->func_identifiers = VECTOR_INIT_SIZE(PTR_SIZE);
1607     s->func_types = VECTOR_INIT_SIZE(PTR_SIZE);
1608     s->var_identifiers = VECTOR_INIT_SIZE(PTR_SIZE);
1609     s->var_types = VECTOR_INIT_SIZE(PTR_SIZE);
1610     s->ambiguous_refs = HASH_MAP_INIT_SIZE(HASH_MAP_SIZE_5,
1611         (Hash_Map_Comparator) symbol_rec_ambiguous_ref_compare);
1612     s->rec_name = NULL;
1613     s->cycle_mark = SYMBOL_REC_CYCLE_MARK_NONE;
1614     s->unique_name = string_duplicate(unique_name);
1615     s->scope_id = scope_id;
1616     s->body_resolved = false;
1617     s->body_resolved_last = false;
1618     s->append_str_cycle_marked = false;
1619     s->super_fields_appended = false;
1620     s->imp_table_updated = false;
1621     s->imp_table_name_updated = false;
1622     s->missing_finalize_name = NULL;
1623     s->missing_record_func_name = NULL;
1624     return symbol_type_struct_alloc_finalize(
1625         SYMBOL_TYPE_STRUCT_OF_CONTAINER(s),
1626         &symbol_type_rec_methods, t);
1627 }
1628
1629 /*****
1630
1631 static Symbol_Type symbol_type_unknown_get_type()
1632 {
1633     return SYMBOL_TYPE_UNKNOWN;
1634 }
1635
1636 static bool symbol_type_unknown_same_type(Symbol_Type_Struct *self UNUSED,
1637     Symbol_Type_Struct *oth UNUSED)
1638 {
1639     return true;
1640 }
1641
1642 static Symbol_Type_Struct *symbol_type_unknown_resolve(
1643     Symbol_Type_Struct *self, Symbol_Table *t UNUSED)
1644 {
1645     return self;
1646 }
1647
1648 static void symbol_type_unknown_append_str(Symbol_Type_Struct *self UNUSED,
1649     String_Builder *sb)

```

```

1650 {
1651     string_builder_append(sb, S("unknown"));
1652 }
1653
1654 static void symbol_type_unknown_destroy(Symbol_Type_Struct *self)
1655 {
1656     Symbol_Type_Unknown *utype = SYMBOL_TYPE_STRUCT_CONTAINER(self,
1657         Symbol_Type_Unknown);
1658     symbol_type_struct_destroy_common(self);
1659     free_mem(utype);
1660 }
1661
1662 static Symbol_Type_Struct_Methods symbol_type_unknown_methods = {
1663     .get_type = symbol_type_unknown_get_type,
1664     .resolve = symbol_type_unknown_resolve,
1665     .append_str = symbol_type_unknown_append_str,
1666     .destroy = symbol_type_unknown_destroy,
1667     .same_type = symbol_type_unknown_same_type
1668 };
1669
1670 Symbol_Type_Struct *symbol_type_unknown_alloc(Symbol_Table *t)
1671 {
1672     if (!t->symbol_type_struct_unknown) {
1673         Symbol_Type_Unknown *s = ALLOC_NEW(Symbol_Type_Unknown);
1674         t->symbol_type_struct_unknown = symbol_type_struct_alloc_finalize(
1675             SYMBOL_TYPE_STRUCT_OF_CONTAINER(s),
1676             &symbol_type_unknown_methods, t);
1677     }
1678     return t->symbol_type_struct_unknown;
1679 }
1680
1681 /*****
1682
1683 static Symbol_Type symbol_type_bool_get_type()
1684 {
1685     return SYMBOL_TYPE_BOOL;
1686 }
1687
1688 static Symbol_Type_Struct *symbol_type_bool_resolve(
1689     Symbol_Type_Struct *self, Symbol_Table *t UNUSED)
1690 {
1691     return self;
1692 }
1693
1694 static bool symbol_type_bool_same_type(Symbol_Type_Struct *self UNUSED,
1695     Symbol_Type_Struct *oth)
1696 {
1697     Symbol_Type oth_t = oth->methods->get_type();
1698     return oth_t == SYMBOL_TYPE_BOOL || oth_t == SYMBOL_TYPE_UNKNOWN;
1699 }
1700
1701 static void symbol_type_bool_append_str(Symbol_Type_Struct *self UNUSED,
1702     String_Builder *sb)
1703 {
1704     string_builder_append(sb, S("bool"));
1705 }
1706
1707 static void symbol_type_bool_destroy(Symbol_Type_Struct *self)
1708 {
1709     Symbol_Type_Bool *t = SYMBOL_TYPE_STRUCT_CONTAINER(self,
1710         Symbol_Type_Bool);
1711     symbol_type_struct_destroy_common(self);
1712     free_mem(t);
1713 }
1714
1715 static Symbol_Type_Struct_Methods symbol_type_bool_methods = {
1716     .get_type = symbol_type_bool_get_type,
1717     .resolve = symbol_type_bool_resolve,
1718     .append_str = symbol_type_bool_append_str,
1719     .destroy = symbol_type_bool_destroy,
1720     .same_type = symbol_type_bool_same_type
1721 };
1722
1723 Symbol_Type_Struct *symbol_type_bool_alloc(Symbol_Table *t)

```

```

1724 {
1725     if (!t->symbol_type_struct_bool) {
1726         Symbol_Type_Bool *s = ALLOC_NEW(Symbol_Type_Bool);
1727         t->symbol_type_struct_bool = symbol_type_struct_alloc_finalize(
1728             SYMBOL_TYPE_STRUCT_OF_CONTAINER(s),
1729             &symbol_type_bool_methods, t);
1730     }
1731     return t->symbol_type_struct_bool;
1732 }
1733
1734 /*****
1735
1736 static Symbol_Type symbol_type_char_get_type()
1737 {
1738     return SYMBOL_TYPE_CHAR;
1739 }
1740
1741 static Symbol_Type_Struct *symbol_type_char_resolve(
1742     Symbol_Type_Struct *self, Symbol_Table *t UNUSED)
1743 {
1744     return self;
1745 }
1746
1747 static bool symbol_type_char_same_type(Symbol_Type_Struct *self UNUSED,
1748     Symbol_Type_Struct *oth)
1749 {
1750     Symbol_Type oth_t = oth->methods->get_type();
1751     return oth_t == SYMBOL_TYPE_CHAR || oth_t == SYMBOL_TYPE_UNKNOWN;
1752 }
1753
1754 static void symbol_type_char_append_str(Symbol_Type_Struct *self UNUSED,
1755     String_Builder *sb)
1756 {
1757     string_builder_append(sb, S("char"));
1758 }
1759
1760 static void symbol_type_char_destroy(Symbol_Type_Struct *self)
1761 {
1762     Symbol_Type_Char *t = SYMBOL_TYPE_STRUCT_CONTAINER(self,
1763         Symbol_Type_Char);
1764     symbol_type_struct_destroy_common(self);
1765     free_mem(t);
1766 }
1767
1768 static Symbol_Type_Struct_Methods symbol_type_char_methods = {
1769     .get_type = symbol_type_char_get_type,
1770     .resolve = symbol_type_char_resolve,
1771     .append_str = symbol_type_char_append_str,
1772     .destroy = symbol_type_char_destroy,
1773     .same_type = symbol_type_char_same_type
1774 };
1775
1776 Symbol_Type_Struct *symbol_type_char_alloc(Symbol_Table *t)
1777 {
1778     if (!t->symbol_type_struct_char) {
1779         Symbol_Type_Char *s = ALLOC_NEW(Symbol_Type_Char);
1780         t->symbol_type_struct_char = symbol_type_struct_alloc_finalize(
1781             SYMBOL_TYPE_STRUCT_OF_CONTAINER(s),
1782             &symbol_type_char_methods, t);
1783     }
1784     return t->symbol_type_struct_char;
1785 }
1786
1787 Symbol_Type_Rec *symbol_type_rec_assignment_compatible(
1788     Symbol_Type_Struct *rec_struct, Symbol_Type_Struct *oth_struct)
1789 {
1790     Symbol_Type_Rec *ret = NULL;
1791     Vector *extended_records;
1792
1793     if (rec_struct->methods->same_type(rec_struct, oth_struct)) {
1794         ret = SYMBOL_TYPE_STRUCT_CONTAINER(oth_struct, Symbol_Type_Rec);
1795         goto out;
1796     }
1797

```

```

1798     extended_records = &SYMBOL_TYPE_STRUCT_CONTAINER(oth_struct,
1799         Symbol_Type_Rec->extended_types;
1800     VECTOR_FOR_EACH_ENTRY(extended_records, oth_struct)
1801         if ((ret = symbol_type_rec_assignment_compatible(rec_struct,
1802             oth_struct)))
1803             break;
1804 out:
1805     return ret;
1806 }
1807
1808 /*****
1809
1810 static Symbol_Type symbol_type_string_get_type()
1811 {
1812     return SYMBOL_TYPE_STRING;
1813 }
1814
1815 static Symbol_Type_Struct *symbol_type_string_resolve(
1816     Symbol_Type_Struct *self, Symbol_Table *t UNUSED)
1817 {
1818     return self;
1819 }
1820
1821 static bool symbol_type_string_same_type(Symbol_Type_Struct *self UNUSED,
1822     Symbol_Type_Struct *oth)
1823 {
1824     Symbol_Type oth_t = oth->methods->get_type();
1825     return oth_t == SYMBOL_TYPE_STRING || oth_t == SYMBOL_TYPE_UNKNOWN;
1826 }
1827
1828 static void symbol_type_string_append_str(Symbol_Type_Struct *self UNUSED,
1829     String_Builder *sb)
1830 {
1831     string_builder_append(sb, S("string"));
1832 }
1833
1834 static void symbol_type_string_destroy(Symbol_Type_Struct *self)
1835 {
1836     Symbol_Type_String *t = SYMBOL_TYPE_STRUCT_CONTAINER(self,
1837         Symbol_Type_String);
1838     symbol_type_struct_destroy_common(self);
1839     free_mem(t);
1840 }
1841
1842 static Symbol_Type_Struct_Methods symbol_type_string_methods = {
1843     .get_type = symbol_type_string_get_type,
1844     .resolve = symbol_type_string_resolve,
1845     .append_str = symbol_type_string_append_str,
1846     .destroy = symbol_type_string_destroy,
1847     .same_type = symbol_type_string_same_type
1848 };
1849
1850 Symbol_Type_Struct *symbol_type_string_alloc(Symbol_Table *t)
1851 {
1852     if (!t->symbol_type_struct_string) {
1853         Symbol_Type_String *s = ALLOC_NEW(Symbol_Type_String);
1854         t->symbol_type_struct_string = symbol_type_struct_alloc_finalize(
1855             SYMBOL_TYPE_STRUCT_OF_CONTAINER(s),
1856             &symbol_type_string_methods, t);
1857     }
1858     return t->symbol_type_struct_string;
1859 }
1860
1861 /*****
1862
1863 static Symbol_Type symbol_type_ary_get_type()
1864 {
1865     return SYMBOL_TYPE_ARY;
1866 }
1867
1868 static Symbol_Type_Struct *symbol_type_ary_resolve(
1869     Symbol_Type_Struct *self, Symbol_Table *t)
1870 {
1871     Symbol_Type_Struct *tmp_t;

```



```

1872
1873     Symbol_Type_Ary *ary = SYMBOL_TYPE_STRUCT_CONTAINER(self, Symbol_Type_Ary);
1874
1875     tmp_t = ary->ary_type->methods->resolve(ary->ary_type, t);
1876     if (tmp_t) {
1877         ary->ary_type = tmp_t;
1878         if (tmp_t->methods->get_type() == SYMBOL_TYPE_UNKNOWN)
1879             self = symbol_type_unknown_alloc(t);
1880     } else {
1881         assert(!t->last_resolve_pass);
1882         self = NULL;
1883     }
1884
1885     return self;
1886 }
1887
1888 static bool symbol_type_ary_same_type(Symbol_Type_Struct *self,
1889     Symbol_Type_Struct *oth)
1890 {
1891     Symbol *oth_sym;
1892     Symbol_Type_Ary *self_ary, *oth_ary;
1893     Symbol_Type oth_t = oth->methods->get_type();
1894     bool ret = true;
1895
1896     if (self == oth || oth_t == SYMBOL_TYPE_UNKNOWN)
1897         goto out;
1898
1899     self_ary = SYMBOL_TYPE_STRUCT_CONTAINER(self, Symbol_Type_Ary);
1900
1901     if (oth_t == SYMBOL_TYPE_ARY) {
1902         oth_ary = SYMBOL_TYPE_STRUCT_CONTAINER(oth, Symbol_Type_Ary);
1903         ret = self_ary->ary_type->methods->same_type(self_ary->ary_type,
1904             oth_ary->ary_type);
1905     } else if (oth_t == SYMBOL_TYPE_CYCLE) {
1906         Symbol_Type_Cycle *oth_cy =
1907             SYMBOL_TYPE_STRUCT_CONTAINER(oth, Symbol_Type_Cycle);
1908         oth_sym = symbol_table_node_lookup(oth_cy->sym_node,
1909             oth_cy->name, SYMBOL_PROPERTY_TYPE_DEF);
1910         ret = symbol_type_ary_same_type(self, oth_sym->resolved_type);
1911     } else {
1912         ret = false;
1913     }
1914 out:
1915     return ret;
1916 }
1917
1918 static void symbol_type_ary_append_str(Symbol_Type_Struct *self,
1919     String_Builder *sb)
1920 {
1921     Symbol_Type_Ary *t = SYMBOL_TYPE_STRUCT_CONTAINER(self, Symbol_Type_Ary);
1922     if (t->dump_cycle_marked) {
1923         string_builder_append(sb, S("array cycle"));
1924     } else {
1925         t->dump_cycle_marked = true;
1926         string_builder_append(sb, S("array of "));
1927         t->ary_type->methods->append_str(t->ary_type, sb);
1928         t->dump_cycle_marked = false;
1929     }
1930 }
1931
1932 void symbol_type_ary_set_name(Symbol_Type_Ary *self, String name)
1933 {
1934     String new_name = NULL;
1935     if (name)
1936         new_name = string_duplicate(name);
1937
1938     if (self->ary_name)
1939         string_destroy(self->ary_name);
1940
1941     self->ary_name = new_name;
1942 }
1943
1944 static void symbol_type_ary_destroy(Symbol_Type_Struct *self)
1945 {

```

```

1946     Symbol_Type_Ary *t = SYMBOL_TYPE_STRUCT_CONTAINER(self,
1947         Symbol_Type_Ary);
1948     if(t->ary_name)
1949         string_destroy(t->ary_name);
1950
1951     symbol_type_struct_destroy_common(self);
1952     free_mem(t);
1953 }
1954
1955 static Symbol_Type_Struct_Methods symbol_type_ary_methods = {
1956     .get_type = symbol_type_ary_get_type,
1957     .resolve = symbol_type_ary_resolve,
1958     .append_str = symbol_type_ary_append_str,
1959     .destroy = symbol_type_ary_destroy,
1960     .same_type = symbol_type_ary_same_type
1961 };
1962
1963 Symbol_Type_Struct *symbol_type_ary_alloc(Symbol_Table *t,
1964     Symbol_Type_Struct *ary_type, Uns scope_id)
1965 {
1966     Symbol_Type_Ary *s = ALLOC_NEW(Symbol_Type_Ary);
1967     s->ary_type = ary_type;
1968     s->dump_cycle_marked = false;
1969     s->ary_name = NULL;
1970     s->scope_id = scope_id;
1971     s->imp_table_updated = false;
1972     return symbol_type_struct_alloc_finalize(
1973         SYMBOL_TYPE_STRUCT_OF_CONTAINER(s),
1974         &symbol_type_ary_methods, t);
1975 }
1976
1977 /*****
1978
1979 static Symbol_Type symbol_type_func_get_type()
1980 {
1981     return SYMBOL_TYPE_FUNC;
1982 }
1983
1984 static Symbol_Type_Struct *symbol_type_func_resolve(
1985     Symbol_Type_Struct *self, Symbol_Table *t)
1986 {
1987     Vector *tmp_vec;
1988     String tmp_str;
1989     Symbol *saved_sym, *sym;
1990     Symbol_Type_Struct *tmp_type;
1991     Symbol_Type_Func *tfunc = SYMBOL_TYPE_STRUCT_CONTAINER(self,
1992         Symbol_Type_Func);
1993
1994     if (tfunc->is_resolved)
1995         goto out;
1996
1997     if (t->last_resolve_pass)
1998         tfunc->is_resolved = true;
1999
2000     tmp_type = tfunc->return_type->methods->resolve(tfunc->return_type, t);
2001     if (tmp_type)
2002         tfunc->return_type = tmp_type;
2003     else
2004         assert(!tfunc->is_resolved);
2005
2006     saved_sym = t->current_symbol;
2007
2008     tmp_vec = &tfunc->param_identifiers;
2009     VECTOR_FOR_EACH_ENTRY(tmp_vec, tmp_str) {
2010         DLOG("lookup param: %S\n", tmp_str);
2011         sym = symbol_table_node_get(tfunc->body_sym_node,
2012             tmp_str, string_hash_code(tmp_str), SYMBOL_PROPERTY_VAR);
2013         assert(sym);
2014
2015         t->current_symbol = sym;
2016
2017         tmp_type = sym->resolved_type->methods->resolve(
2018             sym->resolved_type, t);
2019

```

```

2020     DEBUGT(def,
2021         if (tfunc->is_resolved)
2022             assert(tmp_type)
2023     );
2024
2025     if (tmp_type) {
2026         sym->resolved_type = tmp_type;
2027         DLOG("parameter: %d\n", tmp_type->methods->get_type());
2028     }
2029
2030     if (tfunc->is_resolved) {
2031         if (tmp_type->methods->get_type() == SYMBOL_TYPE_CYCLE) {
2032             Symbol_Type_Cycle *cycle = SYMBOL_TYPE_STRUCT_CONTAINER(
2033                 tmp_type, Symbol_Type_Cycle);
2034             sym->resolved_type = symbol_table_node_lookup(
2035                 cycle->sym_node, cycle->name,
2036                 SYMBOL_PROPERTY_TYPE_DEF->resolved_type;
2037             }
2038             vector_append(&tfunc->param_types, sym->resolved_type);
2039             DLOG("last func pass. Num params: %U\n",
2040                 vector_size(&tfunc->param_types));
2041         }
2042     }
2043     t->current_symbol = saved_sym;
2044
2045 out:
2046     return self;
2047 }
2048
2049 static bool symbol_type_func_same_type(Symbol_Type_Struct *self,
2050     Symbol_Type_Struct *oth)
2051 {
2052     Symbol_Type_Func *self_func, *oth_func;
2053     Symbol_Type oth_t = oth->methods->get_type();
2054     bool ret = true;
2055     if (oth_t == SYMBOL_TYPE_UNKNOWN)
2056         goto out;
2057     if (oth_t != SYMBOL_TYPE_FUNC) {
2058         ret = false;
2059         goto out;
2060     }
2061     self_func = SYMBOL_TYPE_STRUCT_CONTAINER(self, Symbol_Type_Func);
2062     oth_func = SYMBOL_TYPE_STRUCT_CONTAINER(oth, Symbol_Type_Func);
2063
2064     assert(self_func->is_resolved);
2065     assert(oth_func->is_resolved);
2066
2067     if (self_func->compare_return_type) {
2068         assert(oth_func->compare_return_type);
2069         if (!self_func->return_type->methods->same_type(self_func->return_type,
2070             oth_func->return_type)) {
2071             ret = false;
2072             goto out;
2073         }
2074     }
2075
2076     /* No comparison of parameter names. */
2077     if (!symbol_type_vectors_same_type(&self_func->param_types,
2078         &oth_func->param_types)) {
2079         ret = false;
2080         goto out;
2081     }
2082
2083 out:
2084     return ret;
2085 }
2086
2087 static void symbol_type_func_append_str(Symbol_Type_Struct *self,
2088     String_Builder *sb)
2089 {
2090     Uns vec_size, idx;
2091     Vector *tmp_vec;
2092     Symbol_Type_Struct *tmp_type;
2093     Symbol_Type_Func *tfunc =

```

```

2094     SYMBOL_TYPE_STRUCT_CONTAINER(self, Symbol_Type_Func);
2095
2096     if (tfunc->is_extern_c)
2097         string_builder_append(sb, S("extern (C) func ( ");
2098     else
2099         string_builder_append(sb, S("func ( ");
2100
2101     tmp_vec = &tfunc->param_types;
2102     vec_size = vector_size(tmp_vec);
2103
2104     if (!vec_size)
2105         goto skip_params;
2106
2107     for (idx = 0; idx < vec_size - 1; idx++) {
2108         tmp_type = vector_get(tmp_vec, idx);
2109         tmp_type->methods->append_str(tmp_type, sb);
2110         string_builder_append(sb, S(", "));
2111     }
2112     tmp_type = vector_get(tmp_vec, idx);
2113     tmp_type->methods->append_str(tmp_type, sb);
2114
2115 skip_params:
2116     string_builder_append(sb, S(") : ");
2117
2118     tfunc->return_type->methods->append_str(tfunc->return_type, sb);
2119 }
2120
2121 void symbol_type_func_append_param_identifer(Symbol_Type_Func *self,
2122     String iden)
2123 {
2124     vector_append(&self->param_identifiers, string_duplicate(iden));
2125 }
2126
2127 static void symbol_type_func_destroy(Symbol_Type_Struct *self)
2128 {
2129     Symbol_Type_Func *t = SYMBOL_TYPE_STRUCT_CONTAINER(self,
2130         Symbol_Type_Func);
2131     symbol_type_struct_destroy_common(self);
2132     vector_for_each_destroy(&t->param_identifiers,
2133         (Vector_Destructor) string_destroy);
2134     vector_clear(&t->param_types);
2135
2136     free_mem(t);
2137 }
2138
2139 static Symbol_Type_Struct_Methods symbol_type_func_methods = {
2140     .get_type = symbol_type_func_get_type,
2141     .resolve = symbol_type_func_resolve,
2142     .append_str = symbol_type_func_append_str,
2143     .destroy = symbol_type_func_destroy,
2144     .same_type = symbol_type_func_same_type
2145 };
2146
2147 Symbol_Type_Struct *symbol_type_func_alloc(Symbol_Table *t, bool is_extern_c)
2148 {
2149     Symbol_Type_Func *s = ALLOC_NEW(Symbol_Type_Func);
2150     s->param_identifiers = VECTOR_INIT_SIZE(4);
2151     s->param_types = VECTOR_INIT_SIZE(4);
2152     s->is_resolved = false;
2153     s->compare_return_type = true;
2154     s->is_extern_c = is_extern_c;
2155     s->main_err_int_reported = false;
2156     s->main_param_err_reported = false;
2157     s->imp_table_updated = false;
2158     s->is_concrete_func = true;
2159     return symbol_type_struct_alloc_finalize(
2160         SYMBOL_TYPE_STRUCT_OF_CONTAINER(s),
2161         &symbol_type_func_methods, t);
2162 }
2163
2164 /*****
2165
2166 static Symbol_Type symbol_type_cycle_get_type()
2167 {

```

```

2168     return SYMBOL_TYPE_CYCLE;
2169 }
2170
2171 static Symbol_Type_Struct *symbol_type_cycle_resolve(
2172     Symbol_Type_Struct *self, Symbol_Table *t UNUSED)
2173 {
2174     return self;
2175 }
2176
2177 static bool symbol_type_cycle_same_type(Symbol_Type_Struct *self,
2178     Symbol_Type_Struct *oth)
2179 {
2180     Symbol *oth_sym;
2181     Symbol_Type oth_t = oth->methods->get_type();
2182     bool ret = true;
2183
2184     if (oth_t == SYMBOL_TYPE_UNKNOWN)
2185         goto out;
2186
2187     if (oth_t == SYMBOL_TYPE_CYCLE) {
2188         Symbol_Type_Cycle *oth_cy =
2189             SYMBOL_TYPE_STRUCT_CONTAINER(oth, Symbol_Type_Cycle);
2190         oth_sym = symbol_table_node_lookup(oth_cy->sym_node,
2191             oth_cy->name, SYMBOL_PROPERTY_TYPE_DEF);
2192         ret = oth_sym->resolved_type->methods->same_type(
2193             oth_sym->resolved_type, self);
2194     } else {
2195         ret = oth->methods->same_type(oth, self);
2196     }
2197
2198 out:
2199     return ret;
2200 }
2201
2202 static void symbol_type_cycle_append_str(Symbol_Type_Struct *self UNUSED,
2203     String_Builder *sb)
2204 {
2205     Symbol_Type_Cycle *s = SYMBOL_TYPE_STRUCT_CONTAINER(self,
2206         Symbol_Type_Cycle);
2207     string_builder_append(sb, S("cycle["));
2208     string_builder_append(sb, s->name);
2209     string_builder_append(sb, S("]"));
2210 }
2211
2212 static void symbol_type_cycle_destroy(Symbol_Type_Struct *self)
2213 {
2214     Symbol_Type_Cycle *t = SYMBOL_TYPE_STRUCT_CONTAINER(self,
2215         Symbol_Type_Cycle);
2216     symbol_type_struct_destroy_common(self);
2217     free_mem(t);
2218 }
2219
2220 static Symbol_Type_Struct_Methods symbol_type_cycle_methods = {
2221     .get_type = symbol_type_cycle_get_type,
2222     .resolve = symbol_type_cycle_resolve,
2223     .append_str = symbol_type_cycle_append_str,
2224     .destroy = symbol_type_cycle_destroy,
2225     .same_type = symbol_type_cycle_same_type
2226 };
2227
2228 Symbol_Type_Struct *symbol_type_cycle_alloc(Symbol_Table *t, String name)
2229 {
2230     Symbol_Type_Cycle *s = ALLOC_NEW(Symbol_Type_Cycle);
2231     s->name = name;
2232     s->scope_id = t->current_sym_table_node->scope_id;
2233     s->sym_node = t->current_sym_table_node;
2234     return symbol_type_struct_alloc_finalize(
2235         SYMBOL_TYPE_STRUCT_OF_CONTAINER(s),
2236         &symbol_type_cycle_methods, t);
2237 }
2238
2239 /*****
2240
2241 /*****

```

```

2242
2243 bool symbol_rec_comp_struct_compare(Symbol_Rec_Comp_Struct *search_comp,
2244     Hash_Map_Slot *map_slot)
2245 {
2246     Symbol_Rec_Comp_Struct *map_comp = SYMBOL_REC_COMP_STRUCT_OF(map_slot);
2247     return (map_comp->rec1 == search_comp->rec1 &&
2248         map_comp->rec2 == search_comp->rec2) ||
2249         (map_comp->rec2 == search_comp->rec1 &&
2250         (map_comp->rec1 == search_comp->rec2));
2251 }
2252
2253 void symbol_rec_comp_struct_destroy(Hash_Map_Slot *slot)
2254 {
2255     Symbol_Rec_Comp_Struct *c = SYMBOL_REC_COMP_STRUCT_OF(slot);
2256     free_mem(c);
2257 }
2258
2259 bool ___symbol_map_comparator(String search_iden, Hash_Map_Slot *slot)
2260 {
2261     Symbol *map_sym = SYMBOL_OF_SLOT(slot);
2262     return !string_compare(search_iden, map_sym->identifier);
2263 }
2264
2265 static inline void symbol_destroy(Symbol *sym)
2266 {
2267     if (sym->unique_name)
2268         string_destroy(sym->unique_name);
2269     string_destroy(sym->identifier);
2270     free_mem(sym);
2271 }
2272
2273 static void symbol_dblist_destroy(Double_List_Node *n)
2274 {
2275     symbol_destroy(&TYPE_DEF_SYMBOL_OF_DBNODE(n)->symbol);
2276 }
2277
2278 static void symbol_hash_destructor(Hash_Map_Slot *slot)
2279 {
2280     Symbol *sym = SYMBOL_OF_SLOT(slot);
2281     symbol_destroy(sym);
2282 }
2283
2284 static void type_def_symbol_hash_destructor(Hash_Map_Slot *slot)
2285 {
2286     Symbol *sym = SYMBOL_OF_SLOT(slot);
2287     double_list_for_each_destroy(&TYPE_DEF_SYMBOL_OF_SYMBOL(sym)->dbnode,
2288         symbol_dblist_destroy);
2289     symbol_destroy(sym);
2290 }
2291
2292 static void symbol_type_destructor(Double_List_Node *n)
2293 {
2294     Symbol_Type_Struct *t = SYMBOL_TYPE_STRUCT_OF_DBNODE(n);
2295     t->methods->destroy(t);
2296 }
2297
2298 static void symbol_table_node_destroy(Double_List_Node *dbnode)
2299 {
2300     Hash_Map *map;
2301     Symbol_Table_Node *n = SYMBOL_TABLE_NODE_OF(dbnode);
2302     map = &n->symbol_maps[SYMBOL_PROPERTY_TYPE_DEF];
2303     hash_map_for_each_destroy(map, type_def_symbol_hash_destructor);
2304     map = &n->symbol_maps[SYMBOL_PROPERTY_FUNC];
2305     hash_map_for_each_destroy(map, symbol_hash_destructor);
2306     map = &n->symbol_maps[SYMBOL_PROPERTY_VAR];
2307     hash_map_for_each_destroy(map, symbol_hash_destructor);
2308     vector_for_each_destroy(&n->import_dependencies,
2309         (Vector_Destructor) string_destroy);
2310     vector_clear(&n->import_dependencies_loc);
2311     hash_map_for_each_destroy(&n->func_iden_map, symbol_func_map_hash_destroy);
2312     hash_map_for_each_destroy(&n->func_remaps, symbol_func_remap_hash_destroy);
2313     free_mem(n);
2314 }

```

```

2316 }
2317
2318 void symbol_table_clear(Symbol_Table *t)
2319 {
2320     double_list_for_each_destroy(&t->all_nodes, symbol_table_node_destroy);
2321     double_list_for_each_destroy(&t->all_symbol_types, symbol_type_destructor);
2322     vector_for_each_destroy(&t->location_names,
2323         (Vector_Destructor)string_destroy);
2324     hash_map_for_each_destroy(&t->record_comparisons,
2325         (Hash_Map_Destructor)symbol_rec_comp_struct_destroy);
2326     t->symbol_type_struct_bool = NULL;
2327     t->symbol_type_struct_char = NULL;
2328     t->symbol_type_struct_int = NULL;
2329     t->symbol_type_struct_string = NULL;
2330     t->symbol_type_struct_unknown = NULL;
2331 }
2332
2333
2334 void symbol_table_dump(Symbol_Table *t, Const_String prefix)
2335 {
2336     Double_List *list;
2337     Double_List_Node *dbnode;
2338     Symbol_Table_Node *n;
2339     Hash_Map_Slot *slot;
2340     Hash_Map *map;
2341     Symbol *sym;
2342     String_Builder sb;
2343
2344     String dumpf = string_from_format(S("%S.vitaly.symbol-table"), prefix);
2345     FILE *stream = file_open(dumpf, S("w"));
2346     if (!stream) {
2347         report_error(prefix, S("Unable to open file %S for symbol "
2348             "table dump [%m]\n"), dumpf);
2349         goto out_dumpf;
2350     }
2351
2352     sb = STRING_BUILDER_INIT();
2353     list = &t->all_nodes;
2354
2355     DOUBLE_LIST_FOR_EACH(list, dbnode) {
2356         n = SYMBOL_TABLE_NODE_OF(dbnode);
2357
2358         map = &n->symbol_maps[SYMBOL_PROPERTY_TYPE_DEF];
2359         file_print_message(stream, S("-----\n"));
2360         file_print_message(stream, S("Node: %p | Parent: %p\n"),
2361             n, n->parent);
2362         file_print_message(stream, S("-----\n"));
2363         HASH_MAP_FOR_EACH(map, slot) {
2364             sym = SYMBOL_OF_SLOT(slot);
2365             string_builder_assign(&sb, S("type "));
2366             string_builder_append(&sb, sym->identifier);
2367             string_builder_append(&sb, S(" => "));
2368             sym->resolved_type->methods->append_str(
2369                 sym->resolved_type, &sb);
2370             string_builder_append(&sb, S("\n"));
2371             file_print_message(stream, S("%S\n"),
2372                 string_builder_const_str(&sb));
2373         }
2374         map = &n->symbol_maps[SYMBOL_PROPERTY_FUNC];
2375         HASH_MAP_FOR_EACH(map, slot) {
2376             sym = SYMBOL_OF_SLOT(slot);
2377             string_builder_assign(&sb, S("func "));
2378             string_builder_append(&sb, sym->identifier);
2379             string_builder_append(&sb, S(" => "));
2380             sym->resolved_type->methods->append_str(
2381                 sym->resolved_type, &sb);
2382             string_builder_append(&sb, S("\n"));
2383             file_print_message(stream, S("%S\n"),
2384                 string_builder_const_str(&sb));
2385         }
2386         map = &n->symbol_maps[SYMBOL_PROPERTY_VAR];
2387         HASH_MAP_FOR_EACH(map, slot) {

```

```

2390         sym = SYMBOL_OF_SLOT(slot);
2391         string_builder_assign(&sb, S("var "));
2392         string_builder_append(&sb, sym->identifier);
2393         string_builder_append(&sb, S(" => "));
2394         sym->resolved_type->methods->append_str(
2395             sym->resolved_type, &sb);
2396         string_builder_append(&sb, S("\n"));
2397         file_print_message(stream, S("%S\n"),
2398             string_builder_const_str(&sb));
2399     }
2400 }
2401 file_print_message(stream, S("-----"
2402     "-----\n"));
2403 string_builder_clear(&sb);
2404
2405 file_close(stream);
2406
2407 out_dumpf:
2408     string_destroy(dumpf);
2409 }
2410
2411 void symbol_table_dump_graph(Symbol_Table *t, Const_String file_prefix)
2412 {
2413     Dot_Printer *printer = dot_printer_init(file_prefix,
2414         S("vitaly.symbol-table"), S("BT"));
2415     Symbol_Table_Node *n;
2416     Hash_Map_Slot *slot;
2417     Hash_Map *map;
2418     Symbol *sym;
2419     String tmp_str;
2420     Const_String node_type;
2421     String_Builder sb = STRING_BUILDER_INIT();
2422     Double_List_Node *dbnode;
2423     Double_List *list = &t->all_nodes;
2424     DOUBLE_LIST_FOR_EACH(list, dbnode) {
2425         n = SYMBOL_TABLE_NODE_OF(dbnode);
2426
2427         switch (n->type) {
2428             case SYMBOL_TABLE_NODE_GLOBAL:
2429                 node_type = S("GLOBAL SCOPE");
2430                 break;
2431             case SYMBOL_TABLE_NODE_IMPORT:
2432                 node_type = S("IMPORT SCOPE");
2433                 break;
2434             case SYMBOL_TABLE_NODE_FUNC:
2435                 node_type = S("FUNCTION SCOPE");
2436                 break;
2437             case SYMBOL_TABLE_NODE_INTERMEDIATE:
2438                 node_type = S("INTERMEDIATE SCOPE");
2439                 break;
2440             case SYMBOL_TABLE_NODE_REC:
2441                 node_type = S("RECORD SCOPE");
2442                 break;
2443             default:
2444                 fatal_error(S("unexpected symbol table node type for "
2445                     "symbol table graph dump\n"));
2446         }
2447
2448         dot_printer_insert_relation(printer, n, n->parent);
2449         dot_printer_begin_table(printer, n, 2);
2450         map = &n->symbol_maps[SYMBOL_PROPERTY_TYPE_DEF];
2451         dot_printer_insert_merge_row(printer, node_type);
2452         dot_printer_insert_merge_row(printer, S("Type definitions:"));
2453         HASH_MAP_FOR_EACH(map, slot) {
2454             string_builder_assign(&sb, S(""));
2455             sym = SYMBOL_OF_SLOT(slot);
2456             sym->resolved_type->methods->append_str(
2457                 sym->resolved_type, &sb);
2458             tmp_str = string_from_format(S("type %S"), sym->identifier);
2459             dot_printer_insert_row(printer, 2,
2460                 tmp_str, string_builder_const_str(&sb));
2461             string_destroy(tmp_str);
2462
2463             Double_List_Node *dbnode;

```



```

2464     Type_Def_Symbol *td_sym = TYPE_DEF_SYMBOL_OF_SYMBOL(sym);
2465     DOUBLE_LIST_FOR_EACH(&td_sym->dbnode, dbnode) {
2466         sym = (Symbol *) TYPE_DEF_SYMBOL_OF_DBNODE(dbnode);
2467         if (sym->resolved_type) {
2468             string_builder_assign(&sb, S(""));
2469             sym->resolved_type->methods->append_str(
2470                 sym->resolved_type, &sb);
2471             tmp_str = string_from_format(S("type %S"), sym->identifier);
2472             dot_printer_insert_row(printer, 2,
2473                 tmp_str, string_builder_const_str(&sb));
2474             string_destroy(tmp_str);
2475         }
2476     }
2477 }
2478 map = &n->symbol_maps[SYMBOL_PROPERTY_FUNC];
2479 dot_printer_insert_merge_row(printer, S("Functions:"));
2480 HASH_MAP_FOR_EACH(map, slot) {
2481     string_builder_assign(&sb, S(""));
2482     sym = SYMBOL_OF_SLOT(slot);
2483     sym->resolved_type->methods->append_str(
2484         sym->resolved_type, &sb);
2485     tmp_str = string_from_format(S("func %S"), sym->identifier);
2486     dot_printer_insert_row(printer, 2,
2487         tmp_str, string_builder_const_str(&sb));
2488     string_destroy(tmp_str);
2489 }
2490 map = &n->symbol_maps[SYMBOL_PROPERTY_VAR];
2491 dot_printer_insert_merge_row(printer, S("Variables:"));
2492 HASH_MAP_FOR_EACH(map, slot) {
2493     string_builder_assign(&sb, S(""));
2494     sym = SYMBOL_OF_SLOT(slot);
2495     sym->resolved_type->methods->append_str(
2496         sym->resolved_type, &sb);
2497     tmp_str = string_from_format(S("var %S"), sym->identifier);
2498     dot_printer_insert_row(printer, 2,
2499         tmp_str, string_builder_const_str(&sb));
2500     string_destroy(tmp_str);
2501 }
2502 dot_printer_end_table(printer);
2503 }
2504 string_builder_clear(&sb);
2505 dot_printer_fin_com_des(printer, S("pdf"));
2506 }
2507
2508 static void symbol_table_node_resolve_type_def(Symbol_Table_Node *n,
2509     Symbol_Table *t)
2510 {
2511     Hash_Map_Slot *slot;
2512     Hash_Map *map;
2513     Symbol *sym;
2514     Type_Def_Symbol *td_sym;
2515
2516     map = &n->symbol_maps[SYMBOL_PROPERTY_TYPE_DEF];
2517     t->current_resolve_property = SYMBOL_PROPERTY_TYPE_DEF;
2518     HASH_MAP_FOR_EACH(map, slot) {
2519         sym = SYMBOL_OF_SLOT(slot);
2520         t->current_symbol = sym;
2521         td_sym = TYPE_DEF_SYMBOL_OF_SYMBOL(sym);
2522         td_sym->report_cycle = true;
2523         sym->resolved_type = __type_def_symbol_resolve(td_sym, t);
2524         td_sym->report_cycle = false;
2525     }
2526
2527     t->last_resolve_pass = true;
2528     HASH_MAP_FOR_EACH(map, slot) {
2529         sym = SYMBOL_OF_SLOT(slot);
2530         t->current_symbol = sym;
2531         td_sym = TYPE_DEF_SYMBOL_OF_SYMBOL(sym);
2532         td_sym->report_cycle = true;
2533         sym->resolved_type = __type_def_symbol_resolve(td_sym, t);
2534         td_sym->report_cycle = false;
2535     }
2536
2537     /* We are not destroying all of the type def symbols.
2538      * The first element of the list (td_sym) is not destroyed. */

```

```

2538     double_list_for_each_destroy(&td_sym->dbnode,
2539         symbol_dblist_destroy);
2540     sym->resolved_type = symbol_type_finalize(sym->resolved_type, t);
2541 }
2542 t->last_resolve_pass = false;
2543 }
2544
2545 static inline bool symbol_func_same(Symbol *lhs, Symbol *rhs)
2546 {
2547     Symbol_Type_Struct *tmp;
2548     Symbol_Type_Func *lhs_f, *rhs_f;
2549
2550     if (lhs->resolved_type->methods->get_type() == SYMBOL_TYPE_UNKNOWN ||
2551         rhs->resolved_type->methods->get_type() == SYMBOL_TYPE_UNKNOWN)
2552         return false;
2553
2554     lhs_f = SYMBOL_TYPE_STRUCT_CONTAINER(lhs->resolved_type, Symbol_Type_Func);
2555     rhs_f = SYMBOL_TYPE_STRUCT_CONTAINER(rhs->resolved_type, Symbol_Type_Func);
2556
2557     Vector *v = &lhs_f->param_types;
2558     VECTOR_FOR_EACH_ENTRY(v, tmp) {
2559         if (tmp->methods->get_type() == SYMBOL_TYPE_UNKNOWN)
2560             return false;
2561     }
2562
2563     v = &rhs_f->param_types;
2564     VECTOR_FOR_EACH_ENTRY(v, tmp) {
2565         if (tmp->methods->get_type() == SYMBOL_TYPE_UNKNOWN)
2566             return false;
2567     }
2568
2569     #if 0
2570     if (lhs_f->is_extern_c && rhs_f->is_extern_c)
2571         return true;
2572     #endif
2573
2574     lhs_f->compare_return_type = false;
2575     rhs_f->compare_return_type = false;
2576
2577     bool ret = lhs->resolved_type->methods->same_type(lhs->resolved_type,
2578         rhs->resolved_type);
2579
2580     lhs_f->compare_return_type = true;
2581     rhs_f->compare_return_type = true;
2582
2583     return ret;
2584 }
2585
2586 static void symbol_func_report_errors(Vector *err, bool is_main)
2587 {
2588     if (vector_is_empty(err))
2589         return;
2590
2591     Symbol *fsym = vector_peek_first(err);
2592     STRING(loc_str, "");
2593
2594     Const_String func_name = STRING_AFTER_LAST(fsym->identifier, '.');
2595     assert(func_name);
2596
2597     File_Location *err_location;
2598
2599     inline void import_report_main()
2600     {
2601         report_error_location(err_location,
2602             S("multiple declarations of " QFY("%S")
2603             " function imported from:\n%S"),
2604             func_name, loc_str);
2605     }
2606
2607     inline void import_report_vit()
2608     {
2609         if (!string_is_empty(loc_str))
2610             report_error_location(err_location,
2611                 S("conflicting declarations of function " QFY("%S")

```

```

2612         " imported from:\n%S"),
2613         func_name, loc_str);
2614     }
2615
2616     inline void import_report_c()
2617     {
2618         report_error_location(err_location,
2619             S("conflicting declarations of " QFY("extern(C)")
2620             " function " QFY("%S") " imported from:\n%S"),
2621             func_name, loc_str);
2622     }
2623
2624     inline void report_main()
2625     {
2626         report_error_location(err_location,
2627             S("conflicting declarations of " QFY("%S") " function"
2628             " here:\n%S"),
2629             func_name, loc_str);
2630     }
2631
2632     inline void report_vit()
2633     {
2634         report_error_location(err_location,
2635             S("conflicting declarations of function " QFY("%S")
2636             " involving:\n%S"),
2637             func_name, loc_str);
2638     }
2639
2640     inline void report_c()
2641     {
2642         report_error_location(err_location,
2643             S("conflicting declarations of " QFY("extern(C)")
2644             " function " QFY("%S") " here:\n%S"),
2645             func_name, loc_str);
2646     }
2647
2648     bool is_extern_c_report = true;
2649
2650     inline void set_error_strings(bool use_location) {
2651         VECTOR_FOR_EACH_ENTRY(err, fsym) {
2652             assert(fsym->resolved_type->methods->get_type() ==
2653                 SYMBOL_TYPE_FUNC);
2654
2655             Symbol_Type_Func *func = SYMBOL_TYPE_STRUCT_CONTAINER(
2656                 fsym->resolved_type, Symbol_Type_Func);
2657
2658             assert(fsym->resolved_type->methods->get_type() ==
2659                 SYMBOL_TYPE_FUNC);
2660             if (!func->is_extern_c)
2661                 is_extern_c_report = false;
2662
2663             err_location = fsym->location;
2664
2665             if (use_location) {
2666                 string_append_format(loc_str, S("\t%F\n"), fsym->location);
2667             } else {
2668                 String s = string_between_alloc(fsym->unique_name, '.');
2669                 string_append_format(loc_str, S("\t%S\n"), s);
2670                 string_destroy(s);
2671             }
2672         }
2673     }
2674
2675     if (symbol_get_symbol_table_node_type(fsym) ==
2676         SYMBOL_TABLE_NODE_IMPORT) {
2677         set_error_strings(false);
2678         if (is_main) {
2679             import_report_main();
2680         } else {
2681             if (is_extern_c_report)
2682                 import_report_c();
2683             else
2684                 import_report_vit();
2685         }
2686     }

```

```

2686     } else {
2687         set_error_strings(true);
2688         if (is_main) {
2689             report_main();
2690         } else {
2691             if (is_extern_c_report)
2692                 report_c();
2693             else
2694                 report_vit();
2695         }
2696     }
2697     string_clear(loc_str);
2698 }
2699
2700 static void symbol_func_append_errors(Symbol_Func_Map *func_map,
2701     Vector *errors)
2702 {
2703     Vector *vec = &func_map->overload_idens;
2704
2705     inline void append_symbols(Symbol *lhs, Symbol *rhs)
2706     {
2707         Symbol_Type_Func *lhs_f = SYMBOL_TYPE_STRUCT_CONTAINER(
2708             lhs->resolved_type, Symbol_Type_Func);
2709         Symbol_Type_Func *rhs_f = SYMBOL_TYPE_STRUCT_CONTAINER(
2710             rhs->resolved_type, Symbol_Type_Func);
2711
2712         Vector *err;
2713         Symbol *tmp;
2714         Symbol_Type_Func *tmp_f;
2715
2716         #if 0
2717         if (lhs_f->is_extern_c && rhs_f->is_extern_c)
2718             return;
2719         #endif
2720
2721         if (lhs_f->is_extern_c && rhs_f->is_extern_c) {
2722             if (lhs_f->return_type->methods->same_type(
2723                 lhs_f->return_type, rhs_f->return_type))
2724                 return;
2725
2726             VECTOR_FOR_EACH_ENTRY(errors, err) {
2727                 tmp = vector_get(err, 0);
2728                 tmp_f = SYMBOL_TYPE_STRUCT_CONTAINER(
2729                     tmp->resolved_type, Symbol_Type_Func);
2730                 if (tmp_f->is_extern_c) {
2731                     if (!vector_contains_ptr(err, lhs))
2732                         vector_append(err, lhs);
2733                     if (!vector_contains_ptr(err, rhs))
2734                         vector_append(err, rhs);
2735                     return;
2736                 }
2737             }
2738         } else {
2739             VECTOR_FOR_EACH_ENTRY(errors, err) {
2740                 tmp = vector_get(err, 0);
2741                 if (symbol_func_same(tmp, lhs)) {
2742                     if (!vector_contains_ptr(err, lhs))
2743                         vector_append(err, lhs);
2744                     if (!vector_contains_ptr(err, rhs))
2745                         vector_append(err, rhs);
2746                     return;
2747                 }
2748             }
2749         }
2750         err = vector_alloc_size(4);
2751         vector_append(errors, err);
2752         vector_append(err, lhs);
2753         vector_append(err, rhs);
2754     }
2755
2756     Uns size = vector_size(vec);
2757     for (Uns i = 0; i < size; i++) {
2758         Symbol *lhs = vector_get(vec, i);
2759

```

```

2760     if (lhs->resolved_type->methods->get_type() == SYMBOL_TYPE_UNKNOWN)
2761         continue;
2762     assert(lhs->resolved_type->methods->get_type() == SYMBOL_TYPE_FUNC);
2763
2764     for (Uns j = i + 1; j < size; j++) {
2765         Symbol *rhs = vector_get(vec, j);
2766         if (rhs->resolved_type->methods->get_type() == SYMBOL_TYPE_UNKNOWN)
2767             continue;
2768
2769         assert(rhs->resolved_type->methods->get_type() ==
2770             SYMBOL_TYPE_FUNC);
2771
2772         Symbol_Type_Func *lhs_f, *rhs_f;
2773         lhs_f = SYMBOL_TYPE_STRUCT_CONTAINER(lhs->resolved_type,
2774             Symbol_Type_Func);
2775         rhs_f = SYMBOL_TYPE_STRUCT_CONTAINER(rhs->resolved_type,
2776             Symbol_Type_Func);
2777
2778         if (lhs_f->body_sym_node->parent != rhs_f->body_sym_node->parent)
2779             continue;
2780
2781         if (!file_location_cmp(lhs->location, rhs->location))
2782             continue;
2783
2784         if (symbol_func_same(lhs, rhs))
2785             append_symbols(lhs, rhs);
2786     }
2787 }
2788 }
2789
2790 void symbol_func_map_compare(Symbol_Func_Map *func_map, Symbol_Table *t)
2791 {
2792     if (func_map->is_verified)
2793         return;
2794     if (import_handler_is_merge_table(t))
2795         return;
2796     func_map->is_verified = true;
2797
2798     VECTOR(errors); // Vector of vectors
2799
2800     Vector *vec = &func_map->overload_idens;
2801     if (!string_compare(MAIN_FUNC_STR, func_map->func_iden)) {
2802         assert(!vector_is_empty(vec));
2803         Symbol *sym = vector_get(vec, 0);
2804         if (vector_size(vec) > 1 &&
2805             !string_compare(MAIN_FUNC_STR, sym->identifier)) {
2806             symbol_func_report_errors(&func_map->overload_idens, true);
2807             goto out;
2808         }
2809     }
2810
2811     symbol_func_append_errors(func_map, &errors);
2812
2813     Vector *err;
2814     VECTOR_FOR_EACH_ENTRY(&errors, err)
2815         symbol_func_report_errors(err, false);
2816
2817     void symbol_error_func_destructor(void *vec)
2818     {
2819         Vector *v = vec;
2820         Symbol *fsym;
2821         VECTOR_FOR_EACH_ENTRY(v, fsym)
2822             fsym->resolved_type = symbol_type_unknown_alloc(t);
2823         vector_destroy(v, NULL);
2824     }
2825
2826     out:
2827     vector_for_each_destroy(&errors, symbol_error_func_destructor);
2828 }
2829
2830 static void symbol_table_node_resolve_func(Symbol_Table_Node *n,
2831     Symbol_Table *t)
2832 {
2833     Hash_Map_Slot *slot;

```

```

2834     Hash_Map *map;
2835     Symbol *sym;
2836
2837     t->current_resolve_property = SYMBOL_PROPERTY_FUNC;
2838     map = &n->symbol_maps[SYMBOL_PROPERTY_FUNC];
2839     HASH_MAP_FOR_EACH(map, slot) {
2840         sym = SYMBOL_OF_SLOT(slot);
2841         t->current_symbol = sym;
2842         sym->resolved_type = __symbol_resolve(sym, t);
2843     }
2844
2845     t->last_resolve_pass = true;
2846     map = &n->symbol_maps[SYMBOL_PROPERTY_FUNC];
2847     HASH_MAP_FOR_EACH(map, slot) {
2848         sym = SYMBOL_OF_SLOT(slot);
2849         t->current_symbol = sym;
2850         sym->resolved_type = __symbol_resolve(sym, t);
2851         sym->resolved_type = symbol_type_finalize(sym->resolved_type, t);
2852     }
2853
2854     t->last_resolve_pass = false;
2855
2856     if (n->type == SYMBOL_TABLE_NODE_IMPORT)
2857         return;
2858
2859     map = &n->func_iden_map;
2860     HASH_MAP_FOR_EACH(map, slot) {
2861         Symbol_Func_Map *func_map = SYMBOL_FUNC_MAP_OF(slot);
2862         symbol_func_map_compare(func_map, t);
2863     }
2864 }
2865
2866 static void symbol_table_node_resolve_var(Symbol_Table_Node *n,
2867     Symbol_Table *t)
2868 {
2869     Hash_Map_Slot *slot;
2870     Hash_Map *map;
2871     Symbol *sym;
2872
2873     t->current_resolve_property = SYMBOL_PROPERTY_VAR;
2874     map = &n->symbol_maps[SYMBOL_PROPERTY_VAR];
2875     HASH_MAP_FOR_EACH(map, slot) {
2876         sym = SYMBOL_OF_SLOT(slot);
2877         t->current_symbol = sym;
2878         sym->resolved_type = __symbol_resolve(sym, t);
2879     }
2880
2881     t->last_resolve_pass = true;
2882     map = &n->symbol_maps[SYMBOL_PROPERTY_VAR];
2883     HASH_MAP_FOR_EACH(map, slot) {
2884         sym = SYMBOL_OF_SLOT(slot);
2885         t->current_symbol = sym;
2886         sym->resolved_type = __symbol_resolve(sym, t);
2887         sym->resolved_type = symbol_type_finalize(sym->resolved_type, t);
2888     }
2889
2890     t->last_resolve_pass = false;
2891 }
2892
2893 static void symbol_table_node_resolve(Symbol_Table_Node *n, Symbol_Table *t)
2894 {
2895     t->current_sym_table_node = n;
2896
2897     symbol_table_node_resolve_type_def(n, t);
2898     symbol_table_node_resolve_func(n, t);
2899     symbol_table_node_resolve_var(n, t);
2900 }
2901
2902 void symbol_table_resolve(Symbol_Table *t)
2903 {
2904     Double_List_Node *n;
2905     Double_List *list = &t->all_nodes;
2906     DOUBLE_LIST_FOR_EACH(list, n)
2907         symbol_table_node_resolve(SYMBOL_TABLE_NODE_OF(n), t);

```

```

2908 }
2909
2910 static void symbol_init(Symbol *sym,
2911     File_Location *loc,
2912     Symbol_Type_Struct *type,
2913     Const_String iden,
2914     Const_String unique_name,
2915     Symbol_Table_Node *sym_node)
2916 {
2917     sym->location = loc;
2918     sym->resolved_type = type;
2919     if(unique_name)
2920         sym->unique_name = string_duplicate(unique_name);
2921     else
2922         sym->unique_name = NULL;
2923     sym->sym_node = sym_node;
2924     sym->identifier = string_duplicate(iden);
2925 }
2926
2927 void __symbol_table_node_insert(Symbol_Table_Node *node,
2928     Const_String iden, Const_String unique_name, Uns hash_code,
2929     Symbol_Type_Struct *type, Symbol_Property property,
2930     File_Location *loc)
2931 {
2932     Symbol *symbol;
2933     if (property == SYMBOL_PROPERTY_TYPE_DEF) {
2934         Type_Def_Symbol *type_def_sym = ALLOC_NEW(Type_Def_Symbol);
2935         type_def_sym->cycle_marked = false;
2936         type_def_sym->report_cycle = false;
2937
2938         symbol = &type_def_sym->symbol;
2939
2940         Symbol *dup = symbol_table_node_get(node,
2941             (String)iden, hash_code, property);
2942
2943         if (!dup) {
2944             type_def_sym->dbnode = DOUBLE_LIST_INIT(type_def_sym->dbnode);
2945             hash_map_insert(&node->symbol_maps[property],
2946                 &symbol->hash_slot, hash_code);
2947         } else {
2948             double_list_append(&TYPE_DEF_SYMBOL_OF_SYMBOL(dup)->dbnode,
2949                 &type_def_sym->dbnode);
2950         }
2951     } else if (property == SYMBOL_PROPERTY_FUNC) {
2952         Symbol_Func_Map *fmap;
2953         Hash_Map_Slot *fslot = hash_map_get(&node->func_iden_map,
2954             (String)iden, hash_code);
2955
2956         if (fslot) {
2957             fmap = SYMBOL_FUNC_MAP_OF(fslot);
2958         } else {
2959             fmap = symbol_func_map_alloc(iden, node);
2960             hash_map_insert(&node->func_iden_map, &fmap->hash_slot, hash_code);
2961         }
2962
2963         symbol = ALLOC_NEW(Symbol);
2964         vector_append(&fmap->overload_idens, symbol);
2965         iden = unique_name;
2966         hash_code = string_hash_code(iden);
2967         hash_map_insert(&node->symbol_maps[property],
2968             &symbol->hash_slot, hash_code);
2969     } else {
2970         if (symbol_table_node_get(node, (String)iden,
2971             hash_code, property))
2972             assert(!symbol_table_node_get(node, (String)iden,
2973                 hash_code, property));
2974         symbol = ALLOC_NEW(Symbol);
2975         hash_map_insert(&node->symbol_maps[property],
2976             &symbol->hash_slot, hash_code);
2977     }
2978 }
2979
2980 symbol_init(symbol, loc, type, iden, unique_name, node);
2981 }

```

```

2982
2983 static inline String symbol_table_get_name_from(File_Location *loc)
2984 {
2985     return string_from_format(S("%U.%U"), loc->line, loc->column);
2986 }
2987
2988 static inline String symbol_table_get_unique_name_from(Const_String prefix,
2989 File_Location *loc)
2990 {
2991     return string_from_format(S("%S.%U.%U"), prefix, loc->line, loc->column);
2992 }
2993
2994 void __symbol_table_insert_location(Symbol_Table *t, Symbol_Table_Node *node,
2995 Const_String prefix, Symbol_Type_Struct *type, File_Location *loc,
2996 Symbol_Property property)
2997 {
2998     assert(property != SYMBOL_PROPERTY_FUNC);
2999
3000     String name = symbol_table_get_name_from(loc);
3001     Uns hash = string_hash_code(name);
3002
3003     assert(!symbol_table_node_get(node, name, string_hash_code(name),
3004 property));
3005
3006     String unique_name = symbol_table_get_unique_name_from(prefix, loc);
3007     vector_append(&t->location_names, name);
3008     DLOG("insert %S in %p\n", name, node);
3009     __symbol_table_node_insert(node, name, unique_name, hash, type,
3010 property, loc);
3011     string_destroy(unique_name);
3012 }
3013
3014 void symbol_table_insert_location(Symbol_Table *t, Symbol_Table_Node *node,
3015 Const_String prefix, Symbol_Type_Struct *type, File_Location *loc)
3016 {
3017     __symbol_table_insert_location(t, node, prefix, type, loc,
3018 SYMBOL_PROPERTY_VAR);
3019 }
3020
3021 Symbol *__symbol_table_get_from_location(Symbol_Table_Node *node,
3022 File_Location *loc, Symbol_Property property)
3023 {
3024     String name = symbol_table_get_name_from(loc);
3025     DLOG("get %S in %p\n", name, node);
3026     Symbol *sym = symbol_table_node_get(node, name, string_hash_code(name),
3027 property);
3028     assert(sym);
3029     string_destroy(name);
3030     return sym;
3031 }
3032
3033 Symbol *symbol_table_get_from_location(Symbol_Table_Node *node,
3034 File_Location *loc)
3035 {
3036     return __symbol_table_get_from_location(node, loc, SYMBOL_PROPERTY_VAR);
3037 }
3038
3039 void symbol_table_node_insert(Symbol_Table_Node *node,
3040 Const_String iden, Const_String unique_name,
3041 Symbol_Type_Struct *type, Symbol_Property property,
3042 File_Location *loc)
3043 {
3044     Uns hash = string_hash_code(iden);
3045     DLOG("insert: %S\n", iden);
3046     assert(!hash_map_contains(&node->symbol_maps[property],
3047 (String)iden, hash));
3048     __symbol_table_node_insert(node, iden, unique_name,
3049 hash, type, property, loc);
3050 }
3051
3052 Symbol_Func_Map *__symbol_table_node_lookup_func_map(
3053 Symbol_Table_Node *sym_node, String iden, Uns hash_code)
3054 {
3055     Symbol_Func_Map *fmap;

```



```

3056     for (;;) {
3057         fmap = symbol_table_node_get_func_map(sym_node, iden, hash_code);
3058         if (fmap)
3059             break;
3060         sym_node = sym_node->parent;
3061         if (!sym_node)
3062             goto out;
3063     }
3064 out:
3065     return fmap;
3066 }
3067
3068 Symbol *___symbol_table_node_lookup(Symbol_Table_Node *sym_node,
3069     String iden, Uns hash_code, Symbol_Property property)
3070 {
3071     Symbol *sym;
3072     for (;;) {
3073         sym = symbol_table_node_get(sym_node, iden, hash_code, property);
3074         if (sym)
3075             break;
3076         sym_node = sym_node->parent;
3077         if (!sym_node)
3078             goto out;
3079     }
3080
3081     if (sym->resolved_type->methods->get_type() == SYMBOL_TYPE_CYCLE) {
3082         Symbol_Type_Cycle *cycle = SYMBOL_TYPE_STRUCT_CONTAINER(
3083             sym->resolved_type, Symbol_Type_Cycle);
3084         Symbol *tmp = ___symbol_table_node_lookup(cycle->sym_node, cycle->name,
3085             string_hash_code(cycle->name), SYMBOL_PROPERTY_TYPE_DEF);
3086         sym->resolved_type = tmp->resolved_type;
3087     }
3088 }
3089 out:
3090     return sym;
3091 }
3092
3093 static HASH_MAP(hash_map, pointer_hash_map_compare);
3094
3095 static Symbol_Table_Node *symbol_table_node_copy_hash_insert_node(
3096     Symbol_Table *table, Symbol_Table_Node *old_node, Vector *needed_nodes)
3097 {
3098     Pointer_Slot *ps, *lookup;
3099     Hash_Map_Slot *slot = hash_map_get(&hash_map, old_node,
3100         hash_map_aligned_ptr_hash(old_node));
3101
3102     if (!slot) {
3103         slot = hash_map_get(&hash_map, old_node->parent,
3104             hash_map_aligned_ptr_hash(old_node->parent));
3105         assert(slot);
3106         lookup = POINTER_SLOT_OF(slot);
3107         ps = ALLOC_NEW(Pointer_Slot);
3108         ps->key = old_node;
3109         ps->val = symbol_table_node_alloc_insert(lookup->val, table,
3110             old_node->type, old_node->node_rec);
3111         hash_map_insert(&hash_map, &ps->slot,
3112             hash_map_aligned_ptr_hash(ps->key));
3113         vector_append(needed_nodes, ps->key);
3114         return ps->val;
3115     } else {
3116         lookup = POINTER_SLOT_OF(slot);
3117         return lookup->val;
3118     }
3119 }
3120
3121 static HASH_MAP(type_hash, pointer_hash_map_compare);
3122
3123 static void type_hash_insert(void *key, void *val)
3124 {
3125     Pointer_Slot *ps = ALLOC_NEW(Pointer_Slot);
3126     ps->key = key;
3127     ps->val = val;
3128     hash_map_insert(&type_hash, &ps->slot, hash_map_aligned_ptr_hash(ps->key));
3129 }

```

```

3130 }
3131
3132 static void *type_hash_get(void *key)
3133 {
3134     Hash_Map_Slot *slot;
3135     slot = hash_map_get(&type_hash, key, hash_map_aligned_ptr_hash(key));
3136     if (!slot)
3137         return NULL;
3138     Pointer_Slot *ps = POINTER_SLOT_OF(slot);
3139     return ps->val;
3140 }
3141
3142 static void type_hash_clear()
3143 {
3144     hash_map_for_each_destroy(&type_hash, pointer_hash_map_destructor);
3145 }
3146
3147 Symbol_Type_Struct *symbol_table_insert_symbol_type(Symbol_Table *table,
3148     Symbol_Type_Struct *type_struct, Vector *needed_nodes,
3149     bool use_unique_name)
3150 {
3151     Vector *tmp_vec;
3152     Symbol_Type_Struct *ret = NULL, *tmp_type;
3153     Symbol_Type type = type_struct->methods->get_type();
3154     String tmp_str;
3155     Pointer_Slot *ps;
3156     Hash_Map_Slot *slot;
3157     switch (type) {
3158     case SYMBOL_TYPE_VOID:
3159         ret = symbol_type_void_alloc(table);
3160         break;
3161     case SYMBOL_TYPE_INT:
3162         ret = symbol_type_int_alloc(table);
3163         break;
3164     case SYMBOL_TYPE_BOOL:
3165         ret = symbol_type_bool_alloc(table);
3166         break;
3167     case SYMBOL_TYPE_CHAR:
3168         ret = symbol_type_char_alloc(table);
3169         break;
3170     case SYMBOL_TYPE_STRING:
3171         ret = symbol_type_string_alloc(table);
3172         break;
3173     case SYMBOL_TYPE_UNKNOWN:
3174         ret = symbol_type_unknown_alloc(table);
3175         break;
3176     case SYMBOL_TYPE_ARY:;
3177         Symbol_Type_Ary *old_ary, *new_ary;
3178         old_ary = SYMBOL_TYPE_STRUCT_CONTAINER(type_struct,
3179             Symbol_Type_Ary);
3180         if ((ret = type_hash_get(old_ary)))
3181             break;
3182         Symbol_Type_Struct *ary_type = symbol_table_insert_symbol_type(
3183             table, old_ary->ary_type, needed_nodes, use_unique_name);
3184         ret = symbol_type_ary_alloc(table, ary_type, old_ary->scope_id);
3185         new_ary = SYMBOL_TYPE_STRUCT_CONTAINER(ret, Symbol_Type_Ary);
3186         type_hash_insert(old_ary, new_ary);
3187         symbol_type_ary_set_name(new_ary, old_ary->ary_name);
3188         break;
3189     case SYMBOL_TYPE_REC:;
3190         Symbol_Type_Rec *old_rec, *new_rec;
3191         old_rec = SYMBOL_TYPE_STRUCT_CONTAINER(type_struct,
3192             Symbol_Type_Rec);
3193         if ((ret = type_hash_get(old_rec)))
3194             break;
3195         ret = symbol_type_rec_alloc(table, old_rec->scope_id,
3196             old_rec->unique_name);
3197         new_rec = SYMBOL_TYPE_STRUCT_CONTAINER(ret, Symbol_Type_Rec);
3198         type_hash_insert(old_rec, new_rec);
3199
3200         if (old_rec->rec_name) {
3201

```

```

3204         if (use_unique_name) {
3205             Const_String uname = SYMBOL_TYPE_REC_TYPE_NAME(old_rec);
3206             symbol_type_rec_set_name(new_rec, uname);
3207         } else {
3208             symbol_type_rec_set_name(new_rec, old_rec->rec_name);
3209         }
3210     }
3211
3212     new_rec->rec_sym_node = symbol_table_node_copy_hash_insert_node(
3213         table, old_rec->rec_sym_node, needed_nodes);
3214
3215     new_rec->super_fields_appended = old_rec->super_fields_appended;
3216
3217     new_rec->body_resolved = old_rec->body_resolved;
3218     new_rec->body_resolved_last = old_rec->body_resolved_last;
3219
3220     tmp_vec = &old_rec->extended_types;
3221     VECTOR_FOR_EACH_ENTRY(tmp_vec, tmp_type)
3222         vector_append(&new_rec->extended_types,
3223             symbol_table_insert_symbol_type(table, tmp_type,
3224                 needed_nodes, use_unique_name));
3225
3226     tmp_vec = &old_rec->func_identifiers;
3227     VECTOR_FOR_EACH_ENTRY(tmp_vec, tmp_str)
3228         symbol_type_rec_append_func_identifier(new_rec, tmp_str);
3229
3230     tmp_vec = &old_rec->var_identifiers;
3231     VECTOR_FOR_EACH_ENTRY(tmp_vec, tmp_str)
3232         symbol_type_rec_append_var_identifier(new_rec, tmp_str);
3233
3234     tmp_vec = &old_rec->var_types;
3235     VECTOR_FOR_EACH_ENTRY(tmp_vec, tmp_type)
3236         vector_append(&new_rec->var_types,
3237             symbol_table_insert_symbol_type(table, tmp_type,
3238                 needed_nodes, use_unique_name));
3239
3240     tmp_vec = &old_rec->func_types;
3241     VECTOR_FOR_EACH_ENTRY(tmp_vec, tmp_type)
3242         vector_append(&new_rec->func_types,
3243             symbol_table_insert_symbol_type(table, tmp_type,
3244                 needed_nodes, use_unique_name));
3245
3246     break;
3247
3248     case SYMBOL_TYPE_FUNC:;
3249     Symbol_Type_Func *new_func, *old_func;
3250     old_func = SYMBOL_TYPE_STRUCT_CONTAINER(type_struct,
3251         Symbol_Type_Func);
3252
3253     if ((ret = type_hash_get(old_func)))
3254         break;
3255
3256     ret = symbol_type_func_alloc(table, old_func->is_extern_c);
3257     new_func = SYMBOL_TYPE_STRUCT_CONTAINER(ret,
3258         Symbol_Type_Func);
3259
3260     type_hash_insert(old_func, new_func);
3261
3262     new_func->body_sym_node =
3263         symbol_table_node_copy_hash_insert_node(table,
3264             old_func->body_sym_node, needed_nodes);
3265
3266     new_func->is_resolved = old_func->is_resolved;
3267     new_func->is_concrete_func = old_func->is_concrete_func;
3268
3269     tmp_vec = &old_func->param_identifiers;
3270     VECTOR_FOR_EACH_ENTRY(tmp_vec, tmp_str)
3271         symbol_type_func_append_param_identifier(new_func, tmp_str);
3272
3273     tmp_vec = &old_func->param_types;
3274     VECTOR_FOR_EACH_ENTRY(tmp_vec, tmp_type)
3275         vector_append(&new_func->param_types,
3276             symbol_table_insert_symbol_type(table, tmp_type,
3277                 needed_nodes, use_unique_name));

```

```

3278         new_func->return_type = symbol_table_insert_symbol_type(table,
3279             old_func->return_type, needed_nodes, use_unique_name);
3280         break;
3281     case SYMBOL_TYPE_CYCLE:;
3282     assert(0);
3283     Symbol_Type_Cycle *old_cycle = SYMBOL_TYPE_STRUCT_CONTAINER(
3284         type_struct, Symbol_Type_Cycle);
3285     ret = symbol_type_cycle_alloc(table, old_cycle->name);
3286     Symbol_Type_Cycle *new_cycle = SYMBOL_TYPE_STRUCT_CONTAINER(ret,
3287         Symbol_Type_Cycle);
3288     new_cycle->sym_node = symbol_table_node_copy_hash_insert_node(
3289         table, old_cycle->sym_node, needed_nodes);
3290     new_cycle->scope_id = new_cycle->sym_node->scope_id;
3291     break;
3292     case SYMBOL_TYPE_IDEN:;
3293     Symbol_Type_Iden *new_iden;
3294     Symbol_Type_Iden *old_iden = SYMBOL_TYPE_STRUCT_CONTAINER(
3295         type_struct, Symbol_Type_Iden);
3296     ret = symbol_type_iden_alloc(table, old_iden->iden);
3297     new_iden = SYMBOL_TYPE_STRUCT_CONTAINER(ret, Symbol_Type_Iden);
3298     slot = hash_map_get(&hash_map, old_iden->sym_node,
3299         hash_map_aligned_ptr_hash(old_iden->sym_node));
3300     assert(slot);
3301     ps = POINTER_SLOT_OF(slot);
3302     new_iden->sym_node = ps->val;
3303     new_iden->loc = old_iden->loc;
3304     break;
3305     default:
3306     assert(false);
3307 }
3308 }
3309 return ret;
3310 }
3311
3312 bool symbol_table_node_insert_symbol(Symbol_Table *table,
3313     Symbol_Table_Node *node, Symbol *sym, bool use_unique_name,
3314     Symbol_Property property)
3315 {
3316     bool ret;
3317     Vector needed_nodes = VECTOR_INIT();
3318     ret = ____symbol_table_node_insert_symbol(table, node, sym, use_unique_name,
3319         property, &needed_nodes);
3320     symbol_table_node_copy_needed_nodes(table, &needed_nodes, use_unique_name);
3321     symbol_table_node_copy_hash_clear();
3322     vector_for_each_destroy(&needed_nodes, NULL);
3323     return ret;
3324 }
3325
3326 /* Tries to insert symbol, returns true on success otherwise */
3327 bool ____symbol_table_node_insert_symbol(Symbol_Table *table,
3328     Symbol_Table_Node *node, Symbol *sym, bool use_unique_name,
3329     Symbol_Property property, Vector *needed_nodes)
3330 {
3331     Uns hash;
3332     bool ret;
3333     Symbol_Type_Struct *type = NULL;
3334     String new_sym_name;
3335     Symbol *dup;
3336
3337     if (property == SYMBOL_PROPERTY_FUNC)
3338         new_sym_name = string_duplicate(STRING_AFTER_DOT(sym->identifier));
3339     else if (use_unique_name)
3340         new_sym_name = string_duplicate(sym->unique_name);
3341     else
3342         new_sym_name = string_duplicate(sym->identifier);
3343
3344     hash = string_hash_code(new_sym_name);
3345     if (hash_map_contains(&node->symbol_maps[property], new_sym_name, hash)
3346         && property == SYMBOL_PROPERTY_VAR) {
3347         ret = false;

```

```

3352     assert(false);
3353     goto out;
3354 }
3355
3356 if (property == SYMBOL_PROPERTY_TYPE_DEF) {
3357     Type_Def_Symbol *tsym = TYPE_DEF_SYMBOL_OF_SYMBOL(sym);
3358     Double_List *list = &tsym->dbnode;
3359     Double_List_Node *dbnode;
3360     DOUBLE_LIST_FOR_EACH(list, dbnode) {
3361         dup = (Symbol *) TYPE_DEF_SYMBOL_OF_DBNODE(dbnode);
3362         assert(dup->resolved_type);
3363         type = symbol_table_insert_symbol_type(table,
3364             dup->resolved_type, needed_nodes, use_unique_name);
3365
3366         __symbol_table_node_insert(node, new_sym_name, dup->unique_name,
3367             hash, type, property, dup->location);
3368     }
3369 }
3370
3371 assert(sym->resolved_type);
3372 type = symbol_table_insert_symbol_type(table, sym->resolved_type,
3373     needed_nodes, use_unique_name);
3374
3375 __symbol_table_node_insert(node, new_sym_name, sym->unique_name,
3376     hash, type, property, sym->location);
3377
3378 ret = true;
3379 out:
3380 string_destroy(new_sym_name);
3381 return ret;
3382 }
3383
3384 static void symbol_table_node_merge_remap(Symbol_Table_Node *src,
3385     Symbol_Table_Node *dest)
3386 {
3387     Symbol_Func_Remap *src_map;
3388     Symbol_Func_Remap *dest_map;
3389     Hash_Map_Slot *slot, *dest_slot;
3390     HASH_MAP_FOR_EACH(&src->func_remaps, slot) {
3391         src_map = SYMBOL_FUNC_REMAP_OF(slot);
3392         dest_slot = hash_map_get(&dest->func_remaps, src_map->initial_iden,
3393             hash_map_slot_get_hash_code(slot));
3394         if (!dest_slot) {
3395             DLOG("cpy remap %S => %S\n", src_map->initial_iden,
3396                 src_map->new_iden);
3397             dest_map = symbol_func_remap_alloc(src_map->initial_iden,
3398                 src_map->new_iden);
3399             hash_map_insert(&dest->func_remaps, &dest_map->hash_slot,
3400                 hash_map_slot_get_hash_code(slot));
3401         } else {
3402             DEBUGT(def,
3403                 dest_map = SYMBOL_FUNC_REMAP_OF(dest_slot);
3404                 assert(!string_compare(dest_map->new_iden, src_map->new_iden));
3405             );
3406         }
3407     }
3408 }
3409
3410 Symbol *__symbol_table_node_copy_func_symbols(Symbol_Table *table,
3411     Symbol_Table_Node *node1, Symbol_Table_Node *node2,
3412     bool use_unique_name, Symbol_Table_Node *cond, Vector *needed_nodes)
3413 {
3414     Symbol *sym = NULL, *sym2;
3415     Hash_Map_Slot *slot;
3416     Symbol_Func_Map *map, *map2;
3417     if (cond) {
3418         HASH_MAP_FOR_EACH(&cond->func_iden_map, slot) {
3419             map = SYMBOL_FUNC_MAP_OF(slot);
3420             VECTOR_FOR_EACH_ENTRY(&map->overload_idens, sym2) {
3421                 map2 = symbol_table_node_lookup_func_map(node2,
3422                     (String)STRING_AFTER_DOT(sym2->identifier));
3423                 assert(map2);
3424                 VECTOR_FOR_EACH_ENTRY(&map2->overload_idens, sym) {

```

```

3426         if (!file_location_cmp(sym->location, sym2->location))
3427             break;
3428         DEBUGT(def, sym = NULL);
3429     }
3430     assert(sym);
3431     if (!__symbol_table_node_insert_symbol(table, node1, sym,
3432         false, SYMBOL_PROPERTY_FUNC, needed_nodes))
3433         goto out;
3434     }
3435 }
3436 } else {
3437     HASH_MAP_FOR_EACH(&node2->func_iden_map, slot) {
3438         map = SYMBOL_FUNC_MAP_OF(slot);
3439         VECTOR_FOR_EACH_ENTRY(&map->overload_idens, sym) {
3440             if (!__symbol_table_node_insert_symbol(table, node1, sym,
3441                 use_unique_name, SYMBOL_PROPERTY_FUNC,
3442                 needed_nodes))
3443                 goto out;
3444         }
3445     }
3446 }
3447 symbol_table_node_merge_remap(node2, node1);
3448
3449 sym = NULL;
3450 out:
3451 return sym;
3452 }
3453
3454 Symbol *__symbol_table_node_copy_symbols(Symbol_Table *table,
3455     Symbol_Table_Node *node1, Symbol_Table_Node *node2,
3456     bool use_unique_name, Symbol_Table_Node *cond, Vector *needed_nodes,
3457     Symbol_Property property)
3458 {
3459     if (property == SYMBOL_PROPERTY_FUNC)
3460         return __symbol_table_node_copy_func_symbols(table, node1, node2,
3461             use_unique_name, cond, needed_nodes);
3462
3463     Symbol *sym = NULL, *tmp_sym;
3464     Hash_Map_Slot *slot;
3465     if (cond) {
3466         SYMBOL_TABLE_NODE_FOR_EACH_SYMBOL(cond, slot, tmp_sym, property) {
3467             DLOG("cpy_sym: %S\n", tmp_sym->unique_name);
3468             sym = symbol_table_node_lookup(node2,
3469                 tmp_sym->unique_name, property);
3470             assert(sym);
3471             if (!__symbol_table_node_insert_symbol(table, node1, sym, false,
3472                 property, needed_nodes))
3473                 goto out;
3474         }
3475     } else {
3476         SYMBOL_TABLE_NODE_FOR_EACH_SYMBOL(node2, slot, sym, property) {
3477             DLOG("cpy_sym: %S\n", sym->unique_name);
3478             if (!__symbol_table_node_insert_symbol(table, node1, sym,
3479                 use_unique_name, property, needed_nodes))
3480                 goto out;
3481         }
3482     }
3483     sym = NULL;
3484 out:
3485     return sym;
3486 }
3487
3488 Symbol *__symbol_table_node_copy_symbols(Symbol_Table *table,
3489     Symbol_Table_Node *node1, Symbol_Table_Node *node2,
3490     bool use_unique_name, Symbol_Table_Node *cond, Vector *needed_nodes)
3491 {
3492     bool has_finalize = node1->has_finalize_func || node2->has_finalize_func;
3493     node1->has_finalize_func = has_finalize;
3494     bool has_rec_func = node1->has_record_func || node2->has_record_func;
3495     node1->has_record_func = has_rec_func;
3496
3497     Symbol *sym;
3498     sym = __symbol_table_node_copy_symbols(table, node1, node2,
3499         use_unique_name, cond, needed_nodes,

```

```

3500         SYMBOL_PROPERTY_TYPE_DEF);
3501     assert(!sym);
3502     sym = __symbol_table_node_copy_symbols(table, node1, node2,
3503         use_unique_name, cond, needed_nodes,
3504         SYMBOL_PROPERTY_FUNC);
3505     assert(!sym);
3506     sym = __symbol_table_node_copy_symbols(table, node1, node2,
3507         use_unique_name, cond, needed_nodes,
3508         SYMBOL_PROPERTY_VAR);
3509     (void)sym;
3510     assert(!sym);
3511
3512     return NULL;
3513 }
3514
3515
3516
3517 Symbol *symbol_table_node_copy_needed_nodes(Symbol_Table *table,
3518     Vector *needed_nodes, bool use_unique_name UNUSED)
3519 {
3520     Symbol *sym = NULL;
3521     Symbol_Table_Node *node1, *node2;
3522     Hash_Map_Slot *tmp_slot;
3523
3524     while (vector_size(needed_nodes) > 0) {
3525         node2 = vector_pop_last(needed_nodes);
3526
3527         assert(hash_map_contains(&hash_map, node2->parent,
3528             hash_map_aligned_ptr_hash(node2->parent)));
3529
3530         tmp_slot = hash_map_get(&hash_map, node2,
3531             hash_map_aligned_ptr_hash(node2));
3532         assert(tmp_slot);
3533         node1 = POINTER_SLOT_OF(tmp_slot)->val;
3534
3535         if ((sym = symbol_table_node_copy_symbols(table, node1, node2,
3536             false, NULL, needed_nodes)))
3537             goto error_out;
3538     }
3539
3540 error_out:
3541     return sym;
3542 }
3543
3544 Symbol *symbol_table_node_copy_node(Symbol_Table *table,
3545     Symbol_Table_Node *node1, Symbol_Table_Node *node2,
3546     bool use_unique_name, Symbol_Table_Node *cond, Vector *needed_nodes)
3547 {
3548     Symbol *sym = NULL;
3549     sym = symbol_table_node_copy_symbols(table, node1, node2,
3550         use_unique_name, cond, needed_nodes);
3551     assert(!sym);
3552
3553     if (node1->type != SYMBOL_TABLE_NODE_IMPORT) {
3554         Vector *dep = &node2->import_dependencies;
3555         Vector *dep_loc = &node2->import_dependencies_loc;
3556         Uns size = vector_size(dep);
3557         for (Uns i = 0; i < size; i++) {
3558             vector_append(&node1->import_dependencies,
3559                 string_duplicate(vector_get(dep, i)));
3560             vector_append(&node1->import_dependencies_loc, vector_get(dep_loc, i));
3561         }
3562     }
3563
3564     sym = symbol_table_node_copy_needed_nodes(table, needed_nodes,
3565         use_unique_name);
3566
3567     assert(!sym);
3568
3569     return sym;
3570 }
3571
3572 /* n1 is the sym node n2 copied to */
3573 void symbol_table_node_copy_hash_link(Symbol_Table_Node *n1,

```

```

3574     Symbol_Table_Node *n2)
3575 {
3576     Pointer_Slot *ps = ALLOC_NEW(Pointer_Slot);
3577     ps->key = n2;
3578     ps->val = n1;
3579     hash_map_insert(&hash_map, &ps->slot, hash_map_aligned_ptr_hash(ps->key));
3580 }
3581
3582 void symbol_table_node_copy_hash_clear()
3583 {
3584     hash_map_for_each_destroy(&hash_map, pointer_hash_map_destructor);
3585 }
3586
3587 /*node1 must be a node in table1 and node2 must be a node in table2*/
3588 Symbol *symbol_table_node_merge_cond(
3589     Symbol_Table *table1, Symbol_Table_Node *n1,
3590     Symbol_Table *table2, Symbol_Table_Node *n2,
3591     bool use_unique_name, Symbol_Table_Node *cond)
3592 {
3593     Double_List *list;
3594     Double_List_Node *dbnode;
3595     Symbol_Table_Node *node1 = n1;
3596     Symbol_Table_Node *node2 = n2;
3597     Symbol *sym = NULL;
3598
3599     if (double_list_is_empty(&table2->all_nodes) && !node2)
3600         goto out;
3601     else if (!double_list_is_empty(&table2->all_nodes) && !node2)
3602         node2 = SYMBOL_TABLE_NODE_OF(
3603             double_list_peek_first(&table2->all_nodes));
3604
3605     if (double_list_is_empty(&table1->all_nodes) && !node1)
3606         node1 = symbol_table_node_alloc_insert(NULL, table1,
3607             SYMBOL_TABLE_NODE_GLOBAL, NULL);
3608     else if (!double_list_is_empty(&table2->all_nodes) && !node1)
3609         node1 = SYMBOL_TABLE_NODE_OF(
3610             double_list_peek_first(&table1->all_nodes));
3611     else if (double_list_is_empty(&table1->all_nodes) && node1) {
3612         assert(false);
3613         goto out;
3614     }
3615
3616     list = &table2->all_nodes;
3617     DOUBLE_LIST_FOR_EACH(list, dbnode) {
3618         if (SYMBOL_TABLE_NODE_OF(dbnode) == node2)
3619             goto node2_found;
3620     }
3621
3622     assert(false);
3623
3624     node2_found;
3625     Vector *needed_nodes = vector_alloc();
3626
3627     symbol_table_node_copy_hash_link(node1, node2);
3628
3629     sym = symbol_table_node_copy_node(table1, node1, node2,
3630         use_unique_name, cond, needed_nodes);
3631
3632     vector_destroy(needed_nodes, NULL);
3633 out:
3634     hash_map_for_each_destroy(&hash_map, pointer_hash_map_destructor);
3635     type_hash_clear();
3636     return sym;
3637 }
3638
3639 inline Symbol *symbol_table_node_merge(
3640     Symbol_Table *table1, Symbol_Table_Node *n1,
3641     Symbol_Table *table2, Symbol_Table_Node *n2,
3642     bool use_unique_name)
3643 {
3644     return symbol_table_node_merge_cond(table1, n1, table2, n2,
3645         use_unique_name, NULL);
3646 }
3647

```



```

3648  /* Merges two symbol tables.
3649  * The result of the merge is table1.
3650  * returns NULL on succes, the conflicting symbol on failure. */
3651  inline Symbol *symbol_table_merge(Symbol_Table *table1, Symbol_Table *table2,
3652      bool use_unique_name)
3653  {
3654      return symbol_table_node_merge(table1, NULL, table2, NULL,
3655          use_unique_name);
3656  }
3657
3658  void symbol_func_map_hash_destroy(Hash_Map_Slot *slot)
3659  {
3660      symbol_func_map_destroy(SYMBOL_FUNC_MAP_OF(slot));
3661  }
3662
3663  bool __symbol_func_map_comparator(String func_iden, Hash_Map_Slot *slot)
3664  {
3665      Symbol_Func_Map *m = SYMBOL_FUNC_MAP_OF(slot);
3666      return !string_compare(m->func_iden, func_iden);
3667  }
3668
3669  void symbol_func_remap_hash_destroy(Hash_Map_Slot *slot)
3670  {
3671      symbol_func_remap_destroy(SYMBOL_FUNC_REMAP_OF(slot));
3672  }
3673
3674  bool __symbol_func_remap_comparator(String func_iden, Hash_Map_Slot *slot)
3675  {
3676      Symbol_Func_Remap *r = SYMBOL_FUNC_REMAP_OF(slot);
3677      return !string_compare(r->initial_iden, func_iden);
3678  }

```

:

A.3.13 src/ast/symbol_table.h

```

1  #ifndef SYMBOL_TABLE_H
2  #define SYMBOL_TABLE_H
3
4  #include <hash_map.h>
5  #include <std_include.h>
6  #include <vector.h>
7  #include <debug.h>
8  #include <double_list.h>
9  #include <dot_printer.h>
10
11 #undef DEBUG_TYPE
12 #define DEBUG_TYPE symbol-table
13
14 typedef struct Symbol_Table_Node Symbol_Table_Node;
15
16 typedef struct Symbol_Table Symbol_Table;
17
18 typedef struct Symbol Symbol;
19
20 typedef struct Type_Def_Symbol Type_Def_Symbol;
21
22 typedef struct Symbol_Return Symbol_Return;
23
24 typedef struct Symbol_Type_Struct Symbol_Type_Struct;
25
26 typedef struct String_Builder String_Builder;
27
28 typedef struct Symbol_Type_String Symbol_Type_String;
29
30 void symbol_table_clear(Symbol_Table *t);
31
32 typedef enum Symbol_Property {
33     SYMBOL_PROPERTY_VAR,
34     SYMBOL_PROPERTY_FUNC,
35     SYMBOL_PROPERTY_TYPE_DEF // Must be last

```

```

36 } Symbol_Property;
37
38 #define SYMBOL_PROPERTY_COUNT (SYMBOL_PROPERTY_TYPE_DEF + 1)
39
40 typedef enum Symbol_Type {
41     SYMBOL_TYPE_VOID,      // Symbol has type void
42     SYMBOL_TYPE_INT,       // Symbol has type int
43     SYMBOL_TYPE_BOOL,      // Symbol has type bool
44     SYMBOL_TYPE_CHAR,      // Symbol has type char
45     SYMBOL_TYPE_STRING,    // Symbol has type string
46     SYMBOL_TYPE_ARY,       // Symbol has type array
47     SYMBOL_TYPE_REC,       // Symbol has type record
48     SYMBOL_TYPE_FUNC,      // Symbol has type function
49
50     /* This symbol type is used when a record or array cycle has been detected.
51      * An example of a record cycle is:
52      * type R = record of {r:R, ...};
53      * The following is also a cycle:
54      * type R = record of R {...};
55      * However this last one is an error and will cause R to become an
56      * unknown type instead of a record type. */
57     SYMBOL_TYPE_CYCLE,
58
59     /* Used internally to represent types which are not yet resolved.
60      * I.e. The type is defined by some identifier. Examples:
61      * type T = A; (here symbol T has type A which have not yet been resolved).
62      * var a:A; (here symbol a has type A which have not yet been resolved). */
63     SYMBOL_TYPE_IDEN,
64
65     /* Symbol has unknown type. This might be because a symbol was
66      * initially defined to be some identifier type which we were
67      * not able to resolve. Most likely because the user never
68      * defined the identifier type.
69      * A symbol will also have unknown type when the user has
70      * defined the same symbol with different types. */
71     SYMBOL_TYPE_UNKNOWN // Must be last
72 } Symbol_Type;
73
74 #define SYMBOL_RETURN_TYPE_COUNT (SYMBOL_RETURN_TYPE_UNKNOWN + 1)
75
76 struct Symbol {
77     String identifier;
78     String unique_name;
79     Symbol_Type_Struct *resolved_type;
80     File_Location *location;
81     Symbol_Table_Node *sym_node;
82     Hash_Map_Slot hash_slot;
83 };
84
85 inline static File_Location *symbol_get_location(Symbol *sym)
86 {
87     return sym->location;
88 }
89
90 #define SYMBOL_LOCATION(symbol_ptr) symbol_ptr->location
91
92 #define SYMBOL_OF_SLOT(slot) HASH_MAP_ENTRY(slot, Symbol, hash_slot)
93
94 struct Type_Def_Symbol {
95     Symbol symbol; // Must be first.
96     Double_List_Node dbnode;
97     bool cycle_marked;
98     bool report_cycle;
99 };
100
101 #define TYPE_DEF_SYMBOL_OF_SYMBOL(sym) ((Type_Def_Symbol *) (sym))
102
103 #define TYPE_DEF_SYMBOL_OF_DBNODE(node) \
104     CONTAINER_OF(node, Type_Def_Symbol, dbnode)
105
106 /*****
107
108 typedef struct Symbol_Type_Struct_Methods {
109     Symbol_Type (*get_type)();

```

```

110     Symbol_Type_Struct *(*resolve)(Symbol_Type_Struct *self, Symbol_Table *t);
111     void (*append_str)(Symbol_Type_Struct *self, String_Builder *sb);
112     void (*destroy)(Symbol_Type_Struct *self);
113     bool (*same_type)(Symbol_Type_Struct *self, Symbol_Type_Struct *oth);
114 } Symbol_Type_Struct_Methods;
115
116 struct Symbol_Type_Struct {
117     Symbol_Type_Struct_Methods *methods;
118     Double_List_Node dbnode;
119 };
120
121 #define SYMBOL_TYPE_STRUCT_FIELD sym_struct
122
123 #define SYMBOL_TYPE_STRUCT_BEGIN(name) \
124     typedef struct name { Symbol_Type_Struct SYMBOL_TYPE_STRUCT_FIELD;
125
126 #define SYMBOL_TYPE_STRUCT_END(name) } name;
127
128 #define SYMBOL_TYPE_STRUCT_CONTAINER(sym_struct, container_type) \
129     CONTAINER_OF(sym_struct, container_type, SYMBOL_TYPE_STRUCT_FIELD)
130
131 #define SYMBOL_TYPE_STRUCT_OF_CONTAINER(container) \
132     (&(container)->SYMBOL_TYPE_STRUCT_FIELD)
133
134 #define SYMBOL_TYPE_STRUCT_OF_DBNODE(node) \
135     DOUBLE_LIST_ENTRY(node, Symbol_Type_Struct, dbnode)
136
137 SYMBOL_TYPE_STRUCT_BEGIN(Symbol_Type_Void)
138 SYMBOL_TYPE_STRUCT_END(Symbol_Type_Void)
139
140 SYMBOL_TYPE_STRUCT_BEGIN(Symbol_Type_Int)
141 SYMBOL_TYPE_STRUCT_END(Symbol_Type_Int)
142
143 Symbol_Type_Struct *symbol_type_int_alloc(Symbol_Table *t);
144
145 Symbol_Type_Struct *symbol_type_void_alloc(Symbol_Table *t);
146
147 SYMBOL_TYPE_STRUCT_BEGIN(Symbol_Type_Bool)
148 SYMBOL_TYPE_STRUCT_END(Symbol_Type_Bool)
149
150 Symbol_Type_Struct *symbol_type_bool_alloc(Symbol_Table *t);
151
152 SYMBOL_TYPE_STRUCT_BEGIN(Symbol_Type_Char)
153 SYMBOL_TYPE_STRUCT_END(Symbol_Type_Char)
154
155 Symbol_Type_Struct *symbol_type_char_alloc(Symbol_Table *t);
156
157 SYMBOL_TYPE_STRUCT_BEGIN(Symbol_Type_String)
158 SYMBOL_TYPE_STRUCT_END(Symbol_Type_String)
159
160 Symbol_Type_Struct *symbol_type_string_alloc(Symbol_Table *t);
161
162 SYMBOL_TYPE_STRUCT_BEGIN(Symbol_Type_Ary)
163     Symbol_Type_Struct *ary_type;
164     String ary_name;
165     Uns scope_id;
166     struct {
167         bool dump_cycle_marked : 1;
168         bool imp_table_updated : 1;
169     };
170 SYMBOL_TYPE_STRUCT_END(Symbol_Type_Ary)
171
172 void symbol_type_ary_set_name(Symbol_Type_Ary *self, String name);
173
174 Symbol_Type_Struct *symbol_type_ary_alloc(Symbol_Table *t,
175     Symbol_Type_Struct *ary_type, Uns scope_id);
176
177 SYMBOL_TYPE_STRUCT_BEGIN(Symbol_Type_Iden)
178     String iden;
179     Symbol_Table_Node *sym_node;
180     File_Location *loc;
181 SYMBOL_TYPE_STRUCT_END(Symbol_Type_Iden)
182
183 Symbol_Type_Struct *symbol_type_iden_alloc(Symbol_Table *t, String iden);

```

```

184
185 typedef struct Symbol_Type_Rec_Ambiguous_Ref {
186     Vector sym_locations;
187     Const_String field_name;
188     Hash_Map_Slot hash_slot;
189 } Symbol_Type_Rec_Ambiguous_Ref;
190
191 #define SYMBOL_REC_AMBIGUOUS_REF_OF(slot) \
192     HASH_MAP_ENTRY(slot, Symbol_Type_Rec_Ambiguous_Ref, hash_slot)
193
194 void symbol_type_iden_set_iden(Symbol_Type_Iden *self, String str);
195
196 typedef enum Symbol_Rec_Cycle_Type {
197     SYMBOL_REC_CYCLE_MARK_NONE,
198     SYMBOL_REC_CYCLE_ALLOWED,
199     SYMBOL_REC_CYCLE_NOT_ALLOWED
200 } Symbol_Rec_Cycle_Type;
201
202 SYMBOL_TYPE_STRUCT_BEGIN(Symbol_Type_Rec)
203     Symbol_Table_Node *rec_sym_node;
204     Vector extended_types;
205     Vector func_identifiers;
206     Vector func_types;
207     Vector var_identifiers;
208     Vector var_types;
209     String rec_name;
210     String unique_name;
211     String missing_finalize_name;
212     String missing_record_func_name;
213     Hash_Map ambiguous_refs;
214     Uns scope_id;
215     Symbol_Rec_Cycle_Type cycle_mark;
216     struct {
217         bool body_resolved : 1;
218         bool body_resolved_last : 1;
219         bool append_str_cycle_marked : 1;
220         bool super_fields_appended : 1;
221         bool imp_table_updated : 1;
222         bool imp_table_name_updated : 1;
223     };
224 SYMBOL_TYPE_STRUCT_END(Symbol_Type_Rec)
225
226 #define SYMBOL_TYPE_REC_TYPE_NAME(rec) ({ \
227     const char *__cstr = string_to_cstr(rec->unique_name); \
228     S(__cstr + 1); \
229 })
230
231 static inline void __symbol_type_rec_reset_unique_name(Symbol_Type_Rec *rec,
232     String new_unique)
233 {
234     string_destroy(rec->unique_name);
235     rec->unique_name = new_unique;
236 }
237
238 static inline void symbol_type_rec_add_ambiguous_ref(
239     Symbol_Type_Rec *rec, Symbol_Type_Rec_Ambiguous_Ref *r)
240 {
241     Uns hash = string_hash_code(r->field_name);
242     hash_map_insert(&rec->ambiguous_refs, &r->hash_slot, hash);
243 }
244
245 static inline Symbol_Type_Rec_Ambiguous_Ref *symbol_type_rec_get_ambiguous_ref(
246     Symbol_Type_Rec *rec, String field_name)
247 {
248     Hash_Map_Slot *slot = hash_map_get(&rec->ambiguous_refs, field_name,
249         string_hash_code(field_name));
250     if (LIKELY(!slot))
251         return NULL;
252     return SYMBOL_REC_AMBIGUOUS_REF_OF(slot);
253 }
254
255 Symbol_Type_Struct *symbol_type_rec_alloc(Symbol_Table *t, Uns scope_id,
256     Const_String unique_name);
257

```

```

258 void symbol_type_rec_append_func_identifier(Symbol_Type_Rec *self,
259     Const_String iden);
260
261 void symbol_type_rec_append_var_identifier(Symbol_Type_Rec *self,
262     Const_String iden);
263
264 void symbol_type_rec_set_name(Symbol_Type_Rec *self, Const_String name);
265
266 Symbol_Type_Rec *symbol_type_rec_assignment_compatible(
267     Symbol_Type_Struct *dest, Symbol_Type_Struct *src);
268
269 static inline Symbol_Type_Rec *symbol_type_rec_cast_compatible(
270     Symbol_Type_Struct *lhs, Symbol_Type_Struct *rhs)
271 {
272     Symbol_Type_Rec *ret = symbol_type_rec_assignment_compatible(lhs, rhs);
273     if (ret)
274         goto out;
275     ret = symbol_type_rec_assignment_compatible(rhs, lhs);
276 out:
277     return ret;
278 }
279
280 bool __symbol_type_rec_ambiguous_cast(Symbol_Type_Rec *cast,
281     Symbol_Type_Rec *rec);
282
283 static inline bool symbol_type_rec_ambiguous_cast(Symbol_Type_Struct *cast,
284     Symbol_Type_Struct *rec)
285 {
286     Symbol_Type_Rec *lhs = SYMBOL_TYPE_STRUCT_CONTAINER(cast, Symbol_Type_Rec);
287     Symbol_Type_Rec *rhs = SYMBOL_TYPE_STRUCT_CONTAINER(rec, Symbol_Type_Rec);
288     return __symbol_type_rec_ambiguous_cast(lhs, rhs);
289 }
290
291 SYMBOL_TYPE_STRUCT_BEGIN(Symbol_Type_Cycle)
292     String name;
293     Uns scope_id;
294     Symbol_Table_Node *sym_node;
295 SYMBOL_TYPE_STRUCT_END(Symbol_Type_Cycle)
296
297 Symbol_Type_Struct *symbol_type_cycle_alloc(Symbol_Table *t, String name);
298
299 SYMBOL_TYPE_STRUCT_BEGIN(Symbol_Type_Unknown)
300 SYMBOL_TYPE_STRUCT_END(Symbol_Type_Unknown)
301
302 Symbol_Type_Struct *symbol_type_unknown_alloc(Symbol_Table *t);
303
304 SYMBOL_TYPE_STRUCT_BEGIN(Symbol_Type_Func)
305     Symbol_Table_Node *body_sym_node;
306     Symbol_Type_Struct *return_type;
307     Vector param_identifiers;
308     Vector param_types;
309     struct {
310         bool is_resolved : 1;
311         bool compare_return_type : 1;
312         bool is_extern_c : 1;
313         bool main_err_int_reported : 1;
314         bool main_param_err_reported : 1;
315         bool imp_table_updated : 1;
316         bool is_concrete_func : 1;
317     };
318 SYMBOL_TYPE_STRUCT_END(Symbol_Type_Func)
319
320 typedef struct Symbol_Func_Map {
321     Symbol_Table_Node *sym_node;
322     String func_iden;
323     Int unnamed_type_count;
324     Vector overload_idens;
325     Hash_Map_Slot hash_slot;
326     bool is_verified;
327     bool has_extern_c;
328 } Symbol_Func_Map;
329
330 static inline Symbol_Func_Map *symbol_func_map_alloc(Const_String func_iden,
331     Symbol_Table_Node *sym_node)

```

```

332 {
333     Symbol_Func_Map *ret = ALLOC_NEW(Symbol_Func_Map);
334     ret->func_iden = string_duplicate(func_iden);
335     ret->overload_idens = VECTOR_INIT();
336     ret->sym_node = sym_node;
337     ret->unnamed_type_count = 0;
338     ret->is_verified = false;
339     ret->has_extern_c = false;
340     return ret;
341 }
342
343 static inline bool string_is_ctor(Const_String s)
344 {
345     return !string_compare(STRING_AFTER_DOT(s), S("record"));
346 }
347
348 static inline bool symbol_func_maps_ctor(Symbol_Func_Map *map)
349 {
350     return string_is_ctor(map->func_iden);
351 }
352
353 static inline void symbol_func_map_destroy(Symbol_Func_Map *fmap)
354 {
355     string_destroy(fmap->func_iden);
356     vector_clear(&fmap->overload_idens);
357     free_mem(fmap);
358 }
359
360 typedef struct Symbol_Func_Remap {
361     String initial_iden;
362     String new_iden;
363     Hash_Map_Slot hash_slot;
364 } Symbol_Func_Remap;
365
366 #define SYMBOL_FUNC_REMAP_OF(slot) \
367     HASH_MAP_ENTRY(slot, Symbol_Func_Remap, hash_slot)
368
369 static inline Symbol_Func_Remap *symbol_func_remap_alloc(
370     Const_String initial_iden, Const_String new_iden)
371 {
372     Symbol_Func_Remap *ret = ALLOC_NEW(Symbol_Func_Remap);
373     ret->initial_iden = string_duplicate(initial_iden);
374     ret->new_iden = string_duplicate(new_iden);
375     return ret;
376 }
377
378 static inline void symbol_func_remap_destroy(Symbol_Func_Remap *r)
379 {
380     string_destroy(r->initial_iden);
381     string_destroy(r->new_iden);
382     free_mem(r);
383 }
384
385 void symbol_func_remap_hash_destroy(Hash_Map_Slot *slot);
386
387 void symbol_func_map_hash_destroy(Hash_Map_Slot *slot);
388
389 #define SYMBOL_FUNC_MAP_OF(slot) \
390     HASH_MAP_ENTRY(slot, Symbol_Func_Map, hash_slot)
391
392 void symbol_type_func_append_param_identifier(Symbol_Type_Func *self,
393     String iden);
394
395 Symbol_Type_Struct *symbol_type_func_alloc(Symbol_Table *t, bool is_extern_c);
396
397 /*****
398
399 typedef enum Symbol_Rec_Comp {
400     REC_COMP_DIFFERENT,
401     REC_COMP_SAME,
402     REC_COMP_IN_PROGRESS
403 } Symbol_Rec_Comp;
404
405 /* Inserted in Symbol_Table hash map when 2 records (rec1, and rec2)

```

```

406  * have been compared, or are in progress of being compared. */
407  typedef struct Symbol_Rec_Comp_Struct {
408      Hash_Map_Slot hash_slot;
409      Symbol_Type_Rec *rec1, *rec2;
410      Symbol_Rec_Comp comp_result;
411  } ALIGNED(PTR_SIZE) Symbol_Rec_Comp_Struct;
412
413  static inline Uns symbol_rec_comp_struct_hash_code(Symbol_Rec_Comp_Struct *c)
414  {
415      return hash_map_aligned_ptr_hash(c->rec1) +
416             hash_map_aligned_ptr_hash(c->rec2);
417  }
418
419  static inline void symbol_rec_comp_struct_set(Symbol_Rec_Comp_Struct *c,
420      Symbol_Rec_Comp comp_res)
421  {
422      c->comp_result = comp_res;
423  }
424
425  static inline Symbol_Rec_Comp symbol_rec_comp_struct_get(
426      Symbol_Rec_Comp_Struct *c)
427  {
428      return c->comp_result;
429  }
430
431  static inline Symbol_Rec_Comp_Struct *__symbol_rec_comp_struct_alloc(
432      Symbol_Type_Rec *rec1, Symbol_Type_Rec *rec2)
433  {
434      Symbol_Rec_Comp_Struct *c = ALLOC_NEW(Symbol_Rec_Comp_Struct);
435      c->rec1 = rec1;
436      c->rec2 = rec2;
437      return c;
438  }
439
440  /* Initially the comp_result is initialized to REC_COMP_IN_PROGRESS. */
441  static inline Symbol_Rec_Comp_Struct *symbol_rec_comp_struct_alloc(
442      Symbol_Type_Rec *rec1, Symbol_Type_Rec *rec2)
443  {
444      Symbol_Rec_Comp_Struct *c = __symbol_rec_comp_struct_alloc(rec1, rec2);
445      c->comp_result = REC_COMP_IN_PROGRESS;
446      return c;
447  }
448
449  #define SYMBOL_REC_COMP_STRUCT_OF(slot) \
450      HASH_MAP_ENTRY(slot, Symbol_Rec_Comp_Struct, hash_slot)
451
452  bool symbol_rec_comp_struct_compare(Symbol_Rec_Comp_Struct *search_comp,
453      Hash_Map_Slot *map_slot);
454
455  void symbol_rec_comp_struct_destroy(Hash_Map_Slot *slot);
456
457  struct Symbol_Table {
458      Symbol *current_symbol;
459      Symbol_Table_Node *current_sym_table_node;
460      Symbol_Type_Struct *symbol_type_struct_void;
461      Symbol_Type_Struct *symbol_type_struct_int;
462      Symbol_Type_Struct *symbol_type_struct_bool;
463      Symbol_Type_Struct *symbol_type_struct_char;
464      Symbol_Type_Struct *symbol_type_struct_string;
465      Symbol_Type_Struct *symbol_type_struct_unknown;
466      File_Location null_location;
467      Vector location_names;
468      Double_List all_nodes;
469      Double_List all_symbol_types;
470      Hash_Map record_comparisons;
471      Symbol_Property current_resolve_property;
472      Uns next_node_id;
473      Symbol_Rec_Cycle_Type rec_cycle_type;
474      bool last_resolve_pass;
475  };
476
477  static inline void symbol_table_insert_comp_result(Hash_Map *m,
478      Symbol_Rec_Comp_Struct *c)
479  {

```

```

480     Uns hash = symbol_rec_comp_struct_hash_code(c);
481     assert(!hash_map_contains(m, c, hash));
482     hash_map_insert(m, &c->hash_slot, hash);
483 }
484
485 static inline Symbol_Rec_Comp_Struct *symbol_table_get_comp_result(Hash_Map *m,
486     Symbol_Rec_Comp_Struct *search_struct)
487 {
488     Hash_Map_Slot *slot = hash_map_get(m, search_struct,
489         symbol_rec_comp_struct_hash_code(search_struct));
490     if (!slot)
491         return NULL;
492     return SYMBOL_REC_COMP_STRUCT_OF(slot);
493 }
494
495 #define SYMBOL_TABLE_INIT(name, curr_file) \
496     ((Symbol_Table)SYMBOL_TABLE_STATIC_INIT(name, curr_file))
497
498 #define SYMBOL_TABLE_STATIC_INIT(name, curr_file) { \
499     .all_nodes = DOUBLE_LIST_STATIC_INIT((name).all_nodes), \
500     .all_symbol_types = DOUBLE_LIST_STATIC_INIT((name).all_symbol_types), \
501     .next_node_id = 0, \
502     .last_resolve_pass = false, \
503     .location_names = VECTOR_STATIC_INIT(), \
504     .record_comparisons = HASH_MAP_STATIC_INIT( \
505         (Hash_Map_Comparator)symbol_rec_comp_struct_compare), \
506     .rec_cycle_type = SYMBOL_REC_CYCLE_MARK_NONE, \
507     .null_location = FILE_LOCATION_STATIC_INIT(curr_file, 0, 0) \
508 }
509
510 inline static Symbol_Table *symbol_table_alloc(Const_String curr_file_name)
511 {
512     Symbol_Table *ret = ALLOC_NEW(Symbol_Table);
513     *ret = SYMBOL_TABLE_INIT(*ret, curr_file_name);
514     return ret;
515 }
516
517 inline static void symbol_table_destroy(Symbol_Table *table)
518 {
519     symbol_table_clear(table);
520     free_mem(table);
521 }
522
523 typedef enum Symbol_Table_Node_Type {
524     SYMBOL_TABLE_NODE_GLOBAL,
525     SYMBOL_TABLE_NODE_REC,
526     SYMBOL_TABLE_NODE_FUNC,
527     SYMBOL_TABLE_NODE_INTERMEDIATE,
528     SYMBOL_TABLE_NODE_IMPORT
529 } Symbol_Table_Node_Type;
530
531 struct Symbol_Table_Node {
532     Double_List_Node dbnode;
533     Symbol_Table_Node *parent;
534     Symbol_Type_Rec *node_rec;
535     Hash_Map *record_comparisons;
536     Hash_Map symbol_maps[SYMBOL_PROPERTY_COUNT];
537     Hash_Map func_iden_map;
538     Hash_Map func_remaps;
539     Vector import_dependencies;
540     Vector import_dependencies_loc;
541     Uns scope_id;
542     Symbol_Table_Node_Type type;
543     Uns finalize_func_count;
544     bool has_finalize_func;
545     bool has_record_func;
546 };
547
548 inline static Uns symbol_table_node_get_scope_id(Symbol_Table_Node *node)
549 {
550     return node->scope_id;
551 }
552
553 #define SYMBOL_TABLE_FOR_EACH_NODE(table, node, dbnode) \

```



```

554     DOUBLE_LIST_FOR_EACH(&((table)->all_nodes), dbnode) \
555     if ((node = SYMBOL_TABLE_NODE_OF(dbnode)))
556
557 #define SYMBOL_TABLE_FOR_EACH_NODE_REVERSED(table, node, dbnode) \
558     DOUBLE_LIST_FOR_EACH_REVERSED(&((table)->all_nodes), dbnode) \
559     if ((node = SYMBOL_TABLE_NODE_OF(dbnode)))
560
561 #define SYMBOL_TABLE_NODE_FOR_EACH_SYMBOL(node, slot, sym, property) \
562     HASH_MAP_FOR_EACH(&((node)->symbol_maps[property]), slot) \
563     if ((sym = SYMBOL_OF_SLOT(slot)))
564
565 static inline Symbol_Table_Node_Type symbol_get_symbol_table_node_type(
566     Symbol *sym)
567 {
568     return sym->sym_node->type;
569 }
570
571 static inline Symbol_Func_Map *symbol_table_node_get_func_map(
572     Symbol_Table_Node *node, String iden, Uns hash_code)
573 {
574     Hash_Map_Slot *slot = hash_map_get(&node->func_iden_map, iden, hash_code);
575     if (!slot)
576         return NULL;
577     return SYMBOL_FUNC_MAP_OF(slot);
578 }
579
580 static inline Symbol *symbol_table_node_get(Symbol_Table_Node *node,
581     String iden, Uns hash_code, Symbol_Property property)
582 {
583     Hash_Map_Slot *slot;
584     if (property == SYMBOL_PROPERTY_FUNC) {
585         slot = hash_map_get(&node->func_remaps, iden, hash_code);
586         if (slot) {
587             Symbol_Func_Remap *remap = SYMBOL_FUNC_REMAP_OF(slot);
588             return symbol_table_node_get(node,
589                 remap->new_iden,
590                 string_hash_code(remap->new_iden),
591                 SYMBOL_PROPERTY_FUNC);
592         }
593     }
594     slot = hash_map_get(&node->symbol_maps[property], iden, hash_code);
595     if (!slot)
596         return NULL;
597     return SYMBOL_OF_SLOT(slot);
598 }
599
600 static inline Type_Def_Symbol *symbol_table_node_get_type_def(
601     Symbol_Table_Node *node, String iden, Uns hash_code,
602     Symbol_Property property)
603 {
604     Symbol *sym = symbol_table_node_get(node, iden, hash_code, property);
605     if (sym)
606         return TYPE_DEF_SYMBOL_OF_SYMBOL(sym);
607     return NULL;
608 }
609
610 Symbol *__symbol_table_node_lookup(Symbol_Table_Node *sym_node,
611     String iden, Uns hash_code, Symbol_Property property);
612
613 static inline Symbol *symbol_table_node_lookup(Symbol_Table_Node *sym_node,
614     String iden, Symbol_Property property)
615 {
616     Uns hash = string_hash_code(iden);
617     return __symbol_table_node_lookup(sym_node, iden, hash, property);
618 }
619
620 bool __symbol_var_is_in_scope(File_Location *var_loc,
621     Symbol *lookup_sym, Symbol_Table_Node *child_sym_node);
622
623 static inline Symbol *symbol_table_node_lookup_var(
624     Symbol_Table_Node *sym_node,
625     String iden,
626     File_Location *var_loc)
627 {

```

```

628     Symbol_Table_Node *child_sym_node = sym_node;
629
630     Symbol *ret;
631
632     Uns hash = string_hash_code(iden);
633     for (;;) {
634         ret = __symbol_table_node_lookup(sym_node, iden, hash,
635             SYMBOL_PROPERTY_VAR);
636         if (!ret || __symbol_var_is_in_scope(var_loc, ret, child_sym_node))
637             break;
638         sym_node = ret->sym_node->parent;
639         if (!sym_node) {
640             ret = NULL;
641             break;
642         }
643     }
644
645     return ret;
646 }
647
648 Symbol_Func_Map *__symbol_table_node_lookup_func_map(
649     Symbol_Table_Node *sym_node, String iden, Uns hash_code);
650
651 static inline Symbol_Func_Map *symbol_table_node_lookup_func_map(
652     Symbol_Table_Node *sym_node, String iden)
653 {
654     Uns hash = string_hash_code(iden);
655     return __symbol_table_node_lookup_func_map(sym_node, iden, hash);
656 }
657
658 void __symbol_table_node_insert(Symbol_Table_Node *node,
659     Const_String iden, Const_String unique_name, Uns hash_code,
660     Symbol_Type_Struct *type, Symbol_Property property,
661     File_Location *loc);
662
663 /* Warning. Assumes that iden is not in the symbol table node
664  * with property 'property'. */
665 void symbol_table_node_insert(Symbol_Table_Node *node, Const_String iden,
666     Const_String unique_name, Symbol_Type_Struct *type,
667     Symbol_Property property, File_Location *loc);
668
669 /* Warning. Assumes that iden is not in the symbol table node
670  * with property 'property'. */
671 static inline void symbol_table_node_insert_unknown(Symbol_Table_Node *node,
672     Symbol_Table *t, String iden, Symbol_Property property,
673     File_Location *loc)
674 {
675     Uns hash = string_hash_code(iden);
676     if (!symbol_table_node_get(node, iden, hash, property)) {
677         __symbol_table_node_insert(node, iden, NULL, hash,
678             symbol_type_unknown_alloc(t), property, loc);
679     }
680 }
681
682 static inline void symbol_table_node_insert_unknown_var(
683     Symbol_Table_Node *node, Symbol_Table *t, String iden,
684     File_Location *loc)
685 {
686     symbol_table_node_insert_unknown(node, t, iden,
687         SYMBOL_PROPERTY_VAR, loc);
688 }
689
690 static inline void symbol_table_node_insert_unknown_func(
691     Symbol_Table_Node *node, Symbol_Table *t, String iden,
692     File_Location *loc)
693 {
694     Uns hash = string_hash_code(iden);
695     if (!symbol_table_node_get(node, iden, hash, SYMBOL_PROPERTY_FUNC)) {
696         __symbol_table_node_insert(node, iden, iden, hash,
697             symbol_type_unknown_alloc(t), SYMBOL_PROPERTY_FUNC, loc);
698     }
699 }
700
701 bool __symbol_map_comparator(String search_iden, Hash_Map_Slot *slot);

```

```

702
703 bool ____symbol_func_map_comparator(String func_iden, Hash_Map_Slot *slot);
704
705 bool ____symbol_func_remap_comparator(String func_iden, Hash_Map_Slot *slot);
706
707 static inline Symbol_Table_Node *____symbol_table_node_alloc(Symbol_Table *t,
708     Symbol_Table_Node_Type type, Symbol_Type_Rec *node_rec)
709 {
710     Symbol_Table_Node *n = ALLOC_NEW(Symbol_Table_Node);
711     n->record_comparisons = &t->record_comparisons;
712     n->import_dependencies = VECTOR_INIT_SIZE(4);
713     n->import_dependencies_loc = VECTOR_INIT_SIZE(4);
714     for (Uns i = 0; i < SYMBOL_PROPERTY_COUNT; i++)
715         n->symbol_maps[i] = HASH_MAP_INIT_SIZE(HASH_MAP_SIZE_11,
716             (Hash_Map_Comparator)____symbol_map_comparator);
717     n->func_iden_map = HASH_MAP_INIT_SIZE(HASH_MAP_SIZE_11,
718         (Hash_Map_Comparator)____symbol_func_map_comparator);
719     n->func_remaps = HASH_MAP_INIT_SIZE(HASH_MAP_SIZE_11,
720         (Hash_Map_Comparator)____symbol_func_remap_comparator);
721     n->type = type;
722     n->finalize_func_count = 0;
723     n->has_finalize_func = false;
724     n->has_record_func = false;
725     n->node_rec = node_rec;
726     DEBUGT(def,
727         if (type == SYMBOL_TABLE_NODE_REC) {
728             assert(node_rec);
729         } else {
730             assert(!node_rec);
731         }
732     );
733
734     return n;
735 }
736
737 static inline Symbol_Table_Node *symbol_table_node_alloc_insert(
738     Symbol_Table_Node *parent, Symbol_Table *t,
739     Symbol_Table_Node_Type type, Symbol_Type_Rec *node_rec)
740 {
741     Symbol_Table_Node *n = ____symbol_table_node_alloc(t, type, node_rec);
742     n->parent = parent;
743     n->scope_id = t->next_node_id++;
744     double_list_append(&t->all_nodes, &n->dbnode);
745     return n;
746 }
747
748 void ____symbol_table_insert_location(Symbol_Table *t, Symbol_Table_Node *node,
749     Const_String prefix, Symbol_Type_Struct *type, File_Location *loc,
750     Symbol_Property property);
751
752 void symbol_table_insert_location(Symbol_Table *t, Symbol_Table_Node *node,
753     Const_String prefix, Symbol_Type_Struct *type, File_Location *loc);
754
755 Symbol *symbol_table_get_from_location(Symbol_Table_Node *node,
756     File_Location *loc);
757
758 void symbol_table_dump(Symbol_Table *t, Const_String prefix);
759
760 void symbol_table_dump_graph(Symbol_Table *t, Const_String file_prefix);
761
762 void symbol_table_resolve(Symbol_Table *t);
763
764 void symbol_table_node_copy_hash_link(Symbol_Table_Node *n1,
765     Symbol_Table_Node *n2);
766
767 void symbol_table_node_copy_hash_clear();
768
769 Symbol *symbol_table_node_copy_needed_nodes(Symbol_Table *table,
770     Vector *needed_nodes, bool use_unique_name);
771
772 bool symbol_table_node_insert_symbol(Symbol_Table *table,
773     Symbol_Table_Node *node, Symbol *sym, bool use_unique_name,
774     Symbol_Property property);
775

```

```

776 bool ____symbol_table_node_insert_symbol(Symbol_Table *table,
777     Symbol_Table_Node *node, Symbol *sym, bool use_unique_name,
778     Symbol_Property property, Vector *needed_nodes);
779
780 Symbol *symbol_table_merge(Symbol_Table *table1, Symbol_Table *table2,
781     bool use_unique_name);
782
783 Symbol *symbol_table_node_merge(
784     Symbol_Table *table1, Symbol_Table_Node *n1,
785     Symbol_Table *table2, Symbol_Table_Node *n2,
786     bool use_unique_name);
787
788 Symbol *symbol_table_node_merge_cond(
789     Symbol_Table *table1, Symbol_Table_Node *n1,
790     Symbol_Table *table2, Symbol_Table_Node *n2,
791     bool use_unique_name, Symbol_Table_Node *cond);
792
793 #define SYMBOL_TABLE_NODE_OF(node) \
794     DOUBLE_LIST_ENTRY(node, Symbol_Table_Node, dbnode)
795
796 inline static Symbol_Table_Node *symbol_table_get_root(Symbol_Table *t)
797 {
798     Symbol_Table_Node *node = SYMBOL_TABLE_NODE_OF(
799         double_list_peek_first(&t->all_nodes));
800     return node;
801 }
802
803 #undef DEBUG_TYPE
804 #define DEBUG_TYPE def
805
806 #endif // SYMBOL_TABLE_H

```

:

A.3.14 src/parser.c

```

1  #include <parser.h>
2  #include <main.h>
3  #include <std_include.h>
4
5  extern FILE *yyin;
6
7  void scanner_finalize();
8  void parser_finalize();
9  Ast *parser_parse(Const_String file);
10
11 Ast *parse(Const_String file)
12 {
13     Ast *ast;
14     yyin = file_open(file, S("r"));
15     if (!yyin) {
16         report_error(file, S("cannot open file [%m]\n"), file);
17         ast = NULL;
18     } else {
19         ast = parser_parse(file);
20         file_close(yyin);
21     }
22     return ast;
23 }
24
25 void parse_cleanup()
26 {
27     parser_finalize();
28     scanner_finalize();
29 }

```

:

A.3.15 src/parser.h

```

1  #ifndef PARSER_H
2  #define PARSER_H
3
4  #include <std_define.h>
5  #include <ast/ast.h>
6
7  typedef enum yytokentype Parser-Token_Type;
8
9  /* Parse the file with name 'current_file_name' and return root of
10 * the parse tree. If NULL then the parse tree could not be created
11 * and parsing failed. */
12 Ast *parse(Const_String file);
13
14 /* Free memory used by parser and scanner. Call this function once
15 * when parsing of all input files are done. */
16 void parse_cleanup();
17
18 void parser_report_error(Uns line, Uns column, Const_String format, ...);
19
20 #endif // PARSER_H

```

A.3.16 src/parser/parser.y

```

1  %{
2
3  #include <parser.h>
4  #include <std_include.h>
5  #include <vector.h>
6  #include <debug.h>
7  #include <main.h>
8  #include <ast/ast_visitor_delete.h>
9
10 int yylex();
11 int yyparse();
12 void scanner_restart();
13 void scanner_push_back_token();
14
15 static Uns func_nest;
16 static Uns curl_nest;
17 static VECTOR(curl_nest_stack);
18
19 static bool same_as_prev_node(Uns line, Uns col);
20 static void func_head_recover();
21 static void extern_func_recover();
22 static void decl_stmt_recover();
23 static void record_body_recover();
24
25 static Uns last_err_line;
26 static Uns last_err_column;
27 static Const_String last_err_text;
28
29 static VECTOR(decl_stmt_stack);
30 static Uns consistent_stmt_decl_top;
31
32 static VECTOR(extend_stack);
33 static VECTOR(extend_nest_stack);
34 static Uns extend_nest;
35
36 static VECTOR(func_head_stack);
37
38 static VECTOR(expr_stack);
39
40 static Ast *current_ast;
41
42 static Const_String current_file_name;
43
44 void yyerror(const char *str);

```

```

45
46 #define EXPECTED_STRING_START '@'
47 #define EXPECTED_STRING_END '#'
48 static STRING_EMPTY(expect_string);
49
50 #define GET_UNEXPEC_STRING(unexpec) ({ \
51     Const_String __ret; \
52     if (string_is_empty(unexpec)) \
53         __ret = S("end of input"); \
54     else \
55         __ret = unexpec; \
56     __ret; \
57 })
58
59 #define RECOVER(func, line, col, format, ...) \
60 do { \
61     if (!same_as_prev_node(line, col)) { \
62         parser_report_error(line, col, \
63             S(format), ## __VA_ARGS__); \
64         func(); \
65     } \
66     yyerrok; \
67     yyclearin; \
68 } while (false)
69
70 #define RECOVER_UNEXPEC(func, line, col, unexpec) \
71 do { \
72     RECOVER(func, line, col, "unexpected " QFY("%S") "\n", \
73         GET_UNEXPEC_STRING(unexpec)); \
74 } while (false)
75
76 #define RECOVER_EXPECT(func, line, col, unexpec) \
77 do { \
78     if (!string_is_empty(expect_string)) { \
79         RECOVER(func, line, col, "unexpected " QFY("%S") ", expected %S\n", \
80             GET_UNEXPEC_STRING(unexpec), expect_string); \
81     } else { \
82         RECOVER_UNEXPEC(func, line, col, unexpec); \
83     } \
84 } while (false)
85
86 #define RECOVER_EXPECT_SCOLON(func, line, col, unexpec) \
87 do { \
88     string_assign(expect_string, S(QFY(";"))); \
89     RECOVER_EXPECT(func, line, col, unexpec); \
90 } while (false)
91
92 #undef DEBUG_TYPE
93 #define DEBUG_TYPE parser-recover:parser-parse
94
95 void parser_report_error(Uns line, Uns column, Const_String format, ...)
96 {
97     VA_SETUP(vl, format);
98     File_Location loc = FILE_LOCATION_INIT(current_ast->file, line, column);
99     report_vaerror_location(&loc, format, vl);
100     VA_END(vl);
101 }
102
103 static inline void push_curl_nest()
104 {
105     DLOG("PARSER PUSH CURL NEST\n");
106     vector_append(&curl_nest_stack, INT_TO_PTR(curl_nest));
107     curl_nest = 0;
108 }
109
110 static inline void pop_curl_nest()
111 {
112     if (!vector_is_empty(&curl_nest_stack)) {
113         DLOG("PARSER POP CURL NEST\n");
114         curl_nest = PTR_TO_INT(vector_pop_last(&curl_nest_stack));
115     }
116 }
117
118 static inline void inc_curl_nest()

```

```

119 {
120     DLOG("PARSER INC CURL NEST\n");
121     ++curl_nest;
122 }
123
124 static inline void dec_curl_nest()
125 {
126     if (curl_nest) {
127         DLOG("PARSER DEC CURL NEST\n");
128         --curl_nest;
129     }
130 }
131
132 static inline Uns get_curl_nest()
133 {
134     return curl_nest;
135 }
136
137 static inline void inc_func_nest()
138 {
139     ++func_nest;
140 }
141
142 static inline void dec_func_nest()
143 {
144     if (func_nest)
145         --func_nest;
146 }
147
148 static inline void enter_func()
149 {
150     inc_func_nest();
151     push_curl_nest();
152 }
153
154 static inline void leave_func()
155 {
156     dec_func_nest();
157     pop_curl_nest();
158 }
159
160 static inline Uns get_func_nest()
161 {
162     return func_nest;
163 }
164
165 static inline void inc_extend_nest()
166 {
167     ++extend_nest;
168 }
169
170 static inline void dec_extend_nest()
171 {
172     assert(extend_nest);
173     --extend_nest;
174 }
175
176 static inline Uns get_extend_nest()
177 {
178     return extend_nest;
179 }
180
181 static inline void push_extend_nest()
182 {
183     vector_append(&extend_nest_stack, INT_TO_PTR(extend_nest));
184     extend_nest = 0;
185 }
186
187 static inline void pop_extend_nest()
188 {
189     assert(!vector_is_empty(&extend_nest_stack));
190     extend_nest = PTR_TO_INT(vector_pop_last(&extend_nest_stack));
191 }
192

```

```

193 static void clear_vector_stack(Vector *stack)
194 {
195     Ast_Node *n;
196     Vector *v;
197     VECTOR_FOR_EACH_ENTRY(stack, v) {
198         VECTOR_FOR_EACH_ENTRY(v, n)
199             ast_visitor_delete_accept_visitor(n);
200         vector_destroy(v, NULL);
201     }
202     vector_clear(stack);
203 }
204
205 static inline Uns vector_stack_size(Vector *stack)
206 {
207     return vector_size(stack);
208 }
209
210 static inline bool vector_stack_is_empty(Vector *stack)
211 {
212     return vector_is_empty(stack);
213 }
214
215 static inline void push_vector_stack(Vector *stack)
216 {
217     Vector *v = vector_alloc();
218     vector_append(stack, v);
219 }
220
221 static inline Vector *peek_vector_stack(Vector *stack)
222 {
223     assert(!vector_stack_is_empty(stack));
224     return vector_peek_last(stack);
225 }
226
227 static inline Vector *pop_vector_stack(Vector *stack)
228 {
229     assert(!vector_stack_is_empty(stack));
230     return vector_pop_last(stack);
231 }
232
233 static inline void push_ast_node(Vector *stack, Ast_Node *n)
234 {
235     vector_append(peek_vector_stack(stack), n);
236 }
237
238 static inline Ast_Node *pop_ast_node(Vector *stack)
239 {
240     assert(!vector_is_empty(peek_vector_stack(stack)));
241     return vector_pop_last(peek_vector_stack(stack));
242 }
243
244 static inline Ast_Node *peek_ast_node(Vector *stack)
245 {
246     assert(!vector_is_empty(peek_vector_stack(stack)));
247     return vector_peek_last(peek_vector_stack(stack));
248 }
249
250 static inline void commit_decl_stmt()
251 {
252     consistent_stmt_decl_top =
253         vector_size(peek_vector_stack(&decl_stmt_stack));
254 }
255
256 static inline Uns get_consistent_stmt_decl_top()
257 {
258     return consistent_stmt_decl_top;
259 }
260
261 static inline File_Location get_node_location(Uns start_line, Uns start_column)
262 {
263     return FILE_LOCATION_INIT(current_file_name, start_line, start_column);
264 }
265
266 #define PARSER_BINARY_NODE(node_type, first_line, first_column) \

```



```

267 do {
268     Ast_Node *___rhs = pop_ast_node(&decl_stmt_stack); \
269     Ast_Node *___lhs = pop_ast_node(&decl_stmt_stack); \
270     Ast_Node *___n = AST_NODE_BINARY_ALLOC(node_type, \
271         get_node_location(first_line, first_column), \
272         ___lhs, ___rhs); \
273     push_ast_node(&decl_stmt_stack, ___n); \
274 } while (false)
275
276 #define PARSE_BINARY_NODE_REVERSED(node_type, first_line, first_column) \
277 do {
278     Ast_Node *___lhs = pop_ast_node(&decl_stmt_stack); \
279     Ast_Node *___rhs = pop_ast_node(&decl_stmt_stack); \
280     Ast_Node *___n = AST_NODE_BINARY_ALLOC(node_type, \
281         get_node_location(first_line, first_column), \
282         ___lhs, ___rhs); \
283     push_ast_node(&decl_stmt_stack, ___n); \
284 } while (false)
285
286 #define PARSE_BINARY_EXPR(node_type, first_line, first_column) \
287 do {
288     Ast_Node *___rhs = pop_ast_node(&decl_stmt_stack); \
289     Ast_Node *___lhs = pop_ast_node(&decl_stmt_stack); \
290     Ast_Node *___n = AST_EXPR_BINARY_ALLOC(node_type, \
291         get_node_location(first_line, first_column), \
292         ___lhs, ___rhs); \
293     push_ast_node(&decl_stmt_stack, ___n); \
294 } while (false)
295
296 #define PARSE_BINARY_EXPR_REVERSED(node_type, first_line, first_column) \
297 do {
298     Ast_Node *___lhs = pop_ast_node(&decl_stmt_stack); \
299     Ast_Node *___rhs = pop_ast_node(&decl_stmt_stack); \
300     Ast_Node *___n = AST_EXPR_BINARY_ALLOC(node_type, \
301         get_node_location(first_line, first_column), \
302         ___lhs, ___rhs); \
303     push_ast_node(&decl_stmt_stack, ___n); \
304 } while (false)
305
306 #define PARSE_UNARY_NODE(node_type, first_line, first_column) \
307 do {
308     Ast_Node *___expr = pop_ast_node(&decl_stmt_stack); \
309     Ast_Node *___n = AST_NODE_UNARY_ALLOC(node_type, \
310         get_node_location( \
311             first_line, first_column), \
312             ___expr); \
313     push_ast_node(&decl_stmt_stack, ___n); \
314 } while (false)
315
316 #define PARSE_UNARY_EXPR(node_type, first_line, first_column) \
317 do {
318     Ast_Node *___expr = pop_ast_node(&decl_stmt_stack); \
319     Ast_Node *___n = AST_EXPR_UNARY_ALLOC(node_type, \
320         get_node_location( \
321             first_line, first_column), \
322             ___expr); \
323     push_ast_node(&decl_stmt_stack, ___n); \
324 } while (false)
325
326 #define PARSE_VAR_IDEN(node_type, first_line, first_column, text) \
327 do {
328     Ast_Node *___n = AST_VARIABLE_IDEN_ALLOC(node_type, \
329         get_node_location(first_line, first_column), \
330         text); \
331     push_ast_node(&decl_stmt_stack, ___n); \
332 } while (false)
333
334 #define GET_STMT_LIST(node_type, vec_stmts) ({ \
335     Uns ___first_line, ___first_column; \
336     if (!vector_is_empty(vec_stmts)) { \
337         Ast_Node *___tmp_node = vector_peek_first(vec_stmts); \
338         ___first_line = ___tmp_node->location.line; \
339         ___first_column = ___tmp_node->location.column; \
340     } else {

```

```

341     __first_line = yylval.token.lineno;      \
342     __first_column = yylval.token.startcol;  \
343 }                                             \
344 AST_STMT_LIST_ALLOC(node_type,              \
345     get_node_location(                       \
346         __first_line, __first_column), vec_stmts); \
347 })
348
349 static void report_further_errors()
350 {
351     File_Location tmp_loc = FILE_LOCATION_INIT(current_file_name,
352     last_err_line, last_err_column);
353     if (last_err_line == 0 || last_err_column == 0 ||
354         is_error_reported_here(&tmp_loc))
355         return;
356
357     if (!string_is_empty(last_err_text)) {
358         if (!string_is_empty(expect_string)) {
359             parser_report_error(last_err_line, last_err_column,
360                 S("unexpected " QFY("%S") ", expected %S\n"),
361                 last_err_text, expect_string);
362         } else if (!was_error_reported()) {
363             parser_report_error(last_err_line, last_err_column,
364                 S("unexpected " QFY("%S") "\n"), last_err_text);
365         }
366     } else if (!string_is_empty(expect_string)) {
367         parser_report_error(last_err_line, last_err_column,
368             S("expected " QFY("%S") " before " QFY("end of input") "\n"),
369             expect_string);
370     } else if (get_curl_nest()) {
371         parser_report_error(last_err_line, last_err_column,
372             S("expected " QFY("(") " before " QFY("end of input") "\n"));
373     } else if (get_func_nest()) {
374         parser_report_error(last_err_line, last_err_column,
375             S("expected end of function before " QFY("end of input") "\n"),
376             expect_string);
377     } else if (!was_error_reported()) {
378         parser_report_error(last_err_line, last_err_column,
379             S("unexpected " QFY("end of input") "\n"));
380     }
381 }
382
383 void parser_finalize()
384 {
385 }
386
387 #undef DEBUG_TYPE
388 #define DEBUG_TYPE parser-parse
389
390 %}
391
392 %code requires {
393     #include <str.h>
394     #include <parser.h>
395 }
396
397 %union {
398     struct {
399         String text;
400         Parser-Token_Type type;
401         Uns lineno;
402         Uns startcol;
403     } token;
404 }
405
406 %token <token> TKN_PLUS_OP    "@ '+' #"
407 %token <token> TKN_MINUS_OP   "@ '-' #"
408 %token <token> TKN_MUL_OP     "@ '*' #"
409 %token <token> TKN_DIV_OP     "@ '/' #"
410 %token <token> TKN_EQ_OP      "@ '=' #"
411 %token <token> TKN_NEQ_OP     "@ '!=' #"
412 %token <token> TKN_GT_OP      "@ '>' #"
413 %token <token> TKN_LT_OP      "@ '<' #"
414 %token <token> TKN_GTEQ_OP    "@ '>=' #"

```

```

415 %token <token> TKN_LTEQ_OP  "@ '<='#"
416 %token <token> TKN_LAND_OP  "@ '&&'"
417 %token <token> TKN_LOR_OP   "@ '||'"
418 %token <token> TKN_ASSIGN_OP "@ '='#"
419 %token <token> TKN_DOT_OP   "@ '.'#"
420 %token <token> TKN_COMMA_OP "@ ','#"
421 %token <token> TKN_LSQUARE_OP "@ '['#"
422 %token <token> TKN_RSQUARE_OP "@ ']'#"
423 %token <token> TKN_LPAREN_OP "@ '('#"
424 %token <token> TKN_RPAREN_OP "@ ')'#"
425 %token <token> TKN_BANG_OP  "@ '!'"
426 %token <token> TKN_HLINE_OP "@ '|'#"
427 %token <token> TKN_COLON_OP "@ ':'#"
428 %token <token> TKN_SCOLON_OP "@ ';'#"
429 %token <token> TKN_LCURLY_OP "@ '{'"
430 %token <token> TKN_RCURLY_OP "@ '}'#"
431
432 %token <token> TKN_EXTERN_KEY "@ 'extern'#"
433 %token <token> TKN_FUNC_KEY   "@ 'func'#"
434 %token <token> TKN_END_KEY    "@ 'end'#"
435 %token <token> TKN_INT_KEY    "@ 'int'#"
436 %token <token> TKN_VOID_KEY   "@ 'void'#"
437 %token <token> TKN_BOOL_KEY   "@ 'bool'#"
438 %token <token> TKN_ARRAY_KEY  "@ 'array'#"
439 %token <token> TKN_OF_KEY     "@ 'of'#"
440 %token <token> TKN_RECORD_KEY "@ 'record'#"
441 %token <token> TKN_FINALIZE_KEY "@ 'finalize'#"
442 %token <token> TKN_TYPE_KEY   "@ 'type'#"
443 %token <token> TKN_VAR_KEY    "@ 'var'#"
444 %token <token> TKN_IMPORT_KEY "@ 'import'#"
445 %token <token> TKN_PACKAGE_KEY "@ 'package'#"
446
447 %token <token> TKN_CHAR_KEY   "@ 'char'#"
448 %token <token> TKN_STRING_KEY "@ 'string'#"
449
450 %token <token> TKN_RETURN_KEY "@ 'return'#"
451 %token <token> TKN_WRITE_KEY  "@ 'write'#"
452 %token <token> TKN_IF_KEY     "@ 'if'#"
453 %token <token> TKN_ALLOCATE_KEY "@ 'allocate'#"
454 %token <token> TKN_DELETE_KEY "@ 'delete'#"
455 %token <token> TKN_THEN_KEY   "@ 'then'#"
456 %token <token> TKN_WHILE_KEY  "@ 'while'#"
457 %token <token> TKN_DO_KEY     "@ 'do'#"
458 %token <token> TKN_LENGTH_KEY "@ 'length'#"
459 %token <token> TKN_ELSE_KEY   "@ 'else'#"
460 %token <token> TKN_TRUE_KEY   "@ 'true'#"
461 %token <token> TKN_FALSE_KEY  "@ 'false'#"
462 %token <token> TKN_NULL_KEY   "@ 'null'#"
463 %token <token> TKN_CAST_KEY   "@ 'cast'#"
464
465 %token <token> TKN_IDENTIFIER "@ identifier#"
466 %token <token> TKN_INT_CONST  "@ integer constant#"
467 %token <token> TKN_CHAR_CONST "@ char constant#"
468 %token <token> TKN_STRING_CONST "@ string constant#"
469 %token <token> TKN_MODULE_CONST "@ module literal#"
470
471 %token <token> TKN_UNEXPECTED "@ unexpected character#"
472 %token <token> TKN_EOF       "@ 'end of input'#"
473
474 %token <token> error
475
476 %type <token> start
477 %type <token> start_body
478 %type <token> lcurly_op
479 %type <token> rcurly_op
480 %type <token> function
481 %type <token> extern_function
482 %type <token> func_head
483 %type <token> func_head_iden
484 %type <token> func_tail
485 %type <token> type_def
486 %type <token> type
487 %type <token> void_type
488 %type <token> record_decl

```

```

489 %type <token> type_cast
490 %type <token> var_type
491 %type <token> declaration
492 %type <token> decl_stmt_list
493 %type <token> statement
494 %type <token> variable
495 %type <token> direct_record_ref
496 %type <token> identifier
497 %type <token> record_identifier
498 %type <token> variable_identifier
499 %type <token> expression
500 %type <token> var_decl_list
501 %type <token> term
502 %type <token> term_no_var
503 %type <token> exp_list
504 %type <token> record_head
505 %type <token> decl_stmt
506 %type <token> errors
507 %type <token> record_member_last
508 %type <token> record_member_first
509 %type <token> record_seperator
510 %type <token> record_var
511 %type <token> curly_decl_stmt_list
512 %type <token> finalize_decl_stmt
513 %type <token> record_decl_list_last
514 %type <token> record_decl_list_first
515 %type <token> func_key
516 %type <token> func_key_enter
517 %type <token> end_key_leave
518 %type <token> if_stmt
519 %type <token> var_decl_type
520 %type <token> import_start
521 %type <token> package
522 %type <token> colon_list
523
524 /* Expression precedence. */
525 %left TKN_LOR_OP
526 %left TKN_LAND_OP
527 %left TKN_EQ_OP TKN_NEQ_OP TKN_GT_OP TKN_LT_OP TKN_GTEQ_OP TKN_LTEQ_OP
528 %left TKN_PLUS_OP TKN_MINUS_OP
529 %left TKN_MUL_OP TKN_DIV_OP
530
531 /* Precedence to handle if-then-else shift/reduce conflicts. */
532 %right if_stmt_prec
533 %right TKN_ELSE_KEY
534
535 /* Precedence to handle variable shift/reduce conflicts. */
536 %right term_prec
537 %right variable_prec
538
539 /* Handle func tail shift/reduce. */
540 %right func_tail_prec
541
542 /* Handle shift/reduce for package statement errors. */
543 %right package_prec
544
545 /* Precedence to handle type cast shift/reduce conflicts. */
546 %right TKN_IDENTIFIER
547 %right TKN_RPAREN_OP
548 %right TKN_DOT_OP
549 %right TKN_RECORD_KEY
550 %right TKN_LSQUARE_OP
551
552 /* Precedence to handle record seperator shift/reduce conflicts. */
553 %right TKN_SCOLON_OP
554 %right TKN_COMMA_OP
555
556 /* Precedence to handle shift/reduce conflicts related to error recovery
557  * in decl_stmt and record_decl. */
558 %right decl_stmt_prec
559 %right record_member_prec
560 %right TKN_FUNC_KEY
561 %right error
562

```

```

563 %start start
564
565 %define parse.lac full
566 %error-verbose
567
568 %%
569
570 lcurly_op
571 : TKN_LCURLY_OP {
572     inc_curl_nest();
573 }
574 ;
575
576 rcurly_op
577 : TKN_RCURLY_OP {
578     dec_curl_nest();
579 }
580 ;
581
582 func_key
583 : TKN_FUNC_KEY {
584     push_vector_stack(&func_head_stack);
585 }
586 ;
587
588 func_key_enter
589 : func_key {
590     enter_func();
591 }
592
593 end_key_leave
594 : TKN_END_KEY {
595     leave_func();
596 }
597 ;
598
599 errors
600 : errors error
601 | error
602 ;
603
604 start
605 : {
606     push_vector_stack(&decl_stmt_stack);
607 } opt_package start_body {
608     Vector *statements = pop_vector_stack(&decl_stmt_stack);
609     assert(vector_stack_is_empty(&decl_stmt_stack));
610     current_ast->root = GET_STMT_LIST(AST_STMT_LIST, statements);
611 }
612 ;
613
614 start_body
615 : decl_stmt_list TKN_EOF
616 | TKN_EOF
617 ;
618
619 opt_package
620 : opt_scolon_list package %prec package_prec
621 | opt_scolon_list %prec package_prec
622 ;
623
624 opt_scolon_list
625 : scolon_list %prec package_prec
626 | %prec package_prec
627 ;
628
629 scolon_list
630 : scolon_list TKN_SCOLON_OP
631 | TKN_SCOLON_OP
632 ;
633
634 package
635 : TKN_PACKAGE_KEY TKN_MODULE_CONST TKN_SCOLON_OP {
636     Ast_Node *n = AST_MODULE_STRING_ALLOC(AST_PACKAGE_STRING,

```

```

637         get_node_location($2.lineno, $2.startcol, $2.text);
638     push_ast_node(&decl_stmt_stack, n);
639 }
640 | errors %prec package_prec {
641     RECOVER_EXPECT(decl_stmt_recover, $errors.lineno,
642         $errors.startcol, $errors.text);
643 }
644 ;
645
646 function
647 : func_head {
648     push_vector_stack(&decl_stmt_stack);
649 } func_body func_tail[t] {
650     Ast_Variable_Iden *head_iden = AST_CONTAINER_OF(
651         pop_ast_node(&func_head_stack), Ast_Variable_Iden);
652
653     Ast_Node *ret_type = pop_ast_node(&func_head_stack);
654     Vector *params = pop_vector_stack(&func_head_stack);
655     Vector *statements = pop_vector_stack(&decl_stmt_stack);
656     Ast_Node *body = GET_STMT_LIST(AST_FUNC_BODY, statements);
657
658     if ($t.text) {
659         if (string_compare($t.text, head_iden->iden))
660             parser_report_error($t.lineno, $t.startcol,
661                 S("unexpected identifier " QFY("%S")
662                     " after " QFY("end") " of function, "
663                     "expected " QFY("%S") "\n"),
664                 $t.text, head_iden->iden);
665     }
666
667     Ast_Node *n;
668     if (!string_compare(head_iden->iden, S("finalize"))) {
669         n = AST_FUNC_DEF_ALLOC(AST_FIN_FUNC_DEF,
670             get_node_location($1.lineno, $1.startcol), NULL,
671             AST_NODE_OF(head_iden), ret_type, params, body);
672     } else if (!string_compare(head_iden->iden, S("record"))) {
673         n = AST_FUNC_DEF_ALLOC(AST_REC_FUNC_DEF,
674             get_node_location($1.lineno, $1.startcol), NULL,
675             AST_NODE_OF(head_iden), ret_type, params, body);
676     } else {
677         n = AST_FUNC_DEF_ALLOC(AST_FUNC_DEF,
678             get_node_location($1.lineno, $1.startcol), NULL,
679             AST_NODE_OF(head_iden), ret_type, params, body);
680     }
681
682     push_ast_node(&decl_stmt_stack, n);
683 }
684 | func_key_enter func_head_iden TKN_LPAREN_OP opt_par_decl_list
685     TKN_RPAREN_OP error {
686     RECOVER_EXPECT(func_head_recover, $error.lineno,
687         $error.startcol, $error.text);
688 } func_body func_tail
689 | func_key_enter func_head_iden TKN_LPAREN_OP error {
690     RECOVER_UNEXPEC(func_head_recover, $error.lineno,
691         $error.startcol, $error.text);
692 } func_body func_tail
693 | func_key_enter error {
694     RECOVER_EXPECT(func_head_recover, $error.lineno,
695         $error.startcol, $error.text);
696 } func_body func_tail
697 ;
698
699 func_head_iden
700 : TKN_IDENTIFIER
701 | TKN_FINALIZE_KEY
702 | TKN_RECORD_KEY
703 ;
704
705 func_head
706 : func_key_enter func_head_iden[i] TKN_LPAREN_OP opt_par_decl_list
707     TKN_RPAREN_OP TKN_COLON_OP void_type {
708     push_ast_node(&func_head_stack, pop_ast_node(&decl_stmt_stack));
709     Ast_Node *iden = AST_VARIABLE_IDEN_ALLOC(AST_VARIABLE_IDEN,
710         get_node_location($i.lineno, $i.startcol),

```

```

711         $i.text);
712     push_ast_node(&func_head_stack, iden);
713 }
714 ;
715
716 func_body
717 : opt_decl_stmt_list
718 ;
719
720 func_tail
721 : end_key_leave func_head_iden[i] {
722     $$ = $i;
723 }
724 | end_key_leave error %prec func_tail_prec {
725     RECOVER_EXPECT(decl_stmt_recover, $error.lineno,
726         $error.startcol, $error.text);
727     $$ .text = NULL;
728 }
729 ;
730
731 extern_function
732 : TKN_EXTERN_KEY TKN_LPAREN_OP TKN_IDENTIFIER[t] TKN_RPAREN_OP func_key
733   TKN_IDENTIFIER[i] TKN_LPAREN_OP opt_par_decl_list TKN_RPAREN_OP
734   TKN_COLON_OP void_type {
735     Ast_Node *ret_type = pop_ast_node(&decl_stmt_stack);
736     Vector *params = pop_vector_stack(&func_head_stack);
737     Ast_Node *func_iden = AST_VARIABLE_IDEN_ALLOC(AST_VARIABLE_IDEN,
738         get_node_location($i.lineno, $i.startcol),
739         $i.text);
740     Ast_Node *ext_iden = AST_TYPE_IDEN_ALLOC(AST_TYPE_IDEN,
741         get_node_location($t.lineno, $t.startcol),
742         $t.text);
743     Ast_Node *n = AST_FUNC_DEF_ALLOC(AST_EXT_FUNC_DECL,
744         get_node_location($l.lineno, $l.startcol),
745         ext_iden, func_iden, ret_type, params, NULL);
746     push_ast_node(&decl_stmt_stack, n);
747 }
748 | TKN_EXTERN_KEY TKN_LPAREN_OP TKN_IDENTIFIER[t] TKN_RPAREN_OP func_key
749   error {
750     RECOVER_EXPECT(extern_func_recover, $error.lineno,
751         $error.startcol, $error.text);
752 }
753
754 opt_par_decl_list
755 : par_decl_list
756 |
757 ;
758
759 par_decl_list
760 : par_decl_list TKN_COMMA_OP var_type {
761     push_ast_node(&func_head_stack, pop_ast_node(&decl_stmt_stack));
762 }
763 | var_type {
764     push_ast_node(&func_head_stack, pop_ast_node(&decl_stmt_stack));
765 }
766 ;
767
768 type
769 : TKN_IDENTIFIER {
770     Ast_Node *n = AST_TYPE_IDEN_ALLOC(AST_TYPE_IDEN,
771         get_node_location(
772             $TKN_IDENTIFIER.lineno, $TKN_IDENTIFIER.startcol),
773             $TKN_IDENTIFIER.text);
774     push_ast_node(&decl_stmt_stack, n);
775 }
776 | TKN_INT_KEY {
777     Ast_Node *n = AST_EMPTY_ALLOC(AST_SIMPLE_TYPE_INT,
778         get_node_location($l.lineno, $l.startcol));
779     push_ast_node(&decl_stmt_stack, n);
780 }
781 | TKN_BOOL_KEY {
782     Ast_Node *n = AST_EMPTY_ALLOC(AST_SIMPLE_TYPE_BOOL,
783         get_node_location($l.lineno, $l.startcol));
784     push_ast_node(&decl_stmt_stack, n);

```

```

785     }
786     | TKN_ARRAY_KEY TKN_OF_KEY type[t] {
787         Ast_Node *n = AST_TYPE_ALLOC(AST_TYPE_ARY,
788             get_node_location($1.lineno, $1.startcol),
789             pop_ast_node(&decl_stmt_stack));
790         push_ast_node(&decl_stmt_stack, n);
791     }
792     | TKN_CHAR_KEY {
793         Ast_Node *n = AST_EMPTY_ALLOC(AST_SIMPLE_TYPE_CHAR,
794             get_node_location($1.lineno, $1.startcol));
795         push_ast_node(&decl_stmt_stack, n);
796     }
797     | TKN_STRING_KEY {
798         Ast_Node *n = AST_EMPTY_ALLOC(AST_SIMPLE_TYPE_STRING,
799             get_node_location($1.lineno, $1.startcol));
800         push_ast_node(&decl_stmt_stack, n);
801     }
802     | record_decl
803     ;
804
805 void_type
806 : type
807 | TKN_VOID_KEY {
808     Ast_Node *n = AST_EMPTY_ALLOC(AST_SIMPLE_TYPE_VOID,
809         get_node_location($1.lineno, $1.startcol));
810     push_ast_node(&decl_stmt_stack, n);
811 }
812 ;
813
814 record_decl
815 : record_head lcurly_op {
816     dec_extend_nest();
817     push_extend_nest();
818     push_vector_stack(&decl_stmt_stack);
819 } opt_record_decl_list rcurlly_op[r] {
820     DLOG("POP extend_stack\n");
821     Vector *extend = pop_vector_stack(&extend_stack);
822     Vector *body = pop_vector_stack(&decl_stmt_stack);
823     Ast_Node *n = AST_TYPE_REC_ALLOC(AST_TYPE_REC,
824         get_node_location($1.lineno, $1.startcol),
825         extend, body);
826     push_ast_node(&decl_stmt_stack, n);
827     pop_extend_nest();
828 }
829 ;
830
831 record_head
832 : TKN_RECORD_KEY TKN_OF_KEY {
833     DLOG("PUSH extend_stack\n");
834     inc_extend_nest();
835     push_vector_stack(&extend_stack);
836 } opt_extend
837 ;
838
839 opt_extend
840 : record_extend_list
841 |
842 ;
843
844 opt_record_decl_list
845 : record_decl_list_first
846 |
847 ;
848
849 record_extend_list
850 : record_extend_list TKN_COMMA_OP record_var {
851     push_ast_node(&extend_stack, pop_ast_node(&decl_stmt_stack));
852 }
853 | record_var {
854     push_ast_node(&extend_stack, pop_ast_node(&decl_stmt_stack));
855 }
856 ;
857
858 record_var

```



```

859 : TKN_IDENTIFIER {
860     Ast_Node *n = AST_TYPE_IDEN_ALLOC(AST_TYPE_IDEN,
861         get_node_location(
862             $TKN_IDENTIFIER.lineno, $TKN_IDENTIFIER.startcol),
863             $TKN_IDENTIFIER.text);
864     push_ast_node(&decl_stmt_stack, n);
865 }
866 /*
867 | record_decl
868 */
869 ;
870
871 record_decl_list_first
872 : record_decl_list_last record_member_first
873 | record_member_first
874 ;
875
876 record_decl_list_last
877 : record_decl_list_last record_member_last
878 | record_member_last
879 ;
880
881 record_member_first
882 : function {
883     commit_decl_stmt();
884 }
885 | var_type {
886     commit_decl_stmt();
887 }
888 | var_type record_seperator {
889     commit_decl_stmt();
890 }
891 | TKN_VAR_KEY var_type {
892     commit_decl_stmt();
893 }
894 | TKN_VAR_KEY var_type record_seperator {
895     commit_decl_stmt();
896 }
897 | type_def {
898     commit_decl_stmt();
899 }
900 | type_def record_seperator {
901     commit_decl_stmt();
902 }
903 | import_start {
904     Ast_Node *n = AST_MODULE_STRING_ALLOC(AST_IMPORT_STRING,
905         get_node_location($1.lineno, $1.startcol), $1.text);
906     push_ast_node(&decl_stmt_stack, n);
907     commit_decl_stmt();
908 }
909 | import_start record_seperator {
910     Ast_Node *n = AST_MODULE_STRING_ALLOC(AST_IMPORT_STRING,
911         get_node_location($1.lineno, $1.startcol), $1.text);
912     push_ast_node(&decl_stmt_stack, n);
913     commit_decl_stmt();
914 }
915 | record_seperator
916 | errors %prec record_member_prec {
917     RECOVER_UNEXPEC(record_body_recover, $errors.lineno, $errors.startcol,
918         $errors.text);
919 }
920 | errors TKN_LCURLY_OP {
921     RECOVER_UNEXPEC(record_body_recover, $errors.lineno, $errors.startcol,
922         $errors.text);
923 }
924 | errors TKN_FUNC_KEY %prec record_member_prec {
925     RECOVER_UNEXPEC(record_body_recover, $errors.lineno, $errors.startcol,
926         $errors.text);
927 }
928 ;
929
930 record_member_last
931 : function {
932     commit_decl_stmt();

```

```

933     }
934     | var_type record_seperator {
935         commit_decl_stmt();
936     }
937     | TKN_VAR_KEY var_type record_seperator {
938         commit_decl_stmt();
939     }
940     | type_def record_seperator {
941         commit_decl_stmt();
942     }
943     | import_start record_seperator {
944         Ast_Node *n = AST_MODULE_STRING_ALLOC(AST_IMPORT_STRING,
945         get_node_location($1.lineno, $1.startcol), $1.text);
946         push_ast_node(&decl_stmt_stack, n);
947         commit_decl_stmt();
948     }
949     | record_seperator
950     | errors %prec record_member_prec {
951         RECOVER_UNEXPEC(record_body_recover, $errors.lineno, $errors.startcol,
952         $errors.text);
953     }
954     | errors TKN_LCURLY_OP {
955         RECOVER_UNEXPEC(record_body_recover, $errors.lineno, $errors.startcol,
956         $errors.text);
957     }
958     | errors TKN_FUNC_KEY %prec record_member_prec {
959         RECOVER_UNEXPEC(record_body_recover, $errors.lineno, $errors.startcol,
960         $errors.text);
961     }
962     ;
963
964 record_seperator
965     : TKN_COMMA_OP
966     | TKN_SCOLON_OP
967     ;
968
969 var_decl_list
970     : var_decl_list TKN_COMMA_OP var_decl_type
971     | var_decl_type
972     ;
973
974 var_decl_type
975     : var_type
976     | var_type[i] TKN_ASSIGN_OP[a] expression {
977         Ast_Node *expr = pop_ast_node(&decl_stmt_stack);
978
979         Ast_Node *iden_n = AST_VARIABLE_IDEN_ALLOC(AST_VARIABLE_IDEN,
980         get_node_location($i.lineno, $i.startcol),
981         $i.text);
982
983         Ast_Node *n = AST_NODE_BINARY_ALLOC(AST_ASSIGNMENT,
984         get_node_location($a.lineno, $a.startcol),
985         iden_n, expr);
986         push_ast_node(&decl_stmt_stack, n);
987     }
988     ;
989
990 var_type
991     : TKN_IDENTIFIER[i] TKN_COLON_OP type {
992         PARSE_VAR_IDEN(AST_VARIABLE_IDEN, $i.lineno, $i.startcol, $i.text);
993         PARSE_BINARY_NODE_REVERSED(AST_VAR_DECL, $1.lineno, $1.startcol);
994     }
995     ;
996
997 opt_decl_stmt_list
998     : decl_stmt_list
999     |
1000     ;
1001
1002 decl_stmt_list
1003     : decl_stmt_list decl_stmt
1004     | decl_stmt
1005     ;
1006

```

```

1007 decl_stmt
1008 : declaration {
1009     commit_decl_stmt();
1010 }
1011 | statement {
1012     commit_decl_stmt();
1013 }
1014 | errors %prec decl_stmt_prec {
1015     RECOVER_EXPECT(decl_stmt_recover, $errors.lineno,
1016                   $errors.startcol, $errors.text);
1017 }
1018 ;
1019
1020 curly_decl_stmt_list
1021 : lcurly_op {
1022     push_vector_stack(&decl_stmt_stack);
1023 } opt_decl_stmt_list rcurly_op {
1024     Vector *statements = pop_vector_stack(&decl_stmt_stack);
1025     push_ast_node(&decl_stmt_stack,
1026                  GET_STMT_LIST(AST_STMT_LIST, statements));
1027 }
1028 ;
1029
1030 finalize_decl_stmt
1031 : TKN_FINALIZE_KEY {
1032     push_vector_stack(&decl_stmt_stack);
1033 } decl_stmt {
1034     Vector *statements = pop_vector_stack(&decl_stmt_stack);
1035     push_ast_node(&decl_stmt_stack,
1036                  GET_STMT_LIST(AST_FIN_STMT_LIST, statements));
1037 }
1038 ;
1039
1040 declaration
1041 : function
1042 | extern_function TKN_SCOLON_OP
1043 | type_def TKN_SCOLON_OP
1044 | TKN_VAR_KEY var_decl_list TKN_SCOLON_OP /* Ignore. */
1045 | var_decl_list TKN_SCOLON_OP /* Ignore. */
1046 ;
1047
1048 type_def
1049 : TKN_TYPE_KEY TKN_IDENTIFIER TKN_ASSIGN_OP type {
1050     PARSE_VAR_IDEN(AST_VARIABLE_IDEN,
1051                   $TKN_IDENTIFIER.lineno, $TKN_IDENTIFIER.startcol,
1052                   $TKN_IDENTIFIER.text);
1053     PARSE_BINARY_NODE_REVERSED(AST_TYPE_DEF, $1.lineno, $1.startcol);
1054 }
1055 ;
1056
1057 statement
1058 : TKN_SCOLON_OP /* Ignore. */
1059 | import_start TKN_SCOLON_OP {
1060     Ast_Node *n = AST_MODULE_STRING_ALLOC(AST_IMPORT_STRING,
1061                                           get_node_location($1.lineno, $1.startcol), $1.text);
1062     push_ast_node(&decl_stmt_stack, n);
1063 }
1064 | TKN_ALLOCATE_KEY variable TKN_SCOLON_OP {
1065     PARSE_UNARY_NODE(AST_ALLOC_REC, $1.lineno, $1.startcol);
1066 }
1067 | TKN_ALLOCATE_KEY variable[v] TKN_OF_KEY TKN_RECORD_KEY[r] TKN_LPAREN_OP {
1068     push_vector_stack(&expr_stack);
1069 } arg_list TKN_RPAREN_OP {
1070     Vector *v = pop_vector_stack(&expr_stack);
1071
1072     if (vector_is_empty(v)) {
1073         vector_destroy(v, NULL);
1074         PARSE_UNARY_NODE(AST_ALLOC_REC, $1.lineno, $1.startcol);
1075     } else {
1076         Ast_Node *rec_iden = pop_ast_node(&decl_stmt_stack);
1077
1078         Ast_Node *func_iden = AST_VARIABLE_IDEN_ALLOC(AST_VARIABLE_IDEN,
1079                                                       get_node_location($r.lineno, $r.startcol), S("record"));
1080

```

```

1081     Ast_Node *func_call = AST_EXPR_FUNC_CALL_ALLOC(AST_EXPR_FUNC_CALL,
1082     get_node_location($r.lineno, $r.startcol), func_iden, v);
1083
1084     Ast_Node *n = AST_NODE_BINARY_ALLOC(AST_ALLOC_REC_CALL,
1085     get_node_location($l.lineno, $l.startcol),
1086     rec_iden, func_call);
1087     push_ast_node(&decl_stmt_stack, n);
1088 }
1089 }
1090 | TKN_ALLOCATE_KEY variable TKN_OF_KEY TKN_LENGTH_KEY
1091     expression TKN_SCOLON_OP {
1092     PARSE_BINARY_NODE(AST_ALLOC_ARRAY, $l.lineno, $l.startcol);
1093 }
1094 | TKN_DELETE_KEY variable TKN_SCOLON_OP {
1095     PARSE_UNARY_NODE(AST_DELETE, $l.lineno, $l.startcol);
1096 }
1097 | TKN_RETURN_KEY expression TKN_SCOLON_OP {
1098     PARSE_UNARY_NODE(AST_RETURN_STMT, $l.lineno, $l.startcol);
1099 }
1100 | TKN_RETURN_KEY TKN_SCOLON_OP {
1101     Ast_Node *vnode = AST_EMPTY_ALLOC(AST_SIMPLE_TYPE_VOID,
1102     get_node_location($l.lineno, $l.startcol));
1103     push_ast_node(&decl_stmt_stack, vnode);
1104     PARSE_UNARY_NODE(AST_RETURN_STMT, $l.lineno, $l.startcol);
1105 }
1106 | TKN_WRITE_KEY expression TKN_SCOLON_OP {
1107     PARSE_UNARY_NODE(AST_WRITE_STMT, $l.lineno, $l.startcol);
1108 }
1109 | variable TKN_ASSIGN_OP[a] expression TKN_SCOLON_OP {
1110     PARSE_BINARY_NODE(AST_ASSIGNMENT, $a.lineno, $a.startcol);
1111 }
1112 | if_stmt %prec if_stmt_prec {
1113     Vector *statements = pop_vector_stack(&decl_stmt_stack);
1114     Ast_Node *s = GET_STMT_LIST(AST_STMT_LIST, statements);
1115     push_ast_node(&decl_stmt_stack, s);
1116     PARSE_BINARY_NODE(AST_IF_STMT, $l.lineno, $l.startcol);
1117 }
1118 | if_stmt TKN_ELSE_KEY {
1119     push_vector_stack(&decl_stmt_stack);
1120 } decl_stmt {
1121     Vector *if_statements = pop_vector_stack(&decl_stmt_stack);
1122     Vector *else_statements = pop_vector_stack(&decl_stmt_stack);
1123     Ast_Node *rhs = GET_STMT_LIST(AST_STMT_LIST, if_statements);
1124     Ast_Node *mid = GET_STMT_LIST(AST_STMT_LIST, else_statements);
1125     Ast_Node *lhs = pop_ast_node(&decl_stmt_stack);
1126     Ast_Node *n = AST_NODE_TERNARY_ALLOC(AST_IF_ELSE_STMT,
1127     get_node_location($l.lineno, $l.startcol),
1128     lhs, mid, rhs);
1129     push_ast_node(&decl_stmt_stack, n);
1130 }
1131 | TKN_WHILE_KEY expression TKN_DO_KEY {
1132     push_vector_stack(&decl_stmt_stack);
1133 } decl_stmt {
1134     Vector *statements = pop_vector_stack(&decl_stmt_stack);
1135     Ast_Node *s = GET_STMT_LIST(AST_STMT_LIST, statements);
1136     push_ast_node(&decl_stmt_stack, s);
1137     PARSE_BINARY_NODE(AST_WHILE_STMT, $l.lineno, $l.startcol);
1138 }
1139 | expression TKN_SCOLON_OP /* Ignore. */
1140 | curly_decl_stmt_list /* Ignore. */
1141 | finalize_decl_stmt
1142 ;
1143
1144 import_start
1145 : TKN_IMPORT_KEY TKN_MODULE_CONST {
1146     $$ = $2;
1147 }
1148 ;
1149
1150 if_stmt
1151 : TKN_IF_KEY expression TKN_THEN_KEY {
1152     push_vector_stack(&decl_stmt_stack);
1153 } decl_stmt
1154 ;

```

```

1155
1156 expression
1157 : expression TKN_LOR_OP expression {
1158     PARSER_BINARY_EXPR(AST_EXPR_LOR, $2.lineno, $2.startcol);
1159 }
1160 | expression TKN_LAND_OP expression {
1161     PARSER_BINARY_EXPR(AST_EXPR_LAND, $2.lineno, $2.startcol);
1162 }
1163 | expression TKN_EQ_OP expression {
1164     PARSER_BINARY_EXPR(AST_EXPR_EQ, $2.lineno, $2.startcol);
1165 }
1166 | expression TKN_NEQ_OP expression {
1167     PARSER_BINARY_EXPR(AST_EXPR_NEQ, $2.lineno, $2.startcol);
1168 }
1169 | expression TKN_GT_OP expression {
1170     PARSER_BINARY_EXPR(AST_EXPR_GT, $2.lineno, $2.startcol);
1171 }
1172 | expression TKN_LT_OP expression {
1173     PARSER_BINARY_EXPR(AST_EXPR_LT, $2.lineno, $2.startcol);
1174 }
1175 | expression TKN_GTEQ_OP expression {
1176     PARSER_BINARY_EXPR(AST_EXPR_GTEQ, $2.lineno, $2.startcol);
1177 }
1178 | expression TKN_LTEQ_OP expression {
1179     PARSER_BINARY_EXPR(AST_EXPR_LTEQ, $2.lineno, $2.startcol);
1180 }
1181 | expression TKN_PLUS_OP expression {
1182     PARSER_BINARY_EXPR(AST_EXPR_PLUS, $2.lineno, $2.startcol);
1183 }
1184 | expression TKN_MINUS_OP expression {
1185     PARSER_BINARY_EXPR(AST_EXPR_MINUS, $2.lineno, $2.startcol);
1186 }
1187 | expression TKN_MUL_OP expression {
1188     PARSER_BINARY_EXPR(AST_EXPR_MUL, $2.lineno, $2.startcol);
1189 }
1190 | expression TKN_DIV_OP expression {
1191     PARSER_BINARY_EXPR(AST_EXPR_DIV, $2.lineno, $2.startcol);
1192 }
1193 | term /* Ignore. */
1194 ;
1195
1196 term
1197 : variable %prec term_prec /* Ignore. */
1198 | term_no_var
1199 ;
1200
1201 term_no_var
1202 : TKN_LPAREN_OP expression TKN_RPAREN_OP /* Ignore. */
1203 | type_cast term_no_var {
1204     PARSER_BINARY_EXPR(AST_EXPR_CAST, $1.lineno, $1.startcol);
1205 }
1206 | TKN_BANG_OP term {
1207     PARSER_UNARY_EXPR(AST_EXPR_LNOT, $1.lineno, $1.startcol);
1208 }
1209 | TKN_INT_CONST {
1210     Ast_Node *n = AST_EXPR_INT_ALLOC(AST_EXPR_INT,
1211         get_node_location($1.lineno, $1.startcol), $1.text);
1212     push_ast_node(&decl_stmt_stack, n);
1213 }
1214 | TKN_TRUE_KEY {
1215     Ast_Node *n = AST_EXPR_BOOL_ALLOC(AST_EXPR_BOOL,
1216         get_node_location($1.lineno, $1.startcol), true);
1217     push_ast_node(&decl_stmt_stack, n);
1218 }
1219 | TKN_FALSE_KEY {
1220     Ast_Node *n = AST_EXPR_BOOL_ALLOC(AST_EXPR_BOOL,
1221         get_node_location($1.lineno, $1.startcol), false);
1222     push_ast_node(&decl_stmt_stack, n);
1223 }
1224 | TKN_NULL_KEY {
1225     Ast_Node *n = AST_EXPR_NULL_ALLOC(AST_EXPR_NULL,
1226         get_node_location($1.lineno, $1.startcol));
1227     push_ast_node(&decl_stmt_stack, n);
1228 }

```

```

1229 | TKN_CHAR_CONST {
1230 |     Ast_Node *n = AST_EXPR_CHAR_ALLOC(AST_EXPR_CHAR,
1231 |         get_node_location($1.lineno, $1.startcol), $1.text);
1232 |     push_ast_node(&decl_stmt_stack, n);
1233 | }
1234 | TKN_STRING_CONST {
1235 |     Ast_Node *n = AST_EXPR_STRING_ALLOC(AST_EXPR_STRING,
1236 |         get_node_location($1.lineno, $1.startcol), $1.text);
1237 |     push_ast_node(&decl_stmt_stack, n);
1238 | }
1239 | TKN_HLINE_OP expression TKN_HLINE_OP {
1240 |     PARSE_UNARY_EXPR(AST_EXPR_ABS, $1.lineno, $1.startcol);
1241 | }
1242 | ;
1243
1244 variable
1245 : variable TKN_DOT_OP variable_identifier[v] {
1246 |     PARSE_BINARY_EXPR(AST_EXPR_DOT_REF, $1.lineno, $1.startcol);
1247 |     $$ = $v;
1248 | }
1249 | TKN_LPAREN_OP TKN_IDENTIFIER TKN_RPAREN_OP
1250 |     TKN_DOT_OP variable_identifier[v] {
1251 |     PARSE_VAR_IDEN(AST_VARIABLE_IDEN,
1252 |         $TKN_IDENTIFIER.lineno, $TKN_IDENTIFIER.startcol,
1253 |         $TKN_IDENTIFIER.text);
1254 |     PARSE_BINARY_EXPR_REVERSED(AST_EXPR_DOT_REF, $1.lineno, $1.startcol);
1255 |     $$ = $v;
1256 | }
1257 | TKN_LPAREN_OP TKN_IDENTIFIER TKN_RPAREN_OP
1258 |     TKN_LSQUARE_OP expression TKN_RSQUARE_OP[s] {
1259 |     PARSE_VAR_IDEN(AST_VARIABLE_IDEN,
1260 |         $TKN_IDENTIFIER.lineno, $TKN_IDENTIFIER.startcol,
1261 |         $TKN_IDENTIFIER.text);
1262 |     PARSE_BINARY_EXPR_REVERSED(AST_EXPR_ARY_REF, $1.lineno, $1.startcol);
1263 |     $$ = $s;
1264 | }
1265 | variable TKN_LSQUARE_OP expression TKN_RSQUARE_OP[s] {
1266 |     PARSE_BINARY_EXPR(AST_EXPR_ARY_REF, $1.lineno, $1.startcol);
1267 |     $$ = $s;
1268 | }
1269 | type_cast[c] variable %prec variable_prec {
1270 |     PARSE_BINARY_EXPR(AST_EXPR_CAST, $1.lineno, $1.startcol);
1271 |     $$ = $c;
1272 | }
1273 | TKN_LPAREN_OP variable[v] TKN_RPAREN_OP {
1274 |     $$ = $v;
1275 | }
1276 | record_identifier[v] %prec variable_prec {
1277 |     $$ = $v;
1278 | }
1279 | TKN_LPAREN_OP record_identifier[v] TKN_RPAREN_OP {
1280 |     $$ = $v;
1281 | }
1282 | TKN_LPAREN_OP TKN_IDENTIFIER[t] TKN_RPAREN_OP {
1283 |     PARSE_VAR_IDEN(AST_VARIABLE_IDEN, $1.lineno, $1.startcol, $2.text);
1284 |     $$ = $t;
1285 | }
1286 | direct_record_ref TKN_DOT_OP variable_identifier[v] {
1287 |     PARSE_BINARY_EXPR(AST_EXPR_DIRECT_REF, $1.lineno, $1.startcol);
1288 | }
1289 | direct_record_ref TKN_LPAREN_OP[p] {
1290 |     push_vector_stack(&expr_stack);
1291 | } arg_list TKN_RPAREN_OP[r] {
1292 |     Vector *v = pop_vector_stack(&expr_stack);
1293 |
1294 |     Ast_Node *iden = AST_VARIABLE_IDEN_ALLOC(AST_VARIABLE_IDEN,
1295 |         get_node_location($p.lineno, $p.startcol), S("record"));
1296 |
1297 |     Ast_Node *n = AST_EXPR_FUNC_CALL_ALLOC(AST_EXPR_FUNC_CALL,
1298 |         get_node_location($1.lineno, $1.startcol), iden, v);
1299 |     push_ast_node(&decl_stmt_stack, n);
1300 |
1301 |     PARSE_BINARY_EXPR(AST_EXPR_DIRECT_REF, $1.lineno, $1.startcol);
1302 |     $$ = $r;

```

```

1303     }
1304     ;
1305
1306     direct_record_ref
1307     : TKN_RECORD_KEY TKN_LSQUARE_OP record_var TKN_RSQUARE_OP
1308     | TKN_RECORD_KEY TKN_LSQUARE_OP TKN_RSQUARE_OP {
1309         push_ast_node(&decl_stmt_stack, NULL);
1310     }
1311     ;
1312
1313     variable_identifier
1314     : identifier
1315     | TKN_IDENTIFIER[i] TKN_LPAREN_OP {
1316         push_vector_stack(&expr_stack);
1317     } arg_list TKN_RPAREN_OP[p] {
1318         Vector *v = pop_vector_stack(&expr_stack);
1319
1320         Ast_Node *iden = AST_VARIABLE_IDEN_ALLOC(AST_VARIABLE_IDEN,
1321             get_node_location($1.lineno, $1.startcol),
1322             $i.text);
1323
1324         Ast_Node *n = AST_EXPR_FUNC_CALL_ALLOC(AST_EXPR_FUNC_CALL,
1325             get_node_location($1.lineno, $1.startcol), iden, v);
1326         push_ast_node(&decl_stmt_stack, n);
1327         $$ = $p;
1328     }
1329     ;
1330
1331     record_identifier
1332     : identifier
1333     | TKN_RECORD_KEY[r] {
1334         Ast_Node *n = AST_EMPTY_ALLOC(AST_REC_SELF_PTR,
1335             get_node_location($1.lineno, $1.startcol));
1336         push_ast_node(&decl_stmt_stack, n);
1337     }
1338     | TKN_IDENTIFIER[i] TKN_LPAREN_OP {
1339         push_vector_stack(&expr_stack);
1340     } arg_list TKN_RPAREN_OP[p] {
1341         Vector *v = pop_vector_stack(&expr_stack);
1342
1343         Ast_Node *iden = AST_VARIABLE_IDEN_ALLOC(AST_VARIABLE_IDEN,
1344             get_node_location($1.lineno, $1.startcol),
1345             $i.text);
1346
1347         Ast_Node *n = AST_EXPR_FUNC_CALL_ALLOC(AST_EXPR_FUNC_CALL,
1348             get_node_location($1.lineno, $1.startcol), iden, v);
1349         push_ast_node(&decl_stmt_stack, n);
1350         $$ = $p;
1351     }
1352     ;
1353
1354     identifier
1355     : TKN_IDENTIFIER[t] {
1356         PARSE_VAR_IDEN(AST_VARIABLE_IDEN, $1.lineno, $1.startcol, $1.text);
1357         $$ = $t;
1358     }
1359     ;
1360
1361     arg_list
1362     : exp_list
1363     |
1364     ;
1365
1366     exp_list
1367     : exp_list TKN_COMMA_OP expression {
1368         push_ast_node(&expr_stack, pop_ast_node(&decl_stmt_stack));
1369     }
1370     | expression {
1371         push_ast_node(&expr_stack, pop_ast_node(&decl_stmt_stack));
1372     }
1373     ;
1374
1375     type_cast
1376     : TKN_CAST_KEY TKN_LPAREN_OP type TKN_RPAREN_OP

```

```

1377     ;
1378     %%
1379
1380     static Uns prev_err_line, prev_err_col;
1381
1382     static bool same_as_prev_node(Uns line, Uns col)
1383     {
1384         bool ret;
1385         if (prev_err_line == line && prev_err_col == col) {
1386             ret = true;
1387         } else {
1388             prev_err_line = line;
1389             prev_err_col = col;
1390             ret = false;
1391         }
1392         return ret;
1393     }
1394
1395     static void finish_recover()
1396     {
1397         Ast_Node *n;
1398         Vector *v = peek_vector_stack(&decl_stmt_stack);
1399         while (vector_size(v) > get_consistent_stmt_decl_top()) {
1400             n = vector_pop_last(v);
1401             ast_visitor_delete_accept_visitor(n);
1402         }
1403         for (; get_extend_nest(); dec_extend_nest()) {
1404             v = pop_vector_stack(&extend_stack);
1405             VECTOR_FOR_EACH_ENTRY(v, n)
1406                 ast_visitor_delete_accept_visitor(n);
1407             vector_destroy(v, NULL);
1408         }
1409         clear_vector_stack(&expr_stack);
1410     }
1411
1412     static void finish_func_head_stack()
1413     {
1414         Ast_Node *n;
1415         Vector *head = pop_vector_stack(&func_head_stack);
1416         VECTOR_FOR_EACH_ENTRY(head, n)
1417             ast_visitor_delete_accept_visitor(n);
1418         vector_destroy(head, NULL);
1419     }
1420
1421     #undef DEBUG_TYPE
1422     #define DEBUG_TYPE parser-recover:parser-func-recover
1423     static void func_head_recover()
1424     {
1425         Parser-Token_Type tt = yylval.token.type;
1426         Int recover_func_nest = 0, recover_curl_nest = 0;
1427         DLOG(" - START TOKEN: %d\n", tt);
1428         switch (tt) {
1429             case TKN_END_KEY:
1430                 goto out;
1431             default:
1432                 break;
1433         }
1434         while (tt) {
1435             switch (tt) {
1436                 case TKN_FUNC_KEY:
1437                     DLOG("INC FUNC NEST\n");
1438                     ++recover_func_nest;
1439                     break;
1440                 case TKN_LCURLY_OP:
1441                     DLOG("INC CURL NEST\n");
1442                     ++recover_curl_nest;
1443                     break;
1444                 case TKN_END_KEY:
1445                     if (!recover_func_nest) {
1446                         if (!recover_curl_nest)
1447                             goto out;
1448                     } else {
1449                         DLOG("DEC FUNC NEST\n");
1450                         --recover_func_nest;

```



```

1451     }
1452     break;
1453     case TKN_RCURLY_OP:
1454         if (recover_curl_nest) {
1455             DLOG("DEC CURL NEST\n");
1456             --recover_curl_nest;
1457         }
1458         break;
1459     case TKN_EOF:
1460         if (recover_curl_nest)
1461             inc_curl_nest();
1462         if (recover_func_nest)
1463             inc_func_nest();
1464         goto out;
1465     default:
1466         break;
1467 }
1468 tt = yylex();
1469 };
1470 out:
1471 scanner_push_back_token();
1472 finish_func_head_stack();
1473 finish_recover();
1474 }
1475
1476 static void extern_func_recover()
1477 {
1478     decl_stmt_recover();
1479     finish_func_head_stack();
1480 }
1481
1482 #undef DEBUG_TYPE
1483 #define DEBUG_TYPE parser-recover:parser-stmt-recover
1484 static void decl_stmt_recover()
1485 {
1486     Parser-Token_Type tt = yylval.token.type;
1487     Int recover_func_nest = 0, recover_curl_nest = 0;
1488     DLOG("START TOKEN: %u\n", tt);
1489     switch (tt) {
1490     case TKN_RCURLY_OP:
1491         if (get_curl_nest())
1492             goto out;
1493         tt = yylex();
1494         break;
1495     default:
1496         break;
1497     }
1498     while (tt) {
1499         DLOG("TOKEN VAL: %d\n", tt);
1500         switch (tt) {
1501         case TKN_IMPORT_KEY:
1502         case TKN_ALLOCATE_KEY:
1503         case TKN_IF_KEY:
1504         case TKN_WRITE_KEY:
1505         case TKN_RETURN_KEY:
1506         case TKN_TYPE_KEY:
1507         case TKN_VAR_KEY:
1508         case TKN_WHILE_KEY:
1509         case TKN_SCOLON_OP:
1510             if (!recover_func_nest && !recover_curl_nest)
1511                 goto out;
1512             break;
1513         case TKN_FUNC_KEY:
1514             DLOG("INC FUNC NEST\n");
1515             ++recover_func_nest;
1516             break;
1517         case TKN_LCURLY_OP:
1518             DLOG("INC CURL NEST\n");
1519             ++recover_curl_nest;
1520             break;
1521         case TKN_END_KEY:
1522             if (!recover_func_nest) {
1523                 if (!recover_curl_nest) {
1524                     DLOG("END TOKEN PUSH\n");

```

```

1525         goto out;
1526     }
1527 } else {
1528     DLOG("DEC FUNC NEST\n");
1529     if (!--recover_func_nest && !recover_curl_nest) {
1530         tt = yylex();
1531         if (tt != TKN_IDENTIFIER)
1532             goto out;
1533         tt = yylex();
1534         if (tt != TKN_COMMA_OP)
1535             goto out;
1536         continue;
1537     }
1538 }
1539 break;
1540 case TKN_RCURLY_OP:
1541     DLOG("FOUND '}' CURL NEST IS CURRENTLY: %U\n", recover_curl_nest);
1542     if (!recover_curl_nest) {
1543         if (!recover_func_nest) {
1544             if (!get_curl_nest()) {
1545                 DLOG("OUT NO PUSH '}'\n");
1546                 goto out_no_push;
1547             }
1548             DLOG("OUT PUSH '}'\n");
1549             goto out;
1550         }
1551     } else {
1552         DLOG("DEC CURL NEST\n");
1553         --recover_curl_nest;
1554     }
1555     break;
1556 case TKN_EOF:
1557     if (recover_curl_nest)
1558         inc_curl_nest();
1559     if (recover_func_nest)
1560         inc_func_nest();
1561     goto out;
1562 default:
1563     break;
1564 }
1565 tt = yylex();
1566 };
1567 out:
1568 scanner_push_back_token();
1569 out_no_push:
1570 finish_recover();
1571 }
1572
1573 #undef DEBUG_TYPE
1574 #define DEBUG_TYPE parser-recover:parser-record-recover
1575 static void record_body_recover()
1576 {
1577     Parser-Token_Type tt = yylval.token.type;
1578     Int recover_func_nest = 0, recover_curl_nest = 0;
1579     DLOG("--- START TOKEN: %d\n", tt);
1580     while (tt) {
1581         DLOG("TOKEN VAL: %d\n", tt);
1582         switch (tt) {
1583             case TKN_IMPORT_KEY:
1584             case TKN_TYPE_KEY:
1585             case TKN_SCOLON_OP:
1586                 /* Fall through. */
1587             case TKN_COMMA_OP:
1588                 if (!recover_func_nest && !recover_curl_nest)
1589                     goto out;
1590                 break;
1591             case TKN_FUNC_KEY:
1592                 DLOG("INC FUNC NEST\n");
1593                 ++recover_func_nest;
1594                 break;
1595             case TKN_LCURLY_OP:
1596                 DLOG("INC CURL NEST\n");
1597                 ++recover_curl_nest;
1598                 break;

```

```

1599     case TKN_END_KEY:
1600         if (!recover_func_nest) {
1601             if (!recover_curl_nest) {
1602                 DLOG("END TOKEN BREAK\n");
1603                 goto out;
1604             }
1605         } else {
1606             DLOG("DEC FUNC NEST\n");
1607             --recover_func_nest;
1608         }
1609         break;
1610     case TKN_RCURLY_OP:
1611         DLOG("FOUND '}' CURL NEST IS CURRENTLY: %U\n", recover_curl_nest);
1612         if (!recover_curl_nest) {
1613             if (!recover_func_nest) {
1614                 if (!get_curl_nest()) {
1615                     DLOG("OUT NO PUSH '}'\n");
1616                     goto out_no_push;
1617                 }
1618                 DLOG("OUT PUSH '}'\n");
1619                 goto out;
1620             }
1621         } else {
1622             DLOG("DEC CURL NEST\n");
1623             --recover_curl_nest;
1624         }
1625         break;
1626     case TKN_EOF:
1627         if (recover_curl_nest)
1628             inc_curl_nest();
1629         if (recover_func_nest)
1630             inc_func_nest();
1631         goto out;
1632     default:
1633         break;
1634 }
1635 tt = yylex();
1636 };
1637 out:
1638     scanner_push_back_token();
1639 out_no_push:
1640     finish_recover();
1641 }
1642
1643 Ast *parser_parse(Const_String file)
1644 {
1645     current_ast = ast_alloc(file);
1646     current_file_name = file;
1647     last_err_line = 0;
1648     last_err_column = 0;
1649     extend_nest = 0;
1650     consistent_stmt_decl_top = 0;
1651
1652     scanner_restart();
1653     int ret = yyparse();
1654     (void)ret;
1655
1656     report_further_errors();
1657
1658     string_clear(expect_string);
1659
1660     assert(vector_is_empty(&curl_nest_stack) || ret);
1661     vector_clear(&curl_nest_stack);
1662
1663     assert(vector_stack_is_empty(&expr_stack) || ret);
1664     clear_vector_stack(&expr_stack);
1665
1666     assert(vector_stack_is_empty(&decl_stmt_stack) || ret);
1667     clear_vector_stack(&decl_stmt_stack);
1668
1669     assert(vector_stack_is_empty(&extend_stack) || ret);
1670     clear_vector_stack(&extend_stack);
1671
1672     assert(vector_is_empty(&extend_nest_stack) || ret);

```

```

1673     vector_clear(&extend_nest_stack);
1674
1675     assert(vector_stack_is_empty(&func_head_stack) || ret);
1676     clear_vector_stack(&func_head_stack);
1677
1678     return current_ast;
1679 }
1680
1681 static void create_expect_string(const char *str)
1682 {
1683     const char *nchr, *chr = strchr(str, EXPECTED_STRING_START);
1684     string_clear(expect_string);
1685     if (!chr)
1686         return; /* Should never happen. */
1687     /* Go past unexexpected token. */
1688     chr = strchr(chr + 1, EXPECTED_STRING_START);
1689     if (chr) {
1690         ++chr;
1691         string_append(expect_string, S(chr));
1692         string_search_replace_char(expect_string, EXPECTED_STRING_END, '\\0');
1693         chr = strchr(chr, EXPECTED_STRING_START);
1694         for (; chr; chr = nchr) {
1695             nchr = strchr(++chr, EXPECTED_STRING_START);
1696             if (nchr)
1697                 string_append_format(expect_string, S(", %s"), chr);
1698             else
1699                 string_append_format(expect_string, S(" or %s"), chr);
1700             string_search_replace_char(expect_string,
1701                                     EXPECTED_STRING_END, '\\0');
1702         }
1703     }
1704 }
1705
1706 void yyerror(const char *str)
1707 {
1708     last_err_line = yylval.token.lineno;
1709     last_err_column = yylval.token.startcol;
1710     last_err_text = yylval.token.text;
1711     create_expect_string(str);
1712 }

```

:

A.3.17 src/parser/scanner.l

```

1  %{
2
3  #include "parser.tab.h"
4  #include <parser.h>
5  #include <std_include.h>
6  #include <vector.h>
7
8  static Uns block_comment_nest;
9  static Uns column = 1;
10 static Uns eof_count;
11 static bool at_eof;
12
13 /* Used to free memory allocated for token text later on. */
14 static VECTOR(token_text_list);
15
16 static inline void newline_action()
17 {
18     column = 1;
19 }
20
21 static inline void text_read_action()
22 {
23     column += yyleng;
24 }
25
26 static Parser_Token_Type token_action(Parser_Token_Type token_type)

```

```

27 {
28     yylval.token.text = string_duplicate(S(yytext));
29     vector_append(&token_text_list, yylval.token.text);
30     yylval.token.type = token_type;
31     yylval.token.lineno = yylineno;
32     yylval.token.startcol = column;
33     text_read_action();
34     return token_type;
35 }
36
37 %}
38
39 %option noinput
40 %option nounput
41 %option noyywrap
42 %option yylineno
43 %option align
44
45 %s IN_COMMENT
46 %s IN_MODULE
47
48 ID      [a-zA-Z_][a-zA-Z_0-9]*
49 STR     \"(\\.\\.|[^\\\"\\n])*[\\\"\\n]
50 CHAR    \"'(\\.\\.|[^\\'\\n])*[\\'\\n]
51 MODULE  ({ID}\\.|\\.)*{ID}
52
53 %%
54
55 <INITIAL>{
56     \\(\\*      text_read_action(); ++block_comment_nest; BEGIN(IN_COMMENT);
57     \\#\\.\\n    newline_action();
58     [ \\t\\r]   text_read_action();
59     \\n        newline_action();
60
61     \\+        return token_action(TKN_PLUS_OP);
62     \\-        return token_action(TKN_MINUS_OP);
63     \\*        return token_action(TKN_MUL_OP);
64     \\/        return token_action(TKN_DIV_OP);
65     ==        return token_action(TKN_EQ_OP);
66     !=        return token_action(TKN_NEQ_OP);
67     \\>        return token_action(TKN_GT_OP);
68     \\<        return token_action(TKN_LT_OP);
69     \\>=       return token_action(TKN_GTEQ_OP);
70     \\<=       return token_action(TKN_LTEQ_OP);
71     &&        return token_action(TKN LAND_OP);
72     \\|\\|     return token_action(TKN_LOR_OP);
73     =         return token_action(TKN_ASSIGN_OP);
74     \\\\.       return token_action(TKN_DOT_OP);
75     ,         return token_action(TKN_COMMA_OP);
76     \\[        return token_action(TKN_LSQUARE_OP);
77     \\]        return token_action(TKN_RSQUARE_OP);
78     \\(        return token_action(TKN_LPAREN_OP);
79     \\)        return token_action(TKN_RPAREN_OP);
80     \\{        return token_action(TKN_LCURLY_OP);
81     \\}        return token_action(TKN_RCURLY_OP);
82     !         return token_action(TKN_BANG_OP);
83     \\|        return token_action(TKN_HLINE_OP);
84     :         return token_action(TKN_COLON_OP);
85     ;         return token_action(TKN_SCOLON_OP);
86
87     extern    return token_action(TKN_EXTERN_KEY);
88     func      return token_action(TKN_FUNC_KEY);
89     end       return token_action(TKN_END_KEY);
90     int       return token_action(TKN_INT_KEY);
91     void      return token_action(TKN_VOID_KEY);
92     bool      return token_action(TKN_BOOL_KEY);
93     char      return token_action(TKN_CHAR_KEY);
94     string    return token_action(TKN_STRING_KEY);
95     array     return token_action(TKN_ARRAY_KEY);
96     of        return token_action(TKN_OF_KEY);
97     record    return token_action(TKN_RECORD_KEY);
98     finalize  return token_action(TKN_FINALIZE_KEY);
99     type      return token_action(TKN_TYPE_KEY);
100    var       return token_action(TKN_VAR_KEY);

```

```

101     return token_action(TKN_RETURN_KEY);
102     write return token_action(TKN_WRITE_KEY);
103     if return token_action(TKN_IF_KEY);
104     allocate return token_action(TKN_ALLOCATE_KEY);
105     delete return token_action(TKN_DELETE_KEY);
106     then return token_action(TKN_THEN_KEY);
107     while return token_action(TKN_WHILE_KEY);
108     do return token_action(TKN_DO_KEY);
109     length return token_action(TKN_LENGTH_KEY);
110     else return token_action(TKN_ELSE_KEY);
111     true return token_action(TKN_TRUE_KEY);
112     false return token_action(TKN_FALSE_KEY);
113     null return token_action(TKN_NULL_KEY);
114     cast return token_action(TKN_CAST_KEY);
115     import BEGIN(IN_MODULE); return token_action(TKN_IMPORT_KEY);
116     package BEGIN(IN_MODULE); return token_action(TKN_PACKAGE_KEY);
117
118     [0-9]+ return token_action(TKN_INT_CONST);
119     {ID} return token_action(TKN_IDENTIFIER);
120
121     {CHAR} return token_action(TKN_CHAR_CONST);
122     {STR} return token_action(TKN_STRING_CONST);
123
124     . return token_action(TKN_UNEXPECTED);
125
126     <<eof>> {
127         if (at_eof) {
128             return 0;
129         } else {
130             at_eof = true;
131             return token_action(TKN_EOF);
132         }
133     }
134 }
135
136 <IN_COMMENT>{
137     \(\* text_read_action(); ++block_comment_nest;
138     \*\) text_read_action(); if (!--block_comment_nest) BEGIN(INITIAL);
139     \n newline_action();
140     . text_read_action();
141
142     <<eof>> {
143         if (at_eof) {
144             return 0;
145         } else {
146             at_eof = true;
147             if (!eof_count++)
148                 parser_report_error(yylineno, column, S(
149                     "unterminated block comment, "
150                     "missing " QFY("(") " " "\n"));
151             return token_action(TKN_EOF);
152         }
153     }
154 }
155
156 <IN_MODULE>{
157     [ \t\r] text_read_action();
158     \n newline_action();
159     {MODULE} BEGIN(INITIAL); return token_action(TKN_MODULE_CONST);
160     . BEGIN(INITIAL); return token_action(TKN_UNEXPECTED);
161     <<eof>> BEGIN(INITIAL); return token_action(TKN_MODULE_CONST);
162 }
163
164 %%
165
166 int yylex_destroy();
167 void yyrestart(FILE *in);
168
169 void scanner_push_back_token()
170 {
171     if (!yytext) {
172         --column;
173         at_eof = false;
174     } else {

```

```

175     if (!strcmp(yytext, "import"))
176         BEGIN(INITIAL);
177     column -= yyleng;
178     yyless(0);
179 }
180 }
181
182 void scanner_restart()
183 {
184     vector_for_each_destroy(&token_text_list,
185         (Vector_Destructor)string_destroy);
186     yylineno = 1;
187     column = 1;
188     block_comment_nest = 0;
189     eof_count = 0;
190     at_eof = false;
191     BEGIN(INITIAL);
192     yyrestart(yyin);
193 }
194
195 void scanner_finalize()
196 {
197     vector_for_each_destroy(&token_text_list,
198         (Vector_Destructor)string_destroy);
199     if (yylex_destroy())
200         fatal_error(S("error deallocating scanner memory.\n"));
201 }

```

A.4 Imports

:

A.4.1 src/ast/ast_visitor_import.c

```

1  #include "ast_visitor_import.h"
2  #include <import_handler.h>
3  #include "symbol_table.h"
4  #include <debug.h>
5  #include <main.h>
6
7  #undef DEBUG_TYPE
8  #define DEBUG_TYPE import
9
10 AST_VISITOR_STRUCT_BEGIN(Ast_Visitor_Import)
11     Ast *ast;
12     Symbol_Table *sym_table;
13     Symbol_Property next_property;
14     String prev_iden;
15     bool in_def_or_decl;
16     Hash_Map imported_hash;
17 AST_VISITOR_STRUCT_END(Ast_Visitor_Import)
18
19 static void unary_action(Ast_Visitor_Import *v, Ast_Node_Unary *n)
20 {
21     n->expr->accept_visitor(n->expr, AST_VISITOR_OF(v));
22 }
23
24 static void binary_action(Ast_Visitor_Import *v, Ast_Node_Binary *n)
25 {
26     n->lhs->accept_visitor(n->lhs, AST_VISITOR_OF(v));
27     n->rhs->accept_visitor(n->rhs, AST_VISITOR_OF(v));
28 }
29
30 #include <unistd.h>
31
32 static void set_project_root(Ast *ast)
33 {
34     Const_String package = ast_get_package(ast);

```

```

35     cd_working_dir();
36     Uns cnt = 0;
37
38     String tmp_str, cur_dir;
39     char *cstr = realpath(string_to_cstr(ast_get_dirname(ast)), NULL);
40     if (!cstr)
41         fatal_error(S("Unable to locate project root [%m]\n"));
42
43     cur_dir = string_alloc(S(cstr));
44     free_mem(cstr);
45     if (package) {
46         cnt = string_count(package, S("/"));
47         for (Uns i = 0; i < cnt+1; i++) {
48             tmp_str = string_dirname(cur_dir);
49             string_destroy(cur_dir);
50             cur_dir = tmp_str;
51         }
52     }
53
54     import_handler_set_project_root(cur_dir);
55     string_destroy(cur_dir);
56 }
57
58 static void func_call_import(Vector *args, Ast_Visitor *v)
59 {
60     Ast_Node *expr;
61     VECTOR_FOR_EACH_ENTRY(args, expr)
62         expr->accept_visitor(expr, v);
63 }
64
65 typedef struct Imported {
66     Symbol_Table_Node *node;
67     Const_String dep;
68     Hash_Map_Slot slot;
69 } Imported;
70
71 #define IMPORTED_OF_SLOT(s) CONTAINER_OF(s, Imported, slot)
72
73 static bool imported_hash_comparator(Imported *search, Hash_Map_Slot *slot)
74 {
75     Imported *imp = IMPORTED_OF_SLOT(slot);
76     if (search->node != imp->node)
77         return false;
78     if (string_compare(search->dep, imp->dep))
79         return false;
80     return true;
81 }
82
83 static bool imported_hash_insert(Hash_Map *map, Symbol_Table_Node *node,
84     Const_String dep)
85 {
86     Imported *imp = alloc_mem(sizeof(Imported));
87     imp->dep = dep;
88     imp->node = node;
89     Uns hash = string_hash_code(dep);
90     if (!hash_map_contains(map, imp, hash)) {
91         hash_map_insert(map, &imp->slot, hash);
92         return true;
93     } else {
94         free_mem(imp);
95         return false;
96     }
97 }
98
99 static void imported_hash_destructor(Hash_Map_Slot *slot)
100 {
101     free_mem(IMPORTED_OF_SLOT(slot));
102 }
103
104 ASTVF_BEGIN(AST_EXPR_LOR, Ast_Visitor_Import, v, Ast_Node_Binary, n)
105     binary_action(v, n);
106 ASTVF_END
107
108 ASTVF_BEGIN(AST_EXPR_LAND, Ast_Visitor_Import, v, Ast_Node_Binary, n)

```



```

109     binary_action(v, n);
110 ASTVF_END
111
112 ASTVF_BEGIN(AST_EXPR_EQ, Ast_Visitor_Import, v, Ast_Node_Binary, n)
113     binary_action(v, n);
114 ASTVF_END
115
116 ASTVF_BEGIN(AST_EXPR_NEQ, Ast_Visitor_Import, v, Ast_Node_Binary, n)
117     binary_action(v, n);
118 ASTVF_END
119
120 ASTVF_BEGIN(AST_EXPR_GT, Ast_Visitor_Import, v, Ast_Node_Binary, n)
121     binary_action(v, n);
122 ASTVF_END
123
124 ASTVF_BEGIN(AST_EXPR_LT, Ast_Visitor_Import, v, Ast_Node_Binary, n)
125     binary_action(v, n);
126 ASTVF_END
127
128 ASTVF_BEGIN(AST_EXPR_GTEQ, Ast_Visitor_Import, v, Ast_Node_Binary, n)
129     binary_action(v, n);
130 ASTVF_END
131
132 ASTVF_BEGIN(AST_EXPR_LTEQ, Ast_Visitor_Import, v, Ast_Node_Binary, n)
133     binary_action(v, n);
134 ASTVF_END
135
136 ASTVF_BEGIN(AST_EXPR_PLUS, Ast_Visitor_Import, v, Ast_Node_Binary, n)
137     binary_action(v, n);
138 ASTVF_END
139
140 ASTVF_BEGIN(AST_EXPR_MINUS, Ast_Visitor_Import, v, Ast_Node_Binary, n)
141     binary_action(v, n);
142 ASTVF_END
143
144 ASTVF_BEGIN(AST_EXPR_MUL, Ast_Visitor_Import, v, Ast_Node_Binary, n)
145     binary_action(v, n);
146 ASTVF_END
147
148 ASTVF_BEGIN(AST_EXPR_DIV, Ast_Visitor_Import, v, Ast_Node_Binary, n)
149     binary_action(v, n);
150 ASTVF_END
151
152 ASTVF_BEGIN(AST_EXPR_CAST, Ast_Visitor_Import, v, Ast_Node_Binary, n)
153     Symbol_Table_Node *node = ast_node_get_symbol_table_node(AST_NODE_OF(n));
154     Symbol_Property saved_property = v->next_property;
155
156     v->in_def_or_decl = true;
157     v->next_property = SYMBOL_PROPERTY_TYPE_DEF;
158     n->lhs->accept_visitor(n->lhs, AST_VISITOR_OF(v));
159     v->in_def_or_decl = false;
160
161     DLOG("Check update type of CAST '%S'\n", v->prev_iden);
162     v->next_property = SYMBOL_PROPERTY_VAR;
163     n->rhs->accept_visitor(n->rhs, AST_VISITOR_OF(v));
164     v->next_property = saved_property;
165
166     File_Location *loc = ast_node_get_file_location(AST_NODE_OF(n));
167     Symbol *symbol = symbol_table_get_from_location(node, loc);
168     assert(symbol);
169     import_table_update_symbol_type(false, ast_get_symbol_table(v->ast), node,
170     symbol->resolved_type, true);
171 ASTVF_END
172
173 ASTVF_BEGIN(AST_EXPR_LNOT, Ast_Visitor_Import, v, Ast_Node_Unary, n)
174     unary_action(v, n);
175 ASTVF_END
176
177 ASTVF_BEGIN(AST_EXPR_ABS, Ast_Visitor_Import, v, Ast_Node_Unary, n)
178     unary_action(v, n);
179 ASTVF_END
180
181 ASTVF_BEGIN(AST_EXPR_INT, Ast_Visitor_Import, v, Ast_Expr_Int, n)
182     (void) v; (void) n;

```

```

183 ASTVF_END
184
185 ASTVF_BEGIN(AST_EXPR_BOOL, Ast_Visitor_Import, v, Ast_Expr_Bool, n)
186     (void) v; (void) n;
187 ASTVF_END
188
189 ASTVF_BEGIN(AST_EXPR_NULL, Ast_Visitor_Import, v, Ast_Expr_Null, n)
190     (void) v; (void) n;
191 ASTVF_END
192
193 ASTVF_BEGIN(AST_VARIABLE_IDEN, Ast_Visitor_Import, v, Ast_Variable_Iden, n)
194     (void) v; (void) n;
195     Ast_Node *ast_node = AST_NODE_OF(n);
196     Symbol_Table_Node *node = ast_node_get_symbol_table_node(ast_node);
197     File_Location *location = ast_node_get_file_location(ast_node);
198     Vector *sym_vec;
199     Dependency_Symbols *dep_sym = NULL;
200     Symbol *sym;
201     String new_iden = NULL;
202     switch (v->next_property) {
203     case SYMBOL_PROPERTY_TYPE_DEF:
204         DLOG("%S - Found variable iden (type def): %S",
205             ast_get_file_name(v->ast), n->iden);
206         dep_sym = import_table_build_iden_loc(false, node,
207             v->next_property, n->iden, location);
208         break;
209     case SYMBOL_PROPERTY_VAR:
210         DLOG("%S - Found variable iden (var): %S",
211             ast_get_file_name(v->ast), n->iden);
212         dep_sym = import_table_build_iden_loc(false, node,
213             SYMBOL_PROPERTY_VAR, n->iden, location);
214         break;
215     case SYMBOL_PROPERTY_FUNC:
216         DLOG("%S - Found variable iden (func): %S",
217             ast_get_file_name(v->ast), n->iden);
218         dep_sym = import_table_build_iden_loc(false, node,
219             SYMBOL_PROPERTY_FUNC, n->iden, location);
220         break;
221     }
222     if (dep_sym) {
223         sym_vec = dep_sym->sym_vec;
224         if (sym_vec && vector_size(sym_vec) == 1) {
225             sym = vector_pop_last(sym_vec);
226             new_iden = string_duplicate(sym->unique_name);
227         }
228         assert(!sym_vec || vector_size(sym_vec) <= 1);
229         dependency_symbols_destroy(dep_sym);
230     }
231     if (new_iden) {
232         DLOG("\n\treplaced with '%S'\n", new_iden);
233         string_destroy(n->iden);
234         n->iden = new_iden;
235     } else {
236         DLOG("\n\t(ignored)\n");
237     }
238     v->prev_iden = n->iden;
239 ASTVF_END
240
241 ASTVF_BEGIN(AST_REC_SELF_PTR, Ast_Visitor_Import, v, Ast_Empty, n)
242     (void) v; (void) n;
243 ASTVF_END
244
245 ASTVF_BEGIN(AST_EXPR_ARY_REF, Ast_Visitor_Import, v, Ast_Node_Binary, n)
246     binary_action(v, n);
247 ASTVF_END
248
249 ASTVF_BEGIN(AST_EXPR_FUNC_CALL, Ast_Visitor_Import, v, Ast_Expr_Func_Call, n)
250     func_call_import(n->arguments, AST_VISITOR_OF(v));
251 ASTVF_END
252
253 ASTVF_BEGIN(AST_EXPR_DOT_REF, Ast_Visitor_Import, v, Ast_Node_Binary, n)
254     binary_action(v, n);
255 ASTVF_END
256

```

```

257 ASTVF_BEGIN(AST_EXPR_DIRECT_REF, Ast_Visitor_Import, v, Ast_Node_Binary, n)
258     if (n->lhs) {
259         Symbol_Table_Node *node = ast_node_get_symbol_table_node(AST_NODE_OF(n));
260         Symbol_Property saved_property = v->next_property;
261         v->next_property = SYMBOL_PROPERTY_TYPE_DEF;
262         n->lhs->accept_visitor(n->lhs, AST_VISITOR_OF(v));
263         v->next_property = SYMBOL_PROPERTY_VAR;
264
265         File_Location *loc = ast_node_get_file_location(AST_NODE_OF(n));
266         Symbol *symbol = symbol_table_get_from_location(node, loc);
267         assert(symbol);
268         import_table_update_symbol_type(false, ast_get_symbol_table(v->ast), node,
269             symbol->resolved_type, true);
270         v->next_property = saved_property;
271     }
272     n->rhs->accept_visitor(n->rhs, AST_VISITOR_OF(v));
273 ASTVF_END
274
275 ASTVF_BEGIN(AST_SIMPLE_TYPE_INT, Ast_Visitor_Import, v, Ast_Empty, n)
276     (void) v; (void) n;
277 ASTVF_END
278
279 ASTVF_BEGIN(AST_SIMPLE_TYPE_VOID, Ast_Visitor_Import, v, Ast_Empty, n)
280     (void) v; (void) n;
281 ASTVF_END
282
283 ASTVF_BEGIN(AST_SIMPLE_TYPE_BOOL, Ast_Visitor_Import, v, Ast_Empty, n)
284     (void) v; (void) n;
285 ASTVF_END
286
287 ASTVF_BEGIN(AST_TYPE_IDEN, Ast_Visitor_Import, v, Ast_Type_Iden, n)
288     DLOG("%S - TYPE_IDEN: %S", ast_get_file_name(v->ast), n->iden);
289     Ast_Node *ast_node = AST_NODE_OF(n);
290     Symbol_Table_Node *node = ast_node_get_symbol_table_node(ast_node);
291     File_Location *location = ast_node_get_file_location(ast_node);
292     Dependency_Symbols *dep_sym;
293     String new_iden = NULL;
294     dep_sym = import_table_build_iden_loc(false, node,
295         SYMBOL_PROPERTY_TYPE_DEF, n->iden, location);
296     if (dep_sym) {
297         Vector *sym_vec = dep_sym->sym_vec;
298         Symbol *sym;
299         if (sym_vec && vector_size(sym_vec) == 1) {
300             sym = vector_pop_last(sym_vec);
301             new_iden = string_duplicate(sym->unique_name);
302         }
303         assert(!sym_vec || vector_size(sym_vec) <= 1);
304         dependency_symbols_destroy(dep_sym);
305     }
306     if (new_iden) {
307         DLOG("\n\treplaced with " QFY("%S") "\n", new_iden);
308         string_destroy(n->iden);
309         n->iden = new_iden;
310     } else {
311         DLOG("\n\t(ignored)\n");
312     }
313     v->prev_iden = n->iden;
314 ASTVF_END
315
316 ASTVF_BEGIN(AST_VAR_DECL, Ast_Visitor_Import, v, Ast_Node_Binary, n)
317     Symbol_Property saved_property;
318     Symbol *symbol;
319     Symbol_Table_Node *node;
320     String iden;
321
322     node = ast_node_get_symbol_table_node(AST_NODE_OF(n));
323     saved_property = v->next_property;
324     v->next_property = SYMBOL_PROPERTY_VAR;
325     v->in_def_or_decl = true;
326     n->lhs->accept_visitor(n->lhs, AST_VISITOR_OF(v));
327     iden = v->prev_iden;
328     v->prev_iden = NULL;
329
330     v->next_property = SYMBOL_PROPERTY_TYPE_DEF;

```

```

331     n->rhs->accept_visitor(n->rhs, AST_VISITOR_OF(v));
332     v->next_property = saved_property;
333
334     DLOG("Check update type of variable '%S'\n", iden);
335     Uns hash = string_hash_code(iden);
336     symbol = symbol_table_node_get(node, iden, hash, SYMBOL_PROPERTY_VAR);
337     assert(symbol);
338     import_table_update_symbol_type(false, ast_get_symbol_table(v->ast), node,
339     symbol->resolved_type, true);
340     File_Location *loc = ast_node_get_file_location(AST_NODE_OF(n));
341
342     import_handler_search_dependencies(node, SYMBOL_PROPERTY_VAR, iden, hash,
343     loc, loc);
344
345     v->in_def_or_decl = false;
346 ASTVF_END
347
348 ASTVF_BEGIN(AST_TYPE_DEF, Ast_Visitor_Import, v, Ast_Node_Binary, n)
349     Symbol_Table_Node *node = ast_node_get_symbol_table_node(AST_NODE_OF(n));
350     Symbol_Property saved_property = v->next_property;
351     v->next_property = SYMBOL_PROPERTY_TYPE_DEF;
352     v->in_def_or_decl = true;
353     n->lhs->accept_visitor(n->lhs, AST_VISITOR_OF(v));
354     v->in_def_or_decl = false;
355     String iden = v->prev_iden;
356     v->prev_iden = NULL;
357     v->next_property = saved_property;
358
359     n->rhs->accept_visitor(n->rhs, AST_VISITOR_OF(v));
360
361     DLOG("Check update type of type def '%S'\n", iden);
362     Uns hash = string_hash_code(iden);
363     Symbol *symbol = symbol_table_node_get(node, iden, hash,
364     SYMBOL_PROPERTY_TYPE_DEF);
365     assert(symbol);
366     import_table_update_symbol_type(false, ast_get_symbol_table(v->ast), node,
367     symbol->resolved_type, true);
368     Double_List *dblist;
369     dblist = &TYPE_DEF_SYMBOL_OF_SYMBOL(symbol)->dbnode;
370     Double_List_Node *dbnode;
371     DOUBLE_LIST_FOR_EACH(dblist, dbnode) {
372         symbol = (Symbol *) TYPE_DEF_SYMBOL_OF_DBNODE(dbnode);
373         import_table_update_symbol_type(false, ast_get_symbol_table(v->ast),
374         node, symbol->resolved_type, true);
375     }
376 ASTVF_END
377
378 ASTVF_BEGIN(AST_TYPE_ARY, Ast_Visitor_Import, v, Ast_Type, n)
379     n->type->accept_visitor(n->type, AST_VISITOR_OF(v));
380 ASTVF_END
381
382 ASTVF_BEGIN(AST_TYPE_REC, Ast_Visitor_Import, v, Ast_Type_Rec, n)
383     Ast_Node *arg;
384     Vector *vargs = n->extend_list;
385     VECTOR_FOR_EACH_ENTRY(vargs, arg)
386         arg->accept_visitor(arg, AST_VISITOR_OF(v));
387     vargs = n->body;
388     VECTOR_FOR_EACH_ENTRY(vargs, arg)
389         arg->accept_visitor(arg, AST_VISITOR_OF(v));
390 ASTVF_END
391
392 static void stmt_list_action(Ast_Visitor_Import *v, Ast_Stmt_List *n)
393 {
394     Vector *statements;
395     Ast_Node *stmt;
396     statements = n->statements;
397     VECTOR_FOR_EACH_ENTRY(statements, stmt)
398         stmt->accept_visitor(stmt, AST_VISITOR_OF(v));
399 }
400
401 ASTVF_BEGIN(AST_STMT_LIST, Ast_Visitor_Import, v, Ast_Stmt_List, n)
402     stmt_list_action(v, n);
403 ASTVF_END
404

```

```

405 ASTVF_BEGIN(AST_FIN_STMT_LIST, Ast_Visitor_Import, v, Ast_Stmt_List, n)
406     stmt_list_action(v, n);
407 ASTVF_END
408
409 static void func_import(Ast_Visitor_Import *v, Ast_Func_Def *n)
410 {
411     Ast_Node *p;
412     Symbol *symbol;
413     Symbol_Table_Node *node = ast_node_get_symbol_table_node(AST_NODE_OF(n));
414     Symbol_Property saved_property = v->next_property;
415
416     /* Really no need to visit n->extern_type when it's != NULL */
417
418     v->in_def_or_decl = true;
419     v->next_property = SYMBOL_PROPERTY_FUNC;
420     n->iden->accept_visitor(n->iden, AST_VISITOR_OF(v));
421     DLOG("import lookup: %S\n", v->prev_iden);
422     Uns hash = string_hash_code(v->prev_iden);
423     symbol = symbol_table_node_get(node, v->prev_iden, hash,
424     SYMBOL_PROPERTY_FUNC);
425     assert(symbol);
426     import_table_update_symbol_type(false, ast_get_symbol_table(v->ast), node,
427     symbol->resolved_type, true);
428     v->next_property = saved_property;
429
430     n->return_type->accept_visitor(n->return_type, AST_VISITOR_OF(v));
431     v->in_def_or_decl = false;
432
433     Vector *vec = n->parameters;
434     VECTOR_FOR_EACH_ENTRY(vec, p)
435         p->accept_visitor(p, AST_VISITOR_OF(v));
436
437     if (n->statements)
438         n->statements->accept_visitor(n->statements, AST_VISITOR_OF(v));
439 }
440
441 ASTVF_BEGIN(AST_EXT_FUNC_DECL, Ast_Visitor_Import, v, Ast_Func_Def, n)
442     func_import(v, n);
443 ASTVF_END
444
445 ASTVF_BEGIN(AST_FIN_FUNC_DEF, Ast_Visitor_Import, v, Ast_Func_Def, n)
446     func_import(v, n);
447 ASTVF_END
448
449 ASTVF_BEGIN(AST_REC_FUNC_DEF, Ast_Visitor_Import, v, Ast_Func_Def, n)
450     func_import(v, n);
451 ASTVF_END
452
453 ASTVF_BEGIN(AST_FUNC_DEF, Ast_Visitor_Import, v, Ast_Func_Def, n)
454     func_import(v, n);
455 ASTVF_END
456
457 ASTVF_BEGIN(AST_IF_STMT, Ast_Visitor_Import, v, Ast_Node_Binary, n)
458     binary_action(v, n);
459 ASTVF_END
460
461 ASTVF_BEGIN(AST_IF_ELSE_STMT, Ast_Visitor_Import, v, Ast_Node_Ternary, n)
462     n->lhs->accept_visitor(n->lhs, AST_VISITOR_OF(v));
463     n->mid->accept_visitor(n->mid, AST_VISITOR_OF(v));
464     n->rhs->accept_visitor(n->rhs, AST_VISITOR_OF(v));
465 ASTVF_END
466
467 ASTVF_BEGIN(AST_ALLOC_ARY, Ast_Visitor_Import, v, Ast_Node_Binary, n)
468     binary_action(v, n);
469 ASTVF_END
470
471 ASTVF_BEGIN(AST_ALLOC_REC, Ast_Visitor_Import, v, Ast_Node_Unary, n)
472     unary_action(v, n);
473 ASTVF_END
474
475 ASTVF_BEGIN(AST_ALLOC_REC_CALL, Ast_Visitor_Import, v, Ast_Node_Binary, n)
476     binary_action(v, n);
477 ASTVF_END
478

```

```

479 ASTVF_BEGIN(AST_DELETE, Ast_Visitor_Import, v, Ast_Node_Unary, n)
480     unary_action(v, n);
481 ASTVF_END
482
483 ASTVF_BEGIN(AST_WHILE_STMT, Ast_Visitor_Import, v, Ast_Node_Binary, n)
484     binary_action(v, n);
485 ASTVF_END
486
487 ASTVF_BEGIN(AST_RETURN_STMT, Ast_Visitor_Import, v, Ast_Node_Unary, n)
488     unary_action(v, n);
489 ASTVF_END
490
491 ASTVF_BEGIN(AST_WRITE_STMT, Ast_Visitor_Import, v, Ast_Node_Unary, n)
492     assert(v->next_property == SYMBOL_PROPERTY_VAR);
493     unary_action(v, n);
494 ASTVF_END
495
496 ASTVF_BEGIN(AST_ASSIGNMENT, Ast_Visitor_Import, v, Ast_Node_Binary, n)
497     binary_action(v, n);
498 ASTVF_END
499
500 ASTVF_BEGIN(AST_EXPR_CHAR, Ast_Visitor_Import, v, Ast_Expr_Char, n)
501     (void) v; (void) n;
502 ASTVF_END
503
504 ASTVF_BEGIN(AST_EXPR_STRING, Ast_Visitor_Import, v, Ast_Expr_String, n)
505     (void) v; (void) n;
506 ASTVF_END
507
508 ASTVF_BEGIN(AST_IMPORT_STRING, Ast_Visitor_Import, v, Ast_Module_String, n)
509     Ast_Node *ast_node = AST_NODE_OF(n);
510     File_Location *loc = ast_node_get_file_location(ast_node);
511     Symbol_Table *table = ast_get_symbol_table(v->ast);
512     Symbol_Table_Node *node = ast_node_get_symbol_table_node(AST_NODE_OF(n));
513     String real_dep = vector_get(&node->import_dependencies, n->dep_idx);
514     if (imported_hash_insert(&v->imported_hash, node, real_dep))
515         import_handler_import(table, node, real_dep, loc);
516 ASTVF_END
517
518 ASTVF_BEGIN(AST_PACKAGE_STRING, Ast_Visitor_Import, v, Ast_Module_String, n)
519     (void) v; (void) n;
520 ASTVF_END
521
522 ASTVF_BEGIN(AST_SIMPLE_TYPE_CHAR, Ast_Visitor_Import, v, Ast_Empty, n)
523     (void) v; (void) n;
524 ASTVF_END
525
526 ASTVF_BEGIN(AST_SIMPLE_TYPE_STRING, Ast_Visitor_Import, v, Ast_Empty, n)
527     (void) v; (void) n;
528 ASTVF_END
529
530 ASTVF_BEGIN(AST_FUNC_BODY, Ast_Visitor_Import, v, Ast_Stmt_List, n)
531     Vector *stmt_list;
532     Ast_Node *stmt;
533     stmt_list = n->statements;
534     VECTOR_FOR_EACH_ENTRY(stmt_list, stmt)
535         stmt->accept_visitor(stmt, AST_VISITOR_OF(v));
536 ASTVF_END
537
538 bool ast_visitor_import_handle(Ast *ast, bool is_import)
539 {
540     String file_prefix = NULL;
541     char *cur_work_dir = NULL;
542     if (!is_import) {
543         if (!(cur_work_dir = getcwd(NULL, 0)))
544             fatal_error(S("unable to obtain current working directoy [%m]\n"));
545
546         ast_get_symbol_table(ast); // generate package name
547         set_project_root(ast);
548         import_handler_cd_project_root();
549
550         String file_name = string_basename(ast_get_file_name(ast));
551         Const_String package = ast_get_package(ast);
552         if (package)

```

```

553     file_prefix = string_dir_concat(ast_get_package(ast),
554     STRING_AFTER_LAST(file_name, '/'));
555     else
556     file_prefix = string_duplicate(file_name);
557     string_destroy(file_name);
558
559     string_replace_from(file_prefix, '.', S(""));
560     DLOG("Insert AST for %S\n", file_prefix);
561     import_handler_insert_ast(ast, file_prefix);
562 }
563
564 Ast_Node *root;
565 if (ast_is_valid(ast)) {
566     Ast_Visitor_Import import_visitor = {
567         .AST_VISITOR_FIELD = AST_VISITOR_STATIC_INIT(),
568         .next_property = SYMBOL_PROPERTY_VAR,
569         .in_def_or_decl = false,
570         .ast = ast,
571         .sym_table = ast_get_symbol_table(ast),
572         .imported_hash = HASH_MAP_INIT_SIZE(HASH_MAP_SIZE_5,
573         (Hash_Map_Comparator) imported_hash_comparator),
574     };
575
576     root = ast_get_root(ast);
577     root->accept_visitor(root, AST_VISITOR_OF(&import_visitor));
578     assert(import_visitor.next_property == SYMBOL_PROPERTY_VAR);
579     symbol_table_resolve(import_visitor.sym_table);
580     symbol_table_node_copy_hash_clear();
581
582     hash_map_for_each_destroy(&import_visitor.imported_hash,
583     (Hash_Map_Destructor) imported_hash_destructor);
584
585     if (!was_error_reported() && cmdopts.recursive_compile) {
586         import_handler_compile_imports();
587     }
588 }
589
590 if (!is_import) {
591     string_destroy(file_prefix);
592     assert(cur_work_dir);
593     if (chdir(cur_work_dir))
594         fatal_error(S("cannot switch back to directory %s [%m]\n"),
595         cur_work_dir);
596     free_mem(cur_work_dir);
597 }
598 return !import_handler_is_error_reported();
599 }

```

:

A.4.2 src/ast/ast_visitor_import.h

```

1 #ifndef AST_VISITOR_IMPORT_H
2 #define AST_VISITOR_IMPORT_H
3
4 #include "ast_visitor.h"
5
6 bool ast_visitor_import_handle(Ast *ast, bool is_import);
7
8 #endif // AST_VISITOR_IMPORT_H

```

:

A.4.3 src/import_handler.c

```

1 #include <stdio.h>
2 #include <import_handler.h>

```

```

3  #include <parser.h>
4  #include <hash_map.h>
5  #include <vector.h>
6  #include <pointer_hash.h>
7  #include <stdlib.h>
8  #include <errno.h>
9  #include <unistd.h>
10 #include <main.h>
11 #include <ast/ast_visitor_import.h>
12
13 #undef DEBUG_TYPE
14 #define DEBUG_TYPE import-handler
15
16 void import_table_destroy(Import_Table *it);
17
18 static void import_table_init_ast(Import_Table *it, File_Location *loc);
19
20 void import_table_dump_viti(Import_Table *it);
21
22 static void import_handler_merge_in(Import_Table *it);
23
24 void import_handler_merge_tables();
25
26 void import_table_viti_append(String_Builder *sb,
27                               Symbol_Type_Struct *type_struct);
28
29 static String project_root = NULL;
30
31 static Symbol_Table *import_merge_table;
32
33 bool import_handler_is_merge_table(Symbol_Table* t) {
34     return import_merge_table == t;
35 }
36
37 Symbol *import_handler_get_merge_sym(String iden, Symbol_Property property)
38 {
39     Symbol_Table_Node *node = symbol_table_get_root(import_merge_table);
40     assert(node);
41     return symbol_table_node_get(node, iden, string_hash_code(iden), property);
42 }
43
44 // hashmap containing all currently loaded interface tables.
45 struct Import_Table {
46     Hash_Map_Slot slot;
47     Const_String file_prefix;
48     String file_name;
49     Ast *ast;
50     Symbol_Table *cpy_sym_table;
51     bool merged_in;
52     bool is_viti;
53     bool is_compiled;
54     bool in_project;
55     bool is_symbols_updated;
56     bool is_main_file;
57 };
58
59 static bool import_handler_error_reported;
60
61 bool import_handler_is_error_reported()
62 {
63     return import_handler_error_reported;
64 }
65
66 #define IMPORT_TABLE_OF(s) CONTAINER_OF(s, Import_Table, slot)
67
68 bool import_hash_compare(void *search_obj, Hash_Map_Slot *map_slot)
69 {
70     Import_Table *it = IMPORT_TABLE_OF(map_slot);
71     return !string_compare(search_obj, it->file_prefix);
72 }
73
74 void import_table_destructor(Hash_Map_Slot *slot)
75 {
76     import_table_destroy(IMPORT_TABLE_OF(slot));

```



```

77 }
78
79 static HASH_MAP(import_hash, import_hash_compare);
80 // End of hash map.
81
82 Dependency_Symbols *dependency_symbols_alloc(Vector *sym_vec, Vector *dep_vec)
83 {
84     Dependency_Symbols *ret = ALLOC_NEW(Dependency_Symbols);
85     ret->sym_vec = sym_vec;
86     ret->dep_vec = dep_vec;
87     return ret;
88 }
89
90 void dependency_symbols_destroy(Dependency_Symbols *dep_sym)
91 {
92     if (dep_sym->sym_vec)
93         vector_destroy(dep_sym->sym_vec, NULL);
94     if (dep_sym->dep_vec)
95         vector_destroy(dep_sym->dep_vec, (Vector_Destructor) string_destroy);
96     free_mem(dep_sym);
97 }
98
99
100 void import_handler_set_project_root(Const_String root)
101 {
102     DLOG("set project root: %S\n", root);
103     string_destroy(project_root);
104     char *abs_path = realpath(string_to_cstr(root), NULL);
105     project_root = string_alloc(S(abs_path));
106     free_mem(abs_path);
107 }
108
109 void import_handler_cd_project_root()
110 {
111     if (project_root && chdir(string_to_cstr(project_root)))
112         fatal_error(S("unable to change working directory to " QFY("%S")
113             " [%m]\n"), project_root);
114     if (project_root)
115         DLOG("CD to project root: %S\n", project_root);
116 }
117
118 Import_Table *import_table_init(Const_String file_prefix, Uns hash)
119 {
120     Import_Table *it = ALLOC_NEW(Import_Table);
121     it->file_prefix = file_prefix;
122     it->file_name = NULL;
123     it->is_compiled = false;
124     it->in_project = true;
125     it->is_symbols_updated = false;
126     it->is_main_file = false;
127     it->is_viti = false;
128     it->merged_in = false;
129     it->ast = NULL;
130     it->cpy_sym_table = NULL;
131
132     assert(!hash_map_get(&import_hash, (void *)file_prefix, hash));
133
134     hash_map_insert(&import_hash, &it->slot, hash);
135
136     return it;
137 }
138
139 Import_Table *__import_table_get(Const_String file_prefix,
140     File_Location *loc)
141 {
142     Uns hash = string_hash_code(file_prefix);
143     Import_Table *it = IMPORT_TABLE_OF(hash_map_get(&import_hash,
144         (void *)file_prefix, hash));
145     if (it) {
146         goto out;
147     }
148
149     it = import_table_init(file_prefix, hash);
150

```

```

151     import_table_init_ast(it, loc);
152
153     if (cmdopts.generate_viti)
154         import_table_dump_viti(it);
155 out:
156     return it;
157 }
158
159 Import_Table *import_table_get(Const_String file_prefix, File_Location *loc)
160 {
161     Import_Table *ret;
162     int32_t init_num_errors = get_error_count();
163     ret = __import_table_get(file_prefix, loc);
164     import_handler_error_reported != init_num_errors != get_error_count();
165     return ret;
166 }
167
168 void import_table_destroy(Import_Table *it)
169 {
170     if (it->ast && !it->is_main_file) {
171         ast_destroy(it->ast);
172     }
173
174     string_destroy(it->file_name);
175
176     if (it->cpy_sym_table) {
177         symbol_table_destroy(it->cpy_sym_table);
178     }
179
180     free_mem(it);
181 }
182
183 Ast *import_symbol_table_parse(Const_String file_name, File_Location *loc)
184 {
185     int32_t err_count = get_error_count();
186     Ast *ast = parse(file_name);
187     if (!ast)
188         goto out;
189
190     if (!ast_is_valid(ast))
191         goto error_out;
192
193     ast_get_symbol_table(ast);
194
195     if (err_count == get_error_count()) {
196         Uns cnt = string_count(file_name, S("/"));
197         Const_String package = ast_get_package(ast);
198         if (package) {
199             char *abs_path = realpath(string_to_cstr(ast_get_dirname(ast)), NULL);
200             if (!string_ends_with(S(abs_path), package)) {
201                 free(abs_path);
202                 goto error_out;
203             }
204             free(abs_path);
205         } else if (!package && cnt > 0) {
206             Vector *split = string_split(file_name, S("."));
207             report_error_location(loc, S("import " QFY("%S") " does not "
208             "specify a package\n"), vector_peek_first(split));
209             vector_destroy(split, (Vector_Destructor) string_destroy);
210             goto error_out;
211         }
212         goto out;
213     }
214
215 error_out:
216     ast_destroy(ast);
217     ast = NULL;
218 out:
219     return ast;
220 }
221
222 static void import_table_include_imports(Import_Table *it)
223 {
224     Symbol_Table *sym_table = ast_get_symbol_table(it->ast);

```

```

225     Symbol_Table_Node *global_node = symbol_table_get_root(sym_table);
226     assert(global_node->type == SYMBOL_TABLE_NODE_GLOBAL);
227     Const_String dependency;
228     File_Location *loc;
229     for (Uns i = 0; i < vector_size(&global_node->import_dependencies); i++) {
230         dependency = vector_get(&global_node->import_dependencies, i);
231         loc = vector_get(&global_node->import_dependencies_loc, i);
232         import_table_get(dependency, loc);
233     }
234 }
235
236 static void search_viti(Import_Table *it, File_Location *loc)
237 {
238     Const_String file_prefix = it->file_prefix;
239     String file_name_viti = string_from_format(S("%S.viti"), file_prefix);
240
241     it->is_viti = true;
242     if (file_access_read(file_name_viti)) {
243         DLOG("Use " QFY("%S") "\n", file_name_viti);
244         it->ast = import_symbol_table_parse(file_name_viti, loc);
245         if (!ast_is_valid(it->ast))
246             goto err_out;
247         it->file_name = file_name_viti;
248         goto out;
249     }
250 err_out:
251     string_destroy(file_name_viti);
252 out:;
253
254
255 #if 0
256     String dependency, tmp_str;
257     Const_String file_prefix = it->file_prefix;
258     Symbol_Table *sym_table;
259     Symbol_Table_Node *tmp_node;
260     time_t this_time, other_time;
261     String file_name_viti = string_from_format(S("%S.viti"), file_prefix);
262     String file_name_vit = string_from_format(S("%S.vit"), file_prefix);
263
264     if (file_access_read(file_name_viti)) {
265         this_time = file_get_mtime(file_name_viti);
266         if (this_time == -1) {
267             fatal_error(S("unable to access modification time for file "
268                 QFY("%S") " [%m]\n"), file_name_viti);
269         }
270         other_time = file_get_mtime(file_name_vit);
271         if (this_time < other_time || other_time == -1) {
272             if (other_time == -1 && errno == ENOENT) {
273                 DLOG("Viti file " QFY("%S") " does not exist.\n",
274                     file_name_viti);
275                 goto out;
276             } else if (other_time == -1) {
277                 fatal_error(S("unable to access modification time for file "
278                     QFY("%S") " [%m]\n"), file_name_viti);
279             }
280             DLOG("Viti file " QFY("%S") " is older than corresponding .vit "
281                 "file. Use .vit file\n", file_name_viti);
282             goto out;
283         }
284     }
285
286     it->ast = import_symbol_table_parse(file_name_viti, loc);
287     if (!ast_is_valid(it->ast))
288         goto out;
289
290     sym_table = ast_get_symbol_table(it->ast);
291     tmp_node = symbol_table_get_root(sym_table);
292     VECTOR_FOR_EACH_ENTRY(&tmp_node->import_dependencies, dependency){
293         tmp_str = string_from_format(S("%S.vit"), dependency);
294         if (file_access_read(tmp_str)){
295             other_time = file_get_mtime(tmp_str);
296             if (other_time == -1) {
297                 fatal_error(S("unable to access modification time for "
298                     "file " QFY("%S") " [%m]\n"), tmp_str);
299             } else if (this_time < other_time) {

```

```

299         DLOG(QFY("%S") " outdated since " QFY("%S") " has been "
300             "updated\n", file_name_viti, tmp_str);
301         string_destroy(tmp_str);
302         goto err_out;
303     }
304     } else {
305         string_destroy(tmp_str);
306         goto err_out;
307     }
308     string_destroy(tmp_str);
309 }
310
311 DLOG("Viti file " QFY("%S") " is up to date.\n", file_name_viti);
312 it->is_viti = true;
313 it->file_name = file_name_viti;
314 goto out;
315 }
316
317 err_out:
318     ast_destroy(it->ast);
319     it->ast = NULL;
320     string_destroy(file_name_viti);
321 out:
322     string_destroy(file_name_vit);
323 #endif
324 }
325
326 static void search_vit(Import_Table *it, File_Location *loc)
327 {
328     Const_String file_prefix = it->file_prefix;
329     String file_name_vit = string_from_format(S("%S.vit"), file_prefix);
330
331     it->is_viti = false;
332     if (file_access_read(file_name_vit)) {
333         DLOG("Use " QFY("%S") "\n", file_name_vit);
334         it->ast = import_symbol_table_parse(file_name_vit, loc);
335         if (!ast_is_valid(it->ast))
336             goto err_out;
337         it->file_name = file_name_vit;
338         goto out;
339     }
340 err_out:
341     string_destroy(file_name_vit);
342 out:;
343 }
344
345 static void __import_table_init_ast(Import_Table *it, File_Location *loc)
346 {
347     if (cmdopts.recursive_compile) {
348         search_vit(it, loc);
349         if (!ast_is_valid(it->ast))
350             search_viti(it, loc);
351     } else {
352         search_viti(it, loc);
353         if (!ast_is_valid(it->ast))
354             search_vit(it, loc);
355     }
356     if (!ast_is_valid(it->ast)) {
357         it->ast = NULL;
358     } else {
359         /* Get all needed import files */
360         import_table_include_imports(it);
361     }
362 }
363
364
365
366 void import_table_init_ast(Import_Table *it, File_Location *loc)
367 {
368     /* Lookup in project root */
369     import_handler_cd_project_root();
370     DLOG("Init ast for: %S\n", it->file_prefix);
371     __import_table_init_ast(it, loc);
372 }

```

```

373  /* Else lookup in include paths */
374  if (!ast_is_valid(it->ast)) {
375      it->in_project = false;
376      Const_String path;
377      VECTOR_FOR_EACH_ENTRY(&cmdopts.import_search_paths, path){
378          DLOG("Search include path: %S\n", path);
379          assert(path);
380          if (chdir(string_to_cstr(path)))
381              fatal_error(S("unable to change working directory to "
382                  QFY("%S") " [%m]\n"), path);
383          __import_table_init_ast(it, loc);
384          if (ast_is_valid(it->ast)) {
385              break;
386          }
387      }
388  }
389
390  if (!ast_is_valid(it->ast)) {
391      report_error_location(loc, S("unable to import " QFY("%S") "\n"),
392          it->file_prefix);
393  } else {
394      it->cpy_sym_table = symbol_table_alloc(ast_get_file_name(it->ast));
395      symbol_table_merge(it->cpy_sym_table, ast_get_symbol_table(it->ast),
396          false);
397  }
398  }
399
400  static bool import_table_dependency_string_append(String_Builder *sb,
401      Import_Table *it)
402  {
403      bool ret = false;
404      String import_dependency, tmp_module_path;
405      if (ast_is_valid(it->ast)) {
406          Symbol_Table *sym_table = ast_get_symbol_table(it->ast);
407          Symbol_Table_Node *node = symbol_table_get_root(sym_table);
408          VECTOR_FOR_EACH_ENTRY(&node->import_dependencies, import_dependency) {
409              string_builder_append(sb, S("import "));
410              tmp_module_path = string_duplicate(import_dependency);
411              string_replace_all(tmp_module_path, '/', '.');
412              string_builder_append(sb, tmp_module_path);
413              string_destroy(tmp_module_path);
414              string_builder_append(sb, S(";\n"));
415              ret = true;
416          }
417      }
418      return ret;
419  }
420
421  static void __import_table_dump_viti(Import_Table *it)
422  {
423      Symbol_Table *sym_table = ast_get_symbol_table(it->ast);
424      Symbol_Table_Node *node = symbol_table_get_root(sym_table);
425
426      bool inserted_something = false;
427      String file_name = string_from_format(S("%S.viti"), it->file_prefix);
428      FILE *output_file = file_open(file_name, S("w"));
429      if (!output_file)
430          fatal_error(S("unable to open file " QFY("%S")
431              " for interface dump [%m]\n"),
432              file_name);
433
434      String_Builder sb = STRING_BUILDER_INIT();
435
436      string_builder_assign(&sb, S(""));
437      if (ast_get_package(it->ast)) {
438          string_builder_append(&sb, S("package "));
439          String tmp = string_duplicate(ast_get_package(it->ast));
440          string_replace_all(tmp, '/', '.');
441          string_builder_append(&sb, tmp);
442          string_destroy(tmp);
443          string_builder_append(&sb, S(";\n"));
444      }
445
446      inserted_something = import_table_dependency_string_append(&sb, it);

```

```

447
448     if (string_compare(S(""), string_builder_const_str(&sb)))
449         file_print_message(output_file, S("%S\n"),
450             string_builder_const_str(&sb));
451
452     inserted_something = false;
453     Symbol *tmp_sym;
454     Hash_Map_Slot *slot;
455     SYMBOL_TABLE_NODE_FOR_EACH_SYMBOL(node, slot, tmp_sym,
456         SYMBOL_PROPERTY_TYPE_DEF) {
457         string_builder_assign(&sb, S("type "));
458         string_builder_append(&sb, tmp_sym->identifier);
459         string_builder_append(&sb, S(" = "));
460         import_table_viti_append(&sb, tmp_sym->resolved_type);
461         string_builder_append(&sb, S(";\n"));
462         file_print_message(output_file, string_builder_const_str(&sb));
463         inserted_something = true;
464     }
465     if (inserted_something)
466         file_print_message(output_file, S("\n"));
467     inserted_something = false;
468
469     SYMBOL_TABLE_NODE_FOR_EACH_SYMBOL(node, slot, tmp_sym,
470         SYMBOL_PROPERTY_VAR) {
471         string_builder_assign(&sb, S("var "));
472         string_builder_append(&sb, tmp_sym->identifier);
473         string_builder_append(&sb, S(":"));
474         import_table_viti_append(&sb, tmp_sym->resolved_type);
475         string_builder_append(&sb, S(";\n"));
476         file_print_message(output_file, string_builder_const_str(&sb));
477         inserted_something = true;
478     }
479     if (inserted_something)
480         file_print_message(output_file, S("\n"));
481     inserted_something = false;
482
483     SYMBOL_TABLE_NODE_FOR_EACH_SYMBOL(node, slot, tmp_sym,
484         SYMBOL_PROPERTY_FUNC) {
485         Symbol_Type_Func *func = SYMBOL_TYPE_STRUCT_CONTAINER(
486             tmp_sym->resolved_type, Symbol_Type_Func);
487
488         if (func->is_extern_c)
489             string_builder_assign(&sb, S("extern(C) func "));
490         else
491             string_builder_assign(&sb, S("func "));
492
493         string_builder_append(&sb,
494             STRING_AFTER_LAST(tmp_sym->identifier, '.'));
495         import_table_viti_append(&sb, tmp_sym->resolved_type);
496
497         if (func->is_extern_c) {
498             string_builder_append(&sb, S(";\n"));
499         } else {
500             string_builder_append(&sb, S(" end "));
501             string_builder_append(&sb, STRING_AFTER_LAST(tmp_sym->identifier,
502                 '.'));
503             string_builder_append(&sb, S("\n"));
504         }
505
506         file_print_message(output_file, string_builder_const_str(&sb));
507     }
508
509     string_builder_clear(&sb);
510     file_close(output_file);
511     string_destroy(file_name);
512 }
513
514 void import_table_dump_viti(Import_Table *it)
515 {
516     if (!it->in_project)
517         return;
518
519     import_handler_cd_project_root();
520

```

```

521     __import_table_dump_viti(it);
522 }
523
524 static void import_table_update_func_sym_types(Import_Table *it,
525     Symbol_Table_Node *node)
526 {
527     Symbol_Func_Map *map;
528     Hash_Map_Slot *slot;
529     Symbol *sym;
530     HASH_MAP_FOR_EACH(&node->func_iden_map, slot) {
531         map = SYMBOL_FUNC_MAP_OF(slot);
532         VECTOR_FOR_EACH_ENTRY(&map->overload_idens, sym) {
533             import_table_update_symbol_type(true, it->cpy_sym_table,
534                 node, sym->resolved_type, false);
535         }
536     }
537 }
538 }
539
540 static void import_table_update_symbol_types(Import_Table *it,
541     Symbol_Table_Node *node, Symbol_Property prop)
542 {
543     if (prop == SYMBOL_PROPERTY_FUNC) {
544         import_table_update_func_sym_types(it, node);
545         return;
546     }
547
548     Hash_Map_Slot *tmp_slot;
549     Symbol *sym;
550     Symbol_Table *sym_table = it->cpy_sym_table;
551
552     SYMBOL_TABLE_NODE_FOR_EACH_SYMBOL(node, tmp_slot, sym, prop) {
553         import_table_update_symbol_type(true, sym_table, node,
554             sym->resolved_type, false);
555
556         if(prop == SYMBOL_PROPERTY_TYPE_DEF) {
557             Double_List *dblist;
558             dblist = &TYPE_DEF_SYMBOL_OF_SYMBOL(sym)->dbnode;
559             Double_List_Node *dbnode;
560             DOUBLE_LIST_FOR_EACH(dblist, dbnode) {
561                 sym = (Symbol *) TYPE_DEF_SYMBOL_OF_DBNODE(dbnode);
562                 import_table_update_symbol_type(true, sym_table, node,
563                     sym->resolved_type, false);
564             }
565         }
566     }
567 }
568
569 static void import_handler_merge_in(Import_Table *it)
570 {
571     Symbol_Table *sym_table = it->cpy_sym_table;
572
573     if (!it->is_symbols_updated) {
574         Symbol_Table_Node *tmp_node;
575         Double_List_Node *dbnode;
576         SYMBOL_TABLE_FOR_EACH_NODE(sym_table, tmp_node, dbnode) {
577             import_table_update_symbol_types(it, tmp_node,
578                 SYMBOL_PROPERTY_FUNC);
579             import_table_update_symbol_types(it, tmp_node,
580                 SYMBOL_PROPERTY_VAR);
581             import_table_update_symbol_types(it, tmp_node,
582                 SYMBOL_PROPERTY_TYPE_DEF);
583         }
584         it->is_symbols_updated = true;
585     }
586
587     symbol_table_merge(import_merge_table, sym_table, true);
588 }
589
590 void import_handler_merge_tables()
591 {
592     Hash_Map_Slot *slot;
593     Import_Table *it;
594     HASH_MAP_FOR_EACH(&import_hash, slot) {

```

```

595     it = IMPORT_TABLE_OF(slot);
596     if (ast_is_valid(it->ast) && !it->merged_in) {
597         DLOG("Merging in: %S\n", it->file_prefix);
598         import_handler_merge_in(it);
599         it->merged_in = true;
600     }
601 }
602 }
603
604 void import_handler_compile_imports()
605 {
606     Hash_Map_Slot *slot;
607     Import_Table *it;
608     import_handler_cd_project_root();
609     HASH_MAP_FOR_EACH(&import_hash, slot) {
610         it = IMPORT_TABLE_OF(slot);
611         if (ast_is_valid(it->ast) && !it->is_viti) {
612             if (it->in_project && !it->is_compiled &&
613                 cmdopts.recursive_compile) {
614                 it->is_compiled = true;
615                 String obj = __compile(it->ast, true);
616                 add_gen_object_file(obj);
617             }
618         }
619     }
620     cd_working_dir();
621 }
622
623 Dependency_Symbols *import_handler_search_dependencies(Symbol_Table_Node *node,
624     Symbol_Property property, String iden, Uns hash,
625     File_Location *local_loc, File_Location *err_loc UNUSED)
626 {
627     Symbol *sym = NULL, *sym_dup;
628     Import_Table *tmp_it;
629     Symbol_Table *tmp_table;
630     Symbol_Table_Node *tmp_node;
631     Vector *found_in_dep = NULL, *sym_vec = NULL;
632     String dependency;
633     Dependency_Symbols *ret = NULL;
634
635     if (!local_loc)
636         sym_vec = vector_alloc();
637
638     Uns n = vector_size(&node->import_dependencies);
639     if (n > 0)
640         found_in_dep = vector_alloc_size(n);
641
642     Const_String type_str;
643     switch (property) {
644     case SYMBOL_PROPERTY_TYPE_DEF:
645         type_str = S("type definitions");
646         break;
647     case SYMBOL_PROPERTY_VAR:
648         type_str = S("variable declarations");
649         break;
650     default:
651         return NULL;
652     }
653
654     VECTOR_FOR_EACH_ENTRY(&node->import_dependencies, dependency) {
655         tmp_it = __import_table_get(dependency, NULL);
656         if (ast_is_valid(tmp_it->ast)) {
657             tmp_table = tmp_it->cpy_sym_table;
658             tmp_node = symbol_table_get_root(tmp_table);
659             if (!local_loc && (!sym || property == SYMBOL_PROPERTY_TYPE_DEF)) {
660                 sym = symbol_table_node_get(tmp_node, iden, hash, property);
661                 if (sym) {
662                     String str = string_duplicate(dependency);
663                     string_replace_all(str, '/', '.');
664                     vector_append(found_in_dep, str);
665                     vector_append(sym_vec, sym);
666                 }
667             } else {
668                 sym_dup = symbol_table_node_get(tmp_node, iden, hash,

```



```

669         property);
670     if (sym_dup) {
671         String str = string_duplicate(dependency);
672         string_replace_all(str, '/', '.');
673         vector_append(found_in_dep, str);
674         sym_dup = NULL;
675     }
676 }
677 }
678 }
679
680 if (found_in_dep && vector_size(found_in_dep) > 0) {
681     String str;
682     if (!local_loc && vector_size(found_in_dep) > 1 &&
683         property != SYMBOL_PROPERTY_TYPE_DEF) {
684         str = string_from_format(S("ambiguous imported %S of " QFY("%S")
685             ", %1$S is imported from:\n"),
686             type_str, iden);
687         VECTOR_FOR_EACH_ENTRY(found_in_dep, dependency)
688             string_append_format(str, S("\t%S\n"), dependency);
689
690         report_error_location(err_loc, str);
691         string_destroy(str);
692     }
693 }
694
695 if (!local_loc) {
696     ret = dependency_symbols_alloc(sym_vec, found_in_dep);
697 } else if (found_in_dep) {
698     vector_destroy(found_in_dep, (Vector_Destructor) string_destroy);
699 }
700
701 return ret;
702 }
703
704 Dependency_Symbols *import_table_build_iden_loc(bool use_unique_name,
705     Symbol_Table_Node *node, Symbol_Property property, String iden,
706     File_Location *err_loc)
707 {
708     Symbol *local_sym;
709     Vector *sym_vec = NULL;
710     Uns hash = string_hash_code(iden);
711     Dependency_Symbols *ret;
712
713     do {
714         if (property == SYMBOL_PROPERTY_VAR)
715             local_sym = symbol_table_node_lookup_var(node, iden, err_loc);
716         else
717             local_sym = symbol_table_node_get(node, iden, hash, property);
718         if (local_sym) {
719             sym_vec = vector_alloc();
720             if (use_unique_name) {
721                 vector_append(sym_vec, local_sym);
722             }
723             ret = dependency_symbols_alloc(sym_vec, NULL);
724             break;
725         } else {
726             ret = import_handler_search_dependencies(node, property, iden,
727                 hash, NULL, err_loc);
728             if (ret) {
729                 if (vector_size(ret->sym_vec) > 0) {
730                     break;
731                 }
732                 dependency_symbols_destroy(ret);
733                 ret = NULL;
734             }
735         }
736     } while (node->parent && (node = node->parent));
737
738     return ret;
739 }
740
741 #if 0
742 Dependency_Symbols *import_table_build_iden(bool use_unique_name,

```

```

743     Symbol_Table_Node *node, Symbol_Property property, String iden)
744 {
745     File_Location loc = FILE_LOCATION_INIT(NULL, 0, 0);
746     return import_table_build_iden_loc(use_unique_name, node,
747         property, iden, &loc);
748 }
749 #endif
750
751 void import_table_update_symbol_type(bool use_unique_name, Symbol_Table *table,
752     Symbol_Table_Node *node, Symbol_Type_Struct *type_struct,
753     bool as_import)
754 {
755     Uns vec_size, idx;
756     Vector *tmp_vec;
757     Symbol *tmp_sym;
758     Symbol_Type_Struct *tmp_type;
759     Symbol_Type type = type_struct->methods->get_type();
760     Dependency_Symbols *dep_sym;
761
762     switch (type) {
763     case SYMBOL_TYPE_VOID:
764     case SYMBOL_TYPE_INT:
765     case SYMBOL_TYPE_BOOL:
766     case SYMBOL_TYPE_CHAR:
767     case SYMBOL_TYPE_STRING:
768     case SYMBOL_TYPE_CYCLE:
769     case SYMBOL_TYPE_UNKNOWN:
770         break;
771     case SYMBOL_TYPE_ARY:;
772         Symbol_Type_Ary *type_ary;
773         type_ary = SYMBOL_TYPE_STRUCT_CONTAINER(type_struct,
774             Symbol_Type_Ary);
775         if (type_ary->imp_table_updated)
776             break;
777         type_ary->imp_table_updated = true;
778
779         import_table_update_symbol_type(use_unique_name, table, node,
780             type_ary->ary_type, as_import);
781         break;
782     case SYMBOL_TYPE_REC:;
783         Symbol_Type_Rec *type_rec;
784         type_rec = SYMBOL_TYPE_STRUCT_CONTAINER(type_struct,
785             Symbol_Type_Rec);
786
787         if (type_rec->imp_table_updated)
788             break;
789         type_rec->imp_table_updated = true;
790
791         tmp_vec = &type_rec->extended_types;
792         vec_size = vector_size(tmp_vec);
793         for (idx = 0; idx < vec_size; idx++) {
794             tmp_type = vector_get(tmp_vec, idx);
795             import_table_update_symbol_type(use_unique_name, table, node,
796                 tmp_type, as_import);
797         }
798         tmp_vec = &type_rec->var_types;
799         vec_size = vector_size(tmp_vec);
800         for (idx = 0; idx < vec_size; idx++) {
801             tmp_type = vector_get(tmp_vec, idx);
802             import_table_update_symbol_type(use_unique_name, table, node,
803                 tmp_type, as_import);
804         }
805         tmp_vec = &type_rec->func_types;
806         vec_size = vector_size(tmp_vec);
807         for (idx = 0; idx < vec_size; idx++) {
808             tmp_type = vector_get(tmp_vec, idx);
809             import_table_update_symbol_type(use_unique_name, table, node,
810                 tmp_type, as_import);
811         }
812         break;
813     case SYMBOL_TYPE_FUNC:;
814         Symbol_Type_Func *func_type;
815         func_type = SYMBOL_TYPE_STRUCT_CONTAINER(type_struct,
816             Symbol_Type_Func);

```

```

817         if (func_type->imp_table_updated)
818             break;
819         func_type->imp_table_updated = true;
820
821         tmp_vec = &func_type->param_identifiers;
822         vec_size = vector_size(tmp_vec);
823         for (idx = 0; idx < vec_size; idx++) {
824             String par_iden = vector_get(tmp_vec, idx);
825             tmp_sym = symbol_table_node_get(func_type->body_sym_node,
826                 par_iden, string_hash_code(par_iden),
827                 SYMBOL_PROPERTY_VAR);
828             assert(tmp_sym);
829             import_table_update_symbol_type(use_unique_name, table,
830                 tmp_sym->sym_node, tmp_sym->resolved_type, as_import);
831         }
832
833         import_table_update_symbol_type(use_unique_name, table, node,
834             func_type->return_type, as_import);
835         break;
836     case SYMBOL_TYPE_IDEN:;
837         Symbol_Type_Iden *iden_type = SYMBOL_TYPE_STRUCT_CONTAINER(
838             type_struct, Symbol_Type_Iden);
839
840         dep_sym = import_table_build_iden_loc(use_unique_name,
841             iden_type->sym_node, SYMBOL_PROPERTY_TYPE_DEF,
842             iden_type->iden, iden_type->loc);
843
844         if (dep_sym && dep_sym->sym_vec) {
845             tmp_vec = dep_sym->sym_vec;
846             Uns size = vector_size(tmp_vec);
847
848             if (!dep_sym->dep_vec) {
849                 assert(size <= 1);
850                 if (size == 1) {
851                     tmp_sym = vector_peek_first(dep_sym->sym_vec);
852                     if (tmp_sym->sym_node->type !=
853                         SYMBOL_TABLE_NODE_GLOBAL) {
854                         dependency_symbols_destroy(dep_sym);
855                         break;
856                     }
857                 }
858             }
859
860             if (size == 1) {
861                 tmp_sym = vector_pop_last(tmp_vec);
862                 DLOG("update sigle iden: %S => %S\n", iden_type->iden,
863                     tmp_sym->unique_name);
864                 symbol_type_iden_set_iden(iden_type,
865                     string_duplicate(tmp_sym->unique_name));
866                 Symbol_Type_Struct *tstruct = tmp_sym->resolved_type;
867                 if (tstruct->methods->get_type() == SYMBOL_TYPE_REC) {
868                     Symbol_Type_Rec *trec =
869                         SYMBOL_TYPE_STRUCT_CONTAINER(tstruct,
870                             Symbol_Type_Rec);
871                     if (!trec->imp_table_name_updated) {
872                         string_destroy(trec->rec_name);
873                         trec->rec_name = string_duplicate(iden_type->iden);
874                         trec->imp_table_name_updated = true;
875                     }
876                 }
877             } else if (size > 1) {
878                 Uns hash = string_hash_code(iden_type->iden);
879                 /*
880                 String local_unique = string_from_format(S("%.%U.%S"),
881                     unique_import_no++, iden_type->iden);
882                 */
883                 String local_unique = string_duplicate(iden_type->iden);
884
885                 if (as_import && node->type != SYMBOL_TABLE_NODE_IMPORT) {
886                     if (!node->parent ||
887                         node->parent->type != SYMBOL_TABLE_NODE_IMPORT) {
888                         Symbol_Table_Node *tmp_node = node;
889                         node = symbol_table_node_alloc_insert(node->parent,
890                             table, SYMBOL_TABLE_NODE_IMPORT, NULL);

```

```

891         tmp_node->parent = node;
892     } else {
893         node = node->parent;
894     }
895 }
896
897 for (Uns i = 0; i < vector_size(tmp_vec); i++) {
898     tmp_sym = vector_get(tmp_vec, i);
899     Symbol_Type_Struct *ts = symbol_type_iden_alloc(table,
900         tmp_sym->unique_name);
901     Symbol_Type_Iden *tiden = SYMBOL_TYPE_STRUCT_CONTAINER(
902         ts, Symbol_Type_Iden);
903     tiden->sym_node = node;
904
905     DLOG("\tupdate iden: %S => %S\n", local_unique,
906         tiden->iden);
907
908     Symbol_Type_Struct *tstruct = tmp_sym->resolved_type;
909     if (tstruct->methods->get_type() == SYMBOL_TYPE_REC) {
910         Symbol_Type_Rec *trec =
911             SYMBOL_TYPE_STRUCT_CONTAINER(tstruct,
912                 Symbol_Type_Rec);
913         if (!trec->imp_table_name_updated) {
914             string_destroy(trec->rec_name);
915             trec->rec_name = string_duplicate(
916                 iden_type->iden);
917             trec->imp_table_name_updated = true;
918         }
919     }
920
921     if (as_import) {
922         __symbol_table_node_insert(node, local_unique,
923             vector_get(dep_sym->dep_vec, i), hash, ts,
924             SYMBOL_PROPERTY_TYPE_DEF,
925             iden_type->loc);
926     } else {
927         __symbol_table_node_insert(node, local_unique,
928             local_unique, hash, ts,
929             SYMBOL_PROPERTY_TYPE_DEF,
930             iden_type->loc);
931     }
932 }
933 symbol_type_iden_set_iden(iden_type, local_unique);
934 }
935 dependency_symbols_destroy(dep_sym);
936 }
937 break;
938 default:
939     assert(false);
940 }
941 }
942
943 void import_table_viti_append(String_Builder *sb,
944     Symbol_Type_Struct *type_struct)
945 {
946     Uns vec_size, tmp_vec_size, idx;
947     Vector *tmp_vec;
948     Symbol *tmp_sym;
949     Symbol_Type_Struct *tmp_type;
950     Symbol_Type type = type_struct->methods->get_type();
951     switch (type) {
952     case SYMBOL_TYPE_INT:
953     case SYMBOL_TYPE_BOOL:
954     case SYMBOL_TYPE_CHAR:
955     case SYMBOL_TYPE_STRING:
956     case SYMBOL_TYPE_CYCLE:
957     case SYMBOL_TYPE_VOID:
958     case SYMBOL_TYPE_UNKNOWN:
959         type_struct->methods->append_str(type_struct, sb);
960         break;
961     case SYMBOL_TYPE_ARY:;
962         Symbol_Type_Ary *type_ary;
963         type_ary = SYMBOL_TYPE_STRUCT_CONTAINER(type_struct,
964             Symbol_Type_Ary);

```

```

965     string_builder_append(sb, S("array of "));
966     import_table_viti_append(sb, type_ary->ary_type);
967     break;
968 case SYMBOL_TYPE_REC:
969     Symbol_Type_Rec *type_rec;
970     type_rec = SYMBOL_TYPE_STRUCT_CONTAINER(type_struct,
971         Symbol_Type_Rec);
972
973     string_builder_append(sb, S("record of "));
974     tmp_vec = &type_rec->extended_types;
975     vec_size = vector_size(&type_rec->extended_types);
976     if (vec_size) {
977         for (idx = 0; idx < vec_size-1; idx++) {
978             tmp_type = vector_get(tmp_vec, idx);
979             import_table_viti_append(sb, tmp_type);
980             string_builder_append(sb, S(", "));
981         }
982         tmp_type = vector_get(tmp_vec, idx);
983         import_table_viti_append(sb, tmp_type);
984         string_builder_append(sb, S(" "));
985     }
986
987     string_builder_append(sb, S("{ "));
988     tmp_vec = &type_rec->func_identifiers;
989     vec_size = vector_size(tmp_vec);
990     String fname;
991     if (vec_size) {
992         for (idx = 0; idx < vec_size-1; idx++) {
993             string_builder_append(sb, S("func "));
994             fname = vector_get(tmp_vec, idx);
995             string_builder_append(sb, STRING_AFTER_LAST(fname, '.'));
996             tmp_sym = symbol_table_node_get(type_rec->rec_sym_node,
997                 fname, string_hash_code(fname),
998                 SYMBOL_PROPERTY_FUNC);
999             import_table_viti_append(sb, tmp_sym->resolved_type);
1000             string_builder_append(sb, S(" end "));
1001             string_builder_append(sb, STRING_AFTER_LAST(fname, '.'));
1002             string_builder_append(sb, S(", "));
1003         }
1004         string_builder_append(sb, S("func "));
1005         fname = vector_get(tmp_vec, idx);
1006         string_builder_append(sb, STRING_AFTER_LAST(fname, '.'));
1007         tmp_sym = symbol_table_node_get(type_rec->rec_sym_node,
1008             fname, string_hash_code(fname), SYMBOL_PROPERTY_FUNC);
1009         import_table_viti_append(sb, tmp_sym->resolved_type);
1010         string_builder_append(sb, S(" end "));
1011         string_builder_append(sb, STRING_AFTER_LAST(fname, '.'));
1012     }
1013 }
1014
1015 tmp_vec = &type_rec->var_identifiers;
1016 tmp_vec_size = vector_size(tmp_vec);
1017 if (vec_size && tmp_vec_size)
1018     string_builder_append(sb, S(", "));
1019
1020 String var_name;
1021 vec_size = tmp_vec_size;
1022 if (vec_size) {
1023     for (idx = 0; idx < vec_size-1; idx++) {
1024         var_name = vector_get(tmp_vec, idx);
1025         string_builder_append(sb, var_name);
1026         string_builder_append(sb, S(":"));
1027         tmp_sym = symbol_table_node_get(type_rec->rec_sym_node,
1028             var_name, string_hash_code(var_name),
1029             SYMBOL_PROPERTY_VAR);
1030         import_table_viti_append(sb, tmp_sym->resolved_type);
1031         string_builder_append(sb, S(", "));
1032     }
1033     var_name = vector_get(tmp_vec, idx);
1034     string_builder_append(sb, var_name);
1035     string_builder_append(sb, S(":"));
1036     tmp_sym = symbol_table_node_get(type_rec->rec_sym_node,
1037         var_name, string_hash_code(var_name),
1038         SYMBOL_PROPERTY_VAR);

```

```

1039         import_table_viti_append(sb, tmp_sym->resolved_type);
1040     }
1041     string_builder_append(sb, S(" }"));
1042
1043     break;
1044 case SYMBOL_TYPE_FUNC::
1045     Symbol_Type_Func *func_type;
1046     func_type = SYMBOL_TYPE_STRUCT_CONTAINER(type_struct,
1047         Symbol_Type_Func);
1048     tmp_vec = &func_type->param_identifiers;
1049     vec_size = vector_size(tmp_vec);
1050     string_builder_append(sb, S("("));
1051     if (vec_size) {
1052         for (idx = 0; idx < vec_size-1; idx++) {
1053             string_builder_append(sb, vector_get(tmp_vec, idx));
1054             string_builder_append(sb, S(":"));
1055             tmp_sym = symbol_table_node_get(func_type->body_sym_node,
1056                 vector_get(tmp_vec, idx),
1057                 string_hash_code(vector_get(tmp_vec, idx)),
1058                 SYMBOL_PROPERTY_VAR);
1059             assert(tmp_sym);
1060             import_table_viti_append(sb, tmp_sym->resolved_type);
1061             string_builder_append(sb, S(", "));
1062         }
1063         string_builder_append(sb, vector_get(tmp_vec, idx));
1064         string_builder_append(sb, S(":"));
1065         tmp_sym = symbol_table_node_get(func_type->body_sym_node,
1066             vector_get(tmp_vec, idx),
1067             string_hash_code(vector_get(tmp_vec, idx)),
1068             SYMBOL_PROPERTY_VAR);
1069         assert(tmp_sym);
1070         import_table_viti_append(sb, tmp_sym->resolved_type);
1071     }
1072
1073     string_builder_append(sb, S(":"));
1074     import_table_viti_append(sb, func_type->return_type);
1075     break;
1076 case SYMBOL_TYPE_IDEN::
1077     Symbol_Type_Iden *type_iden = SYMBOL_TYPE_STRUCT_CONTAINER(
1078         type_struct, Symbol_Type_Iden);
1079     string_builder_append(sb, type_iden->iden);
1080     break;
1081 default:
1082     assert(false);
1083 }
1084 }
1085
1086 void import_handler_insert_ast(Ast* ast, Const_String file_prefix)
1087 {
1088     Uns hash = string_hash_code(file_prefix);
1089     Import_Table *it = import_table_init(file_prefix, hash);
1090
1091     it->is_main_file = true;
1092     it->is_compiled = true;
1093
1094     if (ast_is_valid(ast)) {
1095         it->ast = ast;
1096
1097         cd_working_dir();
1098
1099         it->cpy_sym_table = symbol_table_alloc(ast_get_file_name(it->ast));
1100         symbol_table_merge(it->cpy_sym_table, ast_get_symbol_table(it->ast),
1101             false);
1102
1103         import_table_include_imports(it);
1104
1105         if (cmdopts.generate_viti)
1106             import_table_dump_viti(it);
1107     }
1108 }
1109
1110 void import_handler_import(Symbol_Table *table, Symbol_Table_Node *local_node,
1111     Const_String file_prefix, File_Location *from_loc)
1112 {

```

```

1113     Import_Table *it;
1114     Symbol_Table *import_table;
1115     Symbol_Table_Node *import_node;
1116     Symbol_Table_Node *local_import_node;
1117
1118     char *cur_work_dir;
1119     if (!(cur_work_dir = getcwd(NULL, 0)))
1120         fatal_error(S("unable to obtain current working directoy [%m]\n"));
1121
1122     it = import_table_get(file_prefix, from_loc);
1123
1124     if (!symbol_table_get_root(import_merge_table))
1125         symbol_table_node_alloc_insert(NULL, import_merge_table,
1126             SYMBOL_TABLE_NODE_GLOBAL, NULL);
1127
1128     import_handler_merge_tables();
1129     symbol_table_dump_graph(import_merge_table, S("MERGE"));
1130     symbol_table_resolve(import_merge_table);
1131
1132     if (ast_is_valid(it->ast)) {
1133         import_table = ast_get_symbol_table(it->ast);
1134         import_node = symbol_table_get_root(import_table);
1135
1136         if (local_node->parent &&
1137             local_node->parent->type == SYMBOL_TABLE_NODE_IMPORT)
1138             local_import_node = local_node->parent;
1139         else {
1140             local_import_node = symbol_table_node_alloc_insert(
1141                 local_node->parent, table, SYMBOL_TABLE_NODE_IMPORT, NULL);
1142             local_node->parent = local_import_node;
1143         }
1144
1145         symbol_table_node_merge_cond(table, local_import_node,
1146             import_merge_table, NULL, it->file_prefix, import_node);
1147     }
1148
1149     if (chdir(cur_work_dir))
1150         fatal_error(S("cannot switch back to directory %s [%m]\n"),
1151             cur_work_dir);
1152     free_mem(cur_work_dir);
1153 }
1154
1155 void import_handler_init()
1156 {
1157     import_merge_table = symbol_table_alloc(NULL);
1158 }
1159
1160 void import_handler_clear()
1161 {
1162     if (project_root)
1163         string_destroy(project_root);
1164     project_root = NULL;
1165
1166     hash_map_for_each_destroy(&import_hash, import_table_destructor);
1167     symbol_table_destroy(import_merge_table);
1168     import_merge_table = NULL;
1169 }

```

:

A.4.4 src/import_handler.h

```

1  #ifndef IMPORT_HANDLER_H
2  #define IMPORT_HANDLER_H
3
4  #include <std_include.h>
5  #include <ast/symbol_table.h>
6  #include <ast/ast.h>
7
8  typedef struct Import_Table Import_Table;
9

```

```

10 typedef struct Dependency_Symbols {
11     Vector *sym_vec;
12     Vector *dep_vec;
13 } Dependency_Symbols;
14
15 Dependency_Symbols *dependency_symbols_alloc(Vector *sym_vec, Vector *dep_vec);
16
17 void dependency_symbols_destroy(Dependency_Symbols *dep_sym);
18
19 void import_handler_import(Symbol_Table *table, Symbol_Table_Node *local_node,
20     Const_String file_prefix, File_Location *from_loc);
21
22 bool import_handler_is_merge_table(Symbol_Table* t);
23
24 Import_Table *import_table_get(Const_String file_prefix, File_Location *loc);
25
26 Dependency_Symbols *import_table_build_iden(bool use_unique_name,
27     Symbol_Table_Node *node, Symbol_Property property, String iden);
28
29 Dependency_Symbols *import_table_build_iden_loc(bool use_unique_name,
30     Symbol_Table_Node *node, Symbol_Property property, String iden,
31     File_Location *err_loc);
32
33 Dependency_Symbols *import_handler_search_dependencies(Symbol_Table_Node *node,
34     Symbol_Property property, String iden, Uns hash,
35     File_Location *local_loc, File_Location *err_loc);
36
37 void import_table_update_symbol_type(bool use_unique_name, Symbol_Table *table,
38     Symbol_Table_Node *node, Symbol_Type_Struct *type_struct,
39     bool as_import);
40
41 Symbol *import_handler_get_merge_sym(String iden, Symbol_Property property);
42
43 void import_handler_compile_imports();
44
45 void import_handler_init();
46
47 void import_handler_clear();
48
49 void import_handler_insert_ast(Ast* ast, Const_String file_prefix);
50
51 /* Set the project root, searched before include paths */
52 void import_handler_set_project_root(Const_String path);
53
54 void import_handler_cd_project_root();
55
56 bool import_handler_is_error_reported();
57
58 #endif

```

A.5 Type Checking

:

A.5.1 src/ast/ast_visitor_type_check.c

```

1 #include "ast_visitor_type_check.h"
2 #include "symbol_table.h"
3 #include <main.h>
4
5 #undef DEBUG_TYPE
6 #define DEBUG_TYPE type-check
7
8 typedef enum Type_Check_Func_Type {
9     TT_FUNC_TYPE_NORMAL,
10     TT_FUNC_TYPE_FINALIZE,
11     TT_FUNC_TYPE_RECORD
12 } Type_Check_Func_Type;

```



```

13
14 AST_VISITOR_STRUCT_BEGIN(Ast_Visitor_Type_Check)
15     Symbol_Table *sym_table;
16     Symbol_Type_Struct *prev_sym_type;
17     Symbol_Func_Map *prev_func_map;
18     /* Variable is set in VARIABLE_IDEN and used by REC_ALLOCATE */
19     Symbol_Type_Struct *prev_variable_sym_type;
20     /* Variable is set in VARIABLE_IDEN and used by REC_ALLOCATE */
21     Symbol_Table_Node *rec_body_sym_node;
22     Symbol_Table_Node *dot_ref_sym_node;
23     /* Symbol_Type_Rec *lookup_rec; */
24     Symbol_Type_Struct *prev_direct_ref_rec;
25     Symbol_Type_Struct *prev_direct_ref_func;
26     Symbol_Property next_property;
27     String prev_iden;
28     Const_String prev_func_call_iden;
29     String curr_func_iden;
30     String prev_var_decl_iden;
31     Ast_Expr_Type prev_expr_type;
32     Ast_Expr_Type func_return_type;
33     Symbol_Type_Struct *func_return_sym;
34     Ast_Visitor_Method prev_method;
35     Uns stmt_list_nest;
36     Type_Check_Func_Type current_func_type;
37     bool lhs_of_assign_is_string_ref;
38     bool next_func_iden_is_decl;
39     bool in_beginning_of_ctor;
40     bool in_allocate_stmt;
41 AST_VISITOR_STRUCT_END(Ast_Visitor_Type_Check)
42
43 #define ASTVF_TC_BEGIN ASTVF_BEGIN
44
45 #define ASTVF_TC_RETURN(node_type, vis) \
46 do { \
47     (vis)->prev_method = AST_VISITOR_FUNC(node_type); \
48     return; \
49 } while (0)
50
51 #define ASTVF_TC_END(node_type, vis) ASTVF_TC_RETURN(node_type, vis); ASTVF_END
52
53 Ast_Expr_Type symbol_type_to_expr_type(Symbol_Type_Struct *st)
54 {
55     Ast_Expr_Type ret;
56     Symbol *tmp_sym;
57     Symbol_Type_Cycle *tmp_cycle;
58     Symbol_Type_Iden *tmp_iden;
59
60     try_again:
61     switch (st->methods->get_type()) {
62     case SYMBOL_TYPE_VOID:
63         ret = AST_EXPR_TYPE_VOID;
64         break;
65     case SYMBOL_TYPE_INT:
66         ret = AST_EXPR_TYPE_INT;
67         break;
68     case SYMBOL_TYPE_BOOL:
69         ret = AST_EXPR_TYPE_BOOL;
70         break;
71     case SYMBOL_TYPE_CHAR:
72         ret = AST_EXPR_TYPE_CHAR;
73         break;
74     case SYMBOL_TYPE_STRING:
75         ret = AST_EXPR_TYPE_STRING;
76         break;
77     case SYMBOL_TYPE_ARY:
78         ret = AST_EXPR_TYPE_ARY;
79         break;
80     case SYMBOL_TYPE_REC:
81         ret = AST_EXPR_TYPE_REC;
82         break;
83     case SYMBOL_TYPE_FUNC:
84         st = SYMBOL_TYPE_STRUCT_CONTAINER(st, Symbol_Type_Func)->return_type;
85         goto try_again;
86     case SYMBOL_TYPE_CYCLE:

```

```

87     tmp_cycle = SYMBOL_TYPE_STRUCT_CONTAINER(st, Symbol_Type_Cycle);
88     tmp_sym = symbol_table_node_lookup(tmp_cycle->sym_node,
89     tmp_cycle->name, SYMBOL_PROPERTY_TYPE_DEF);
90     assert(tmp_sym);
91     st = tmp_sym->resolved_type;
92     goto try_again;
93 case SYMBOL_TYPE_IDEN:
94     tmp_iden = SYMBOL_TYPE_STRUCT_CONTAINER(st, Symbol_Type_Iden);
95     fatal_error(S("unexpected unresolved symbol '%S'. Aborting...\n"),
96     tmp_iden->iden);
97     break;
98
99 default:
100     ret = AST_EXPR_TYPE_UNKNOWN;
101     break;
102 }
103 return ret;
104 }
105
106 static inline void error_not_compatible_types(Ast_Node *n,
107 Ast_Expr_Type lhs_t, Ast_Expr_Type rhs_t)
108 {
109     if (lhs_t != rhs_t)
110         report_error_location(ast_node_get_file_location(n),
111         S("type " QFY("%S") " is incompatible with " QFY("%S") "\n"),
112         ast_expr_type_to_string(lhs_t),
113         ast_expr_type_to_string(rhs_t));
114     else
115         report_error_location(ast_node_get_file_location(n),
116         S("incompatible " QFY("%S") " types\n"),
117         ast_expr_type_to_string(lhs_t));
118 }
119
120 static inline void warn_implicit_cast(File_Location *loc, Ast_Expr_Type from,
121 Ast_Expr_Type to)
122 {
123     if (cmdopts.warn_implicit_cast) {
124         report_warning_location(loc,
125         S("implicit cast from " QFY("%S") " to " QFY("%S") "\n"),
126         ast_expr_type_to_string(from),
127         ast_expr_type_to_string(to));
128     }
129 }
130
131 static inline void warn_implicit_expr_cast(Ast_Node *n, Ast_Expr_Type from,
132 Ast_Expr_Type to, const char *expr_type)
133 {
134     if (cmdopts.warn_implicit_cast) {
135         report_warning_location(ast_node_get_file_location(n),
136         S(QFY("%s") "-expression implicitly casted from " QFY("%S")
137         " to " QFY("%S") "\n"),
138         expr_type,
139         ast_expr_type_to_string(from),
140         ast_expr_type_to_string(to));
141     }
142 }
143
144 static inline void warn_implicit_cast_lhs(Ast_Node *n, Ast_Expr_Type from,
145 Ast_Expr_Type to, const char *operator)
146 {
147     if (cmdopts.warn_implicit_cast) {
148         report_warning_location(ast_node_get_file_location(n),
149         S("implicit cast from " QFY("%S") " to " QFY("%S")
150         " left of " QFY("%s") "\n"),
151         ast_expr_type_to_string(from),
152         ast_expr_type_to_string(to),
153         operator);
154     }
155 }
156
157 static inline void warn_implicit_cast_rhs(Ast_Node *n, Ast_Expr_Type from,
158 Ast_Expr_Type to, const char *operator)
159 {
160     if (cmdopts.warn_implicit_cast) {

```

```

161     report_warning_location(ast_node_get_file_location(n),
162         S("implicit cast from " QFY("%S") " to " QFY("%S")
163         " right of " QFY("%s") "\n"),
164         ast_expr_type_to_string(from),
165         ast_expr_type_to_string(to),
166         operator);
167     }
168 }
169
170 typedef enum Assign_Comatibility_Type {
171     ASSIGN_COMPATIBILITY_TRUE,
172     ASSIGN_COMPATIBILITY_FALSE,
173     ASSIGN_COMPATIBILITY_IMPLICIT,
174     ASSIGN_COMPATIBILITY_AMBIGUOUS
175 } Assign_Comatibility_Type;
176
177 static Assign_Comatibility_Type types_are_assignment_compatible(
178     Symbol_Type_Struct *dest,
179     Ast_Expr_Type dest_type,
180     Symbol_Type_Struct *src,
181     Ast_Expr_Type src_type)
182 {
183     if (dest_type == AST_EXPR_TYPE_UNKNOWN ||
184         src_type == AST_EXPR_TYPE_UNKNOWN)
185         return ASSIGN_COMPATIBILITY_TRUE;
186
187     if (dest_type != src_type) {
188         switch (dest_type) {
189             case AST_EXPR_TYPE_VOID:
190                 return ASSIGN_COMPATIBILITY_FALSE;
191
192             case AST_EXPR_TYPE_INT:
193                 switch (src_type) {
194                     case AST_EXPR_TYPE_CHAR:
195                         // return ASSIGN_COMPATIBILITY_IMPLICIT;
196                         return ASSIGN_COMPATIBILITY_TRUE;
197
198                     case AST_EXPR_TYPE_BOOL:
199                         return ASSIGN_COMPATIBILITY_IMPLICIT;
200
201                     default:
202                         return ASSIGN_COMPATIBILITY_FALSE;
203                 }
204
205             case AST_EXPR_TYPE_CHAR:
206                 switch (src_type) {
207                     case AST_EXPR_TYPE_INT:
208                         // return ASSIGN_COMPATIBILITY_IMPLICIT;
209                         return ASSIGN_COMPATIBILITY_TRUE;
210
211                     case AST_EXPR_TYPE_BOOL:
212                         return ASSIGN_COMPATIBILITY_IMPLICIT;
213
214                     default:
215                         return ASSIGN_COMPATIBILITY_FALSE;
216                 }
217
218             case AST_EXPR_TYPE_STRING:
219                 /* Fall through. */
220             case AST_EXPR_TYPE_ARY:
221                 /* Fall through. */
222             case AST_EXPR_TYPE_REC:
223                 if (src_type != AST_EXPR_TYPE_NULL)
224                     return ASSIGN_COMPATIBILITY_FALSE;
225                 return ASSIGN_COMPATIBILITY_TRUE;
226
227             case AST_EXPR_TYPE_BOOL:
228                 return ASSIGN_COMPATIBILITY_IMPLICIT;
229
230             default:
231                 return ASSIGN_COMPATIBILITY_FALSE;
232         }
233     }
234 }

```

```

235     switch (dest_type) {
236     case AST_EXPR_TYPE_REC:
237         if (symbol_type_rec_assignment_compatible(dest, src)) {
238             if (!symbol_type_rec_ambiguous_cast(dest, src))
239                 return ASSIGN_COMPATIBILITY_TRUE;
240             return ASSIGN_COMPATIBILITY_AMBIGUOUS;
241         }
242         return ASSIGN_COMPATIBILITY_FALSE;
243
244     case AST_EXPR_TYPE_ARY:
245         if (dest->methods->same_type(dest, src))
246             return ASSIGN_COMPATIBILITY_TRUE;
247         return ASSIGN_COMPATIBILITY_FALSE;
248
249     case AST_EXPR_TYPE_VOID:
250         return ASSIGN_COMPATIBILITY_FALSE;
251
252     default:
253         return ASSIGN_COMPATIBILITY_TRUE;
254     }
255 }
256
257 static inline NORETURN void unexpected_operand()
258 {
259     fatal_error(S("unexpected operand. Aborting...\n"));
260 }
261
262 static inline void invalid_bin_operands(Ast_Node *n,
263     Ast_Expr_Type lhs_t, Ast_Expr_Type rhs_t, const char *operator)
264 {
265     if (lhs_t != rhs_t)
266         report_error_location(ast_node_get_file_location(n),
267             S("incompatible " QFY("%S") " and " QFY("%S") " operands"
268             " for " QFY("%s") " operator\n"),
269             ast_expr_type_to_string(lhs_t),
270             ast_expr_type_to_string(rhs_t),
271             operator);
272     else
273         report_error_location(ast_node_get_file_location(n),
274             S("operator " QFY("%s") " is incompatible with "
275             QFY("%S") " operands\n"),
276             operator,
277             ast_expr_type_to_string(lhs_t));
278 }
279
280 static inline void invalid_ary_operands(Ast_Node *n, const char *operator)
281 {
282     report_error_location(ast_node_get_file_location(n),
283         S("incompatible array types for binary operator " QFY("%s") "\n"),
284         operator);
285 }
286
287 static void pure_logic_binop(Ast_Visitor_Type_Check *v, Ast_Node_Binary *n,
288     const char *operator)
289 {
290     Ast_Expr_Type lhs_t, rhs_t;
291     n->lhs->accept_visitor(n->lhs, AST_VISITOR_OF(v));
292     lhs_t = v->prev_expr_type;
293     n->rhs->accept_visitor(n->rhs, AST_VISITOR_OF(v));
294     rhs_t = v->prev_expr_type;
295
296     switch (lhs_t) {
297     case AST_EXPR_TYPE_UNKNOWN:
298         v->prev_expr_type = AST_EXPR_TYPE_UNKNOWN;
299         goto out;
300
301     case AST_EXPR_TYPE_BOOL:
302         break;
303
304     case AST_EXPR_TYPE_VOID:
305         invalid_bin_operands(AST_NODE_OF(n), lhs_t, rhs_t, operator);
306         v->prev_expr_type = AST_EXPR_TYPE_UNKNOWN;
307         goto out;
308

```

```

309     default:
310         warn_implicit_cast_lhs(AST_NODE_OF(n), lhs_t,
311             AST_EXPR_TYPE_BOOL, operator);
312         break;
313     }
314
315     switch (rhs_t) {
316     case AST_EXPR_TYPE_UNKNOWN:
317         v->prev_expr_type = AST_EXPR_TYPE_UNKNOWN;
318         goto out;
319
320     case AST_EXPR_TYPE_BOOL:
321         break;
322
323     case AST_EXPR_TYPE_VOID:
324         invalid_bin_operands(AST_NODE_OF(n), lhs_t, rhs_t, operator);
325         v->prev_expr_type = AST_EXPR_TYPE_UNKNOWN;
326         goto out;
327
328     default:
329         warn_implicit_cast_rhs(AST_NODE_OF(n), rhs_t,
330             AST_EXPR_TYPE_BOOL, operator);
331         break;
332     }
333
334     if (v->prev_expr_type != AST_EXPR_TYPE_UNKNOWN)
335         v->prev_expr_type = AST_EXPR_TYPE_BOOL;
336
337 out:
338     ast_expr_binary_set_expr_type(AST_EXPR_BINARY_OF(n), v->prev_expr_type);
339 }
340
341 static inline void invalid_una_operand(Ast_Node *n, Const_String op,
342     const char *operator)
343 {
344     report_error_location(ast_node_get_file_location(n),
345         S("incompatible " QFY("%S") " operand "
346           " for unary " QFY("%s") " operator\n"),
347         op, operator);
348 }
349
350 static inline void warn_from_to_una(Ast_Node *n, Ast_Expr_Type from,
351     Ast_Expr_Type to, const char *operator)
352 {
353     if (cmdopts.warn_implicit_cast)
354         report_warning_location(ast_node_get_file_location(n),
355             S("implicit cast from " QFY("%S") " to " QFY("%S")
356               " for unary " QFY("%s") " operator\n"),
357             ast_expr_type_to_string(from),
358             ast_expr_type_to_string(to),
359             operator);
360 }
361
362 static void equality_binop(Ast_Visitor_Type_Check *v, Ast_Node_Binary *n,
363     const char *operator)
364 {
365     Ast_Expr_Type lhs_t, rhs_t;
366     n->lhs->accept_visitor(n->lhs, AST_VISITOR_OF(v));
367     lhs_t = v->prev_expr_type;
368     n->rhs->accept_visitor(n->rhs, AST_VISITOR_OF(v));
369     rhs_t = v->prev_expr_type;
370
371     if (lhs_t == AST_EXPR_TYPE_UNKNOWN || rhs_t == AST_EXPR_TYPE_UNKNOWN) {
372         v->prev_expr_type = AST_EXPR_TYPE_UNKNOWN;
373         goto out;
374     }
375     if (lhs_t == rhs_t)
376         goto out_bool;
377
378     switch (lhs_t) {
379     case AST_EXPR_TYPE_INT:
380         /* Fall through. */
381     case AST_EXPR_TYPE_CHAR:
382         switch (rhs_t) {

```

```

383     case AST_EXPR_TYPE_INT:
384         /* Fall through. */
385     case AST_EXPR_TYPE_CHAR:
386         break;
387     case AST_EXPR_TYPE_BOOL:
388         warn_implicit_cast(ast_node_get_file_location(n->rhs),
389             rhs_t, lhs_t);
390         break;
391     default:
392         invalid_bin_operands(AST_NODE_OF(n), lhs_t, rhs_t, operator);
393         v->prev_expr_type = AST_EXPR_TYPE_UNKNOWN;
394         goto out;
395     }
396     break;
397
398 case AST_EXPR_TYPE_BOOL:
399     switch (rhs_t) {
400     case AST_EXPR_TYPE_INT:
401         /* Fall through. */
402     case AST_EXPR_TYPE_CHAR:
403         warn_implicit_cast(ast_node_get_file_location(n->lhs),
404             lhs_t, rhs_t);
405         break;
406     case AST_EXPR_TYPE_BOOL:
407         break;
408     default:
409         invalid_bin_operands(AST_NODE_OF(n), lhs_t, rhs_t, operator);
410         v->prev_expr_type = AST_EXPR_TYPE_UNKNOWN;
411         goto out;
412     }
413     break;
414
415 case AST_EXPR_TYPE_STRING:
416     /* Fall through. */
417 case AST_EXPR_TYPE_REC:
418     /* Fall through. */
419 case AST_EXPR_TYPE_ARY:
420     switch (rhs_t) {
421     case AST_EXPR_TYPE_NULL:
422         /* Fall. */
423     case AST_EXPR_TYPE_STRING:
424         /* Fall. */
425     case AST_EXPR_TYPE_ARY:
426         /* Fall. */
427     case AST_EXPR_TYPE_REC:
428         break;
429     default:
430         invalid_bin_operands(AST_NODE_OF(n), lhs_t, rhs_t, operator);
431         v->prev_expr_type = AST_EXPR_TYPE_UNKNOWN;
432         goto out;
433     }
434     break;
435
436 case AST_EXPR_TYPE_NULL:
437     switch (rhs_t) {
438     case AST_EXPR_TYPE_VOID:
439         invalid_bin_operands(AST_NODE_OF(n), lhs_t, rhs_t, operator);
440         v->prev_expr_type = AST_EXPR_TYPE_UNKNOWN;
441         goto out;
442     }
443
444     case AST_EXPR_TYPE_INT:
445         /* Fall through. */
446     case AST_EXPR_TYPE_BOOL:
447         /* Fall through. */
448     case AST_EXPR_TYPE_CHAR:
449         invalid_bin_operands(AST_NODE_OF(n), lhs_t, rhs_t, operator);
450         v->prev_expr_type = AST_EXPR_TYPE_UNKNOWN;
451         goto out;
452
453     case AST_EXPR_TYPE_STRING:
454         /* Fall through. */
455     case AST_EXPR_TYPE_REC:
456         /* Fall through. */

```

```

457         case AST_EXPR_TYPE_ARY:
458             break;
459
460         default:
461             unexpected_operand();
462     }
463
464     case AST_EXPR_TYPE_VOID:
465         invalid_bin_operands(AST_NODE_OF(n), lhs_t, rhs_t, operator);
466         v->prev_expr_type = AST_EXPR_TYPE_UNKNOWN;
467         goto out;
468
469     default:
470         unexpected_operand();
471 }
472
473 out_bool:
474     v->prev_expr_type = AST_EXPR_TYPE_BOOL;
475
476 out:
477     ast_expr_binary_set_expr_type(AST_EXPR_BINARY_OF(n), v->prev_expr_type);
478 }
479
480 static void compare_binop(Ast_Visitor_Type_Check *v, Ast_Node_Binary *n,
481                          const char *operator)
482 {
483     Ast_Expr_Type lhs_t, rhs_t;
484     n->lhs->accept_visitor(n->lhs, AST_VISITOR_OF(v));
485     lhs_t = v->prev_expr_type;
486     n->rhs->accept_visitor(n->rhs, AST_VISITOR_OF(v));
487     rhs_t = v->prev_expr_type;
488
489     if (lhs_t == AST_EXPR_TYPE_UNKNOWN || rhs_t == AST_EXPR_TYPE_UNKNOWN) {
490         v->prev_expr_type = AST_EXPR_TYPE_UNKNOWN;
491         goto out;
492     }
493
494     switch (lhs_t) {
495     case AST_EXPR_TYPE_INT:
496         /* Fall through. */
497     case AST_EXPR_TYPE_CHAR:
498         switch (rhs_t) {
499         case AST_EXPR_TYPE_INT:
500             /* Fall through. */
501         case AST_EXPR_TYPE_CHAR:
502             break;
503         case AST_EXPR_TYPE_BOOL:
504             warn_implicit_cast(ast_node_get_file_location(n->rhs),
505                               rhs_t, lhs_t);
506             break;
507         default:
508             invalid_bin_operands(AST_NODE_OF(n), lhs_t, rhs_t, operator);
509             v->prev_expr_type = AST_EXPR_TYPE_UNKNOWN;
510             goto out;
511         }
512         break;
513
514     case AST_EXPR_TYPE_VOID:
515         invalid_bin_operands(AST_NODE_OF(n), lhs_t, rhs_t, operator);
516         v->prev_expr_type = AST_EXPR_TYPE_UNKNOWN;
517         goto out;
518
519     case AST_EXPR_TYPE_BOOL:
520         switch (rhs_t) {
521         case AST_EXPR_TYPE_INT:
522             /* Fall through. */
523         case AST_EXPR_TYPE_CHAR:
524             warn_implicit_cast(ast_node_get_file_location(n->lhs),
525                               lhs_t, rhs_t);
526             break;
527         case AST_EXPR_TYPE_BOOL:
528             break;
529
530         default:

```

```

531         invalid_bin_operands(AST_NODE_OF(n), lhs_t, rhs_t, operator);
532         v->prev_expr_type = AST_EXPR_TYPE_UNKNOWN;
533         goto out;
534     }
535     break;
536
537     case AST_EXPR_TYPE_STRING:
538         /* Fall through. */
539     case AST_EXPR_TYPE_REC:
540         /* Fall through. */
541     case AST_EXPR_TYPE_ARY:
542         /* Fall through. */
543     case AST_EXPR_TYPE_NULL:
544         switch (rhs_t) {
545             case AST_EXPR_TYPE_STRING:
546                 /* Fall through. */
547             case AST_EXPR_TYPE_REC:
548                 /* Fall through. */
549             case AST_EXPR_TYPE_ARY:
550                 /* Fall through. */
551             case AST_EXPR_TYPE_NULL:
552                 if (cmdopts.warn_ref_compare) {
553                     if (lhs_t != rhs_t) {
554                         report_warning_location(
555                             ast_node_get_file_location(AST_NODE_OF(n)),
556                             S(QFY("%s")"-comparison between " QFY("%S")
557                                " and " QFY("%S") " reference operands\n"),
558                             operator,
559                             ast_expr_type_to_string(lhs_t),
560                             ast_expr_type_to_string(rhs_t));
561                     } else {
562                         report_warning_location(
563                             ast_node_get_file_location(AST_NODE_OF(n)),
564                             S(QFY("%s")"-comparison between " QFY("%S")
565                                " reference operands\n"),
566                             operator,
567                             ast_expr_type_to_string(lhs_t),
568                             ast_expr_type_to_string(rhs_t));
569                     }
570                 }
571             break;
572
573         default:
574             invalid_bin_operands(AST_NODE_OF(n), lhs_t, rhs_t, operator);
575             v->prev_expr_type = AST_EXPR_TYPE_UNKNOWN;
576             goto out;
577         }
578     break;
579
580     default:
581         unexpected_operand();
582     }
583
584     v->prev_expr_type = AST_EXPR_TYPE_BOOL;
585
586 out:
587     ast_expr_binary_set_expr_type(AST_EXPR_BINARY_OF(n), v->prev_expr_type);
588 }
589
590
591 static void arith_binop(Ast_Visitor_Type_Check *v, Ast_Node_Binary *n,
592                        const char *operator, bool is_ary_operator UNUSED)
593 {
594     //Symbol_Type_Struct *lhs_s, *rhs_s;
595     Ast_Expr_Type lhs_t, rhs_t;
596     n->lhs->accept_visitor(n->lhs, AST_VISITOR_OF(v));
597     //lhs_s = v->prev_sym_type;
598     lhs_t = v->prev_expr_type;
599     n->rhs->accept_visitor(n->rhs, AST_VISITOR_OF(v));
600     //rhs_s = v->prev_sym_type;
601     rhs_t = v->prev_expr_type;
602
603     if (lhs_t == AST_EXPR_TYPE_UNKNOWN || rhs_t == AST_EXPR_TYPE_UNKNOWN) {
604         v->prev_expr_type = AST_EXPR_TYPE_UNKNOWN;

```



```

605     goto out;
606 }
607
608 switch (lhs_t) {
609 case AST_EXPR_TYPE_INT:
610     switch (rhs_t) {
611     case AST_EXPR_TYPE_INT:
612         /* Fall through. */
613     case AST_EXPR_TYPE_CHAR:
614         break;
615     case AST_EXPR_TYPE_BOOL:
616         warn_implicit_cast(ast_node_get_file_location(n->rhs),
617             rhs_t, AST_EXPR_TYPE_INT);
618         break;
619     default:
620         invalid_bin_operands(AST_NODE_OF(n), lhs_t, rhs_t, operator);
621         v->prev_expr_type = AST_EXPR_TYPE_UNKNOWN;
622         goto out;
623     }
624     v->prev_expr_type = AST_EXPR_TYPE_INT;
625     goto out;
626
627 case AST_EXPR_TYPE_CHAR:
628     switch (rhs_t) {
629     case AST_EXPR_TYPE_INT:
630         /* Fall through. */
631     case AST_EXPR_TYPE_CHAR:
632         break;
633     case AST_EXPR_TYPE_BOOL:
634         warn_implicit_cast(ast_node_get_file_location(n->rhs),
635             rhs_t, AST_EXPR_TYPE_INT);
636         break;
637     default:
638         invalid_bin_operands(AST_NODE_OF(n), lhs_t, rhs_t, operator);
639         v->prev_expr_type = AST_EXPR_TYPE_UNKNOWN;
640         goto out;
641     }
642     v->prev_expr_type = AST_EXPR_TYPE_INT;
643     goto out;
644
645 case AST_EXPR_TYPE_ARY:
646     /* Fall. */
647 case AST_EXPR_TYPE_STRING:
648     invalid_bin_operands(AST_NODE_OF(n), lhs_t, rhs_t, operator);
649     v->prev_expr_type = AST_EXPR_TYPE_UNKNOWN;
650     goto out;
651
652 case AST_EXPR_TYPE_BOOL:
653     switch (rhs_t) {
654     case AST_EXPR_TYPE_INT:
655         /* Fall through. */
656     case AST_EXPR_TYPE_CHAR:
657         warn_implicit_cast(ast_node_get_file_location(n->lhs),
658             lhs_t, AST_EXPR_TYPE_INT);
659         break;
660     case AST_EXPR_TYPE_BOOL:
661         if (cmdopts.warn_implicit_cast) {
662             report_warning_location(
663                 ast_node_get_file_location(AST_NODE_OF(n)),
664                 S( QFY("bool") " operands for " QFY("%s") " operator "
665                     "implicitly casted to " QFY("int") "\n"),
666                 operator);
667         }
668         break;
669     default:
670         invalid_bin_operands(AST_NODE_OF(n), lhs_t, rhs_t, operator);
671         v->prev_expr_type = AST_EXPR_TYPE_UNKNOWN;
672         goto out;
673     }
674     v->prev_expr_type = AST_EXPR_TYPE_INT;
675     break;
676
677 case AST_EXPR_TYPE_VOID:

```

```

679     /* Fall through. */
680     case AST_EXPR_TYPE_REC:
681     /* Fall through. */
682     case AST_EXPR_TYPE_NULL:
683         invalid_bin_operands(AST_NODE_OF(n), lhs_t, rhs_t, operator);
684         v->prev_expr_type = AST_EXPR_TYPE_UNKNOWN;
685         goto out;
686
687     default:
688         unexpected_operand();
689     }
690
691 out:
692     ast_expr_binary_set_expr_type(AST_EXPR_BINARY_OF(n), v->prev_expr_type);
693 }
694
695 static void pure_logic_unaop(Ast_Visitor_Type_Check *v, Ast_Node_Unary *n,
696                             const char *operator)
697 {
698     Ast_Expr_Type t;
699     n->expr->accept_visitor(n->expr, AST_VISITOR_OF(v));
700     t = v->prev_expr_type;
701
702     if (t == AST_EXPR_TYPE_UNKNOWN)
703         goto out;
704
705     if (t == AST_EXPR_TYPE_VOID) {
706         invalid_una_operand(AST_NODE_OF(n),
707                             ast_expr_type_to_string(AST_EXPR_TYPE_VOID), operator);
708         v->prev_expr_type = AST_EXPR_TYPE_UNKNOWN;
709         goto out;
710     }
711
712     if (t != AST_EXPR_TYPE_BOOL)
713         warn_from_to_una(AST_NODE_OF(n), t, AST_EXPR_TYPE_BOOL, operator);
714
715     v->prev_expr_type = AST_EXPR_TYPE_BOOL;
716
717 out:
718     ast_expr_unary_set_expr_type(AST_EXPR_UNARY_OF(n), v->prev_expr_type);
719 }
720
721 ASTVF_TC_BEGIN(AST_EXPR_LOR, Ast_Visitor_Type_Check, v, Ast_Node_Binary, n)
722     pure_logic_binop(v, n, "||");
723 ASTVF_TC_END(AST_EXPR_LOR, v)
724
725 ASTVF_TC_BEGIN(AST_EXPR_LAND, Ast_Visitor_Type_Check, v, Ast_Node_Binary, n)
726     pure_logic_binop(v, n, "&&");
727 ASTVF_TC_END(AST_EXPR_LAND, v)
728
729 ASTVF_TC_BEGIN(AST_EXPR_EQ, Ast_Visitor_Type_Check, v, Ast_Node_Binary, n)
730     equality_binop(v, n, "==");
731 ASTVF_TC_END(AST_EXPR_EQ, v)
732
733 ASTVF_TC_BEGIN(AST_EXPR_NEQ, Ast_Visitor_Type_Check, v, Ast_Node_Binary, n)
734     equality_binop(v, n, "!=");
735 ASTVF_TC_END(AST_EXPR_NEQ, v)
736
737 ASTVF_TC_BEGIN(AST_EXPR_GT, Ast_Visitor_Type_Check, v, Ast_Node_Binary, n)
738     compare_binop(v, n, ">");
739 ASTVF_TC_END(AST_EXPR_GT, v)
740
741 ASTVF_TC_BEGIN(AST_EXPR_LT, Ast_Visitor_Type_Check, v, Ast_Node_Binary, n)
742     compare_binop(v, n, "<");
743 ASTVF_TC_END(AST_EXPR_LT, v)
744
745 ASTVF_TC_BEGIN(AST_EXPR_GTEQ, Ast_Visitor_Type_Check, v, Ast_Node_Binary, n)
746     compare_binop(v, n, ">=");
747 ASTVF_TC_END(AST_EXPR_GTEQ, v)
748
749 ASTVF_TC_BEGIN(AST_EXPR_LTEQ, Ast_Visitor_Type_Check, v, Ast_Node_Binary, n)
750     compare_binop(v, n, "<=");
751 ASTVF_TC_END(AST_EXPR_LTEQ, v)
752

```

```

753 ASTVF_TC_BEGIN(AST_EXPR_PLUS, Ast_Visitor_Type_Check, v, Ast_Node_Binary, n)
754     arith_binop(v, n, "+", false);
755 ASTVF_TC_END(AST_EXPR_PLUS, v)
756
757 ASTVF_TC_BEGIN(AST_EXPR_MINUS, Ast_Visitor_Type_Check, v, Ast_Node_Binary, n)
758     arith_binop(v, n, "-", false);
759 ASTVF_TC_END(AST_EXPR_MINUS, v)
760
761 ASTVF_TC_BEGIN(AST_EXPR_MUL, Ast_Visitor_Type_Check, v, Ast_Node_Binary, n)
762     arith_binop(v, n, "*", false);
763 ASTVF_TC_END(AST_EXPR_MUL, v)
764
765 ASTVF_TC_BEGIN(AST_EXPR_DIV, Ast_Visitor_Type_Check, v, Ast_Node_Binary, n)
766     arith_binop(v, n, "/", false);
767 ASTVF_TC_END(AST_EXPR_DIV, v)
768
769 static void invalid_cast_from_to(Ast_Node *n, Ast_Expr_Type lhs_t,
770     Ast_Expr_Type rhs_t, Ast_Visitor_Type_Check *v)
771 {
772     report_error_location(ast_node_get_file_location(n),
773         S("invalid cast from " QFY("%S") " to " QFY("%S") "\n"),
774         ast_expr_type_to_string(lhs_t),
775         ast_expr_type_to_string(rhs_t));
776     v->prev_sym_type = symbol_type_unknown_alloc(v->sym_table);
777 }
778
779 static void invalid_cast_between(Ast_Node *n, Ast_Expr_Type t,
780     Ast_Visitor_Type_Check *v)
781 {
782     report_error_location(ast_node_get_file_location(n),
783         S("invalid cast between " QFY("%S") " types\n"),
784         ast_expr_type_to_string(t));
785     v->prev_sym_type = symbol_type_unknown_alloc(v->sym_table);
786 }
787
788 ASTVF_TC_BEGIN(AST_EXPR_CAST, Ast_Visitor_Type_Check, v, Ast_Node_Binary, n)
789     Ast_Expr_Type lhs_t, rhs_t;
790     Symbol_Type_Struct *lhs_s, *rhs_s;
791
792     Symbol_Property saved_property = v->next_property;
793     v->next_property = SYMBOL_PROPERTY_TYPE_DEF;
794     n->lhs->accept_visitor(n->lhs, AST_VISITOR_OF(v));
795     v->next_property = saved_property;
796
797     Symbol *lhs_sym = symbol_table_get_from_location(
798         AST_NODE_OF(n)->sym_table_node,
799         ast_node_get_file_location(AST_NODE_OF(n)));
800
801     lhs_s = lhs_sym->resolved_type;
802     lhs_t = symbol_type_to_expr_type(lhs_s);
803
804     n->rhs->accept_visitor(n->rhs, AST_VISITOR_OF(v));
805     rhs_t = v->prev_expr_type;
806     rhs_s = v->prev_sym_type;
807
808     switch (lhs_t) {
809     case AST_EXPR_TYPE_UNKNOWN:
810         v->prev_sym_type = symbol_type_unknown_alloc(v->sym_table);
811         break;
812
813     case AST_EXPR_TYPE_BOOL:
814         if (rhs_t == AST_EXPR_TYPE_VOID) {
815             v->prev_sym_type = symbol_type_unknown_alloc(v->sym_table);
816             invalid_cast_from_to(n->lhs, rhs_t, lhs_t, v);
817         } else if (rhs_t == AST_EXPR_TYPE_UNKNOWN) {
818             v->prev_sym_type = symbol_type_unknown_alloc(v->sym_table);
819         } else {
820             v->prev_sym_type = lhs_s;
821         }
822         break;
823
824     case AST_EXPR_TYPE_INT:
825         /* Fall through. */
826     case AST_EXPR_TYPE_CHAR:

```

```

827     switch (rhs_t) {
828     case AST_EXPR_TYPE_UNKNOWN:
829         v->prev_sym_type = symbol_type_unknown_alloc(v->sym_table);
830         break;
831
832     case AST_EXPR_TYPE_INT:
833         /* Fall through. */
834     case AST_EXPR_TYPE_CHAR:
835         /* Fall through. */
836     case AST_EXPR_TYPE_BOOL:
837         v->prev_expr_type = lhs_t;
838         goto out;
839
840     default:
841         invalid_cast_from_to(n->lhs, rhs_t, lhs_t, v);
842         break;
843     }
844     break;
845
846 case AST_EXPR_TYPE_STRING:
847     if (rhs_t == AST_EXPR_TYPE_UNKNOWN) {
848         v->prev_sym_type = symbol_type_unknown_alloc(v->sym_table);
849         break;
850     }
851     if (rhs_t == AST_EXPR_TYPE_ARY) {
852         Symbol_Type_Ary *rhs_ary = SYMBOL_TYPE_STRUCT_CONTAINER(rhs_s,
853             Symbol_Type_Ary);
854         if (rhs_ary->ary_type->methods->get_type() != SYMBOL_TYPE_CHAR) {
855             Const_String ary_str = ast_expr_type_to_string(
856                 symbol_type_to_expr_type(rhs_ary->ary_type));
857             report_error_location(ast_node_get_file_location(n->rhs),
858                 S("invalid type cast from " QFY("array of %S")
859                   " to " QFY("string") "\n"),
860                 ary_str);
861             v->prev_sym_type = symbol_type_unknown_alloc(v->sym_table);
862             break;
863         } else {
864             v->prev_expr_type = AST_EXPR_TYPE_STRING;
865             goto out;
866         }
867     } else if (rhs_t == AST_EXPR_TYPE_STRING ||
868         rhs_t == AST_EXPR_TYPE_NULL) {
869         v->prev_expr_type = AST_EXPR_TYPE_STRING;
870         goto out;
871     } else {
872         invalid_cast_from_to(n->lhs, rhs_t, lhs_t, v);
873     }
874     break;
875
876 case AST_EXPR_TYPE_REC:
877     if (rhs_t == AST_EXPR_TYPE_UNKNOWN) {
878         v->prev_sym_type = symbol_type_unknown_alloc(v->sym_table);
879         break;
880     }
881     if (rhs_t == AST_EXPR_TYPE_NULL) {
882         v->prev_sym_type = lhs_s;
883         v->prev_expr_type = lhs_t;
884         goto out;
885     } else if (rhs_t != AST_EXPR_TYPE_REC) {
886         invalid_cast_from_to(n->lhs, rhs_t, lhs_t, v);
887     } else {
888         if (symbol_type_rec_cast_compatible(rhs_s, lhs_s)) {
889             if (!symbol_type_rec_ambiguous_cast(lhs_s, rhs_s)) {
890                 v->prev_sym_type = lhs_s;
891                 v->prev_expr_type = lhs_t;
892                 goto out;
893             } else {
894                 v->prev_sym_type = symbol_type_unknown_alloc(v->sym_table);
895                 report_error_location(
896                     ast_node_get_file_location(n->lhs),
897                     S("Ambiguous cast, "
898                       "type casted record is directly or "
899                       "indirectly extending multiple records "
900                       "with that structure\n"));

```

```

901         break;
902     }
903 }
904     invalid_cast_between(n->lhs, lhs_t, v);
905 }
906 break;
907
908 case AST_EXPR_TYPE_ARY:
909     if (rhs_t == AST_EXPR_TYPE_UNKNOWN) {
910         v->prev_sym_type = symbol_type_unknown_alloc(v->sym_table);
911         break;
912     }
913     if (rhs_t == AST_EXPR_TYPE_NULL) {
914         v->prev_expr_type = lhs_t;
915         v->prev_sym_type = lhs_s;
916         goto out;
917     } else if (rhs_t != AST_EXPR_TYPE_ARY) {
918         invalid_cast_from_to(n->lhs, rhs_t, lhs_t, v);
919     } else {
920         if (!lhs_s->methods->same_type(lhs_s, rhs_s)) {
921             invalid_cast_between(n->lhs, lhs_t, v);
922         } else {
923             v->prev_expr_type = lhs_t;
924             v->prev_sym_type = lhs_s;
925             goto out;
926         }
927     }
928     break;
929
930 default:
931     invalid_cast_from_to(n->lhs, rhs_t, lhs_t, v);
932     break;
933 }
934
935 v->prev_expr_type = symbol_type_to_expr_type(v->prev_sym_type);
936
937 out:
938     ast_expr_binary_set_expr_type(AST_EXPR_BINARY_OF(n), v->prev_expr_type);
939 ASTVF_TC_END(AST_EXPR_CAST, v)
940
941 ASTVF_TC_BEGIN(AST_EXPR_LNOT, Ast_Visitor_Type_Check, v, Ast_Node_Unary, n)
942     pure_logic_unaop(v, n, "!");
943 ASTVF_TC_END(AST_EXPR_LNOT, v)
944
945 ASTVF_TC_BEGIN(AST_EXPR_ABS, Ast_Visitor_Type_Check, v, Ast_Node_Unary, n)
946     Ast_Expr_Type t;
947     n->expr->accept_visitor(n->expr, AST_VISITOR_OF(v));
948     t = v->prev_expr_type;
949
950     switch (t) {
951     case AST_EXPR_TYPE_UNKNOWN:
952         /* Fall through. */
953     case AST_EXPR_TYPE_INT:
954         /* Fall through. */
955     case AST_EXPR_TYPE_CHAR:
956         break;
957     case AST_EXPR_TYPE_BOOL:
958         if (cmdopts.warn_implicit_cast)
959             report_warning_location(
960                 ast_node_get_file_location(AST_NODE_OF(n)),
961                 S( QFY("bool") " operand for " QFY("|") " operator "
962                   "implicitly casted to " QFY("int") "\n"));
963         break;
964
965     case AST_EXPR_TYPE_VOID:
966         /* Fall through. */
967     case AST_EXPR_TYPE_STRING:
968         /* Fall through. */
969     case AST_EXPR_TYPE_REC:
970         /* Fall through. */
971     case AST_EXPR_TYPE_NULL:
972         v->prev_expr_type = AST_EXPR_TYPE_UNKNOWN;
973         report_error_location(ast_node_get_file_location(AST_NODE_OF(n)),
974             S("invalid " QFY("%S") " expression as " QFY("|") ));

```

```

975         " argument\n"),
976         ast_expr_type_to_string(t));
977     break;
978
979     case AST_EXPR_TYPE_ARY:
980         v->prev_expr_type = AST_EXPR_TYPE_INT;
981         break;
982
983     default:
984         fatal_error(S("unexpected operand for abs operator. Aborting...\n"));
985     }
986
987     ast_expr_unary_set_expr_type(AST_EXPR_UNARY_OF(n), v->prev_expr_type);
988     ASTVF_TC_END(AST_EXPR_ABS, v)
989
990     ASTVF_TC_BEGIN(AST_EXPR_INT, Ast_Visitor_Type_Check, v, Ast_Expr_Int, n)
991     (void)n;
992     v->prev_expr_type = AST_EXPR_TYPE_INT;
993     ASTVF_TC_END(AST_EXPR_INT, v)
994
995     ASTVF_TC_BEGIN(AST_EXPR_BOOL, Ast_Visitor_Type_Check, v, Ast_Expr_Bool, n)
996     (void)n;
997     v->prev_expr_type = AST_EXPR_TYPE_BOOL;
998     ASTVF_TC_END(AST_EXPR_BOOL, v)
999
1000    ASTVF_TC_BEGIN(AST_EXPR_NULL, Ast_Visitor_Type_Check, v, Ast_Expr_Null, n)
1001    (void)n;
1002    v->prev_expr_type = AST_EXPR_TYPE_NULL;
1003    ASTVF_TC_END(AST_EXPR_NULL, v)
1004
1005    static inline bool symbol_var_in_scope(Symbol *tmp_sym UNUSED,
1006        Ast_Node *ast_node UNUSED, Symbol_Table_Node *lookup_node UNUSED)
1007    {
1008        if (!tmp_sym)
1009            return false;
1010        return true;
1011    }
1012
1013    static inline bool type_visitor_record_field_invalid(Ast_Visitor_Type_Check *v,
1014        Symbol_Table_Node *tmp_sym_node)
1015    {
1016        if (v->dot_ref_sym_node)
1017            return false;
1018
1019        if (!v->rec_body_sym_node)
1020            return false;
1021
1022        Symbol_Table_Node_Type tnt = tmp_sym_node->type;
1023        if (v->rec_body_sym_node->scope_id > tmp_sym_node->scope_id &&
1024            (tnt == SYMBOL_TABLE_NODE_REC || tnt == SYMBOL_TABLE_NODE_FUNC))
1025            return true;
1026
1027        return false;
1028    }
1029
1030    static void symbol_property_var_iden_report(Symbol_Table_Node *sym_node,
1031        Symbol *tmp_sym, Ast_Variable_Iden *n, Ast_Visitor_Type_Check *v)
1032    {
1033        Ast_Node *ast_node = AST_NODE_OF(n);
1034
1035        if (symbol_var_in_scope(tmp_sym, ast_node, sym_node)) {
1036            if (type_visitor_record_field_invalid(v, tmp_sym->sym_node)) {
1037
1038                report_error_location(ast_node_get_file_location(ast_node),
1039                    S("record function " QFY("%S")
1040                        " unable to access variable " QFY("%S") "\n"),
1041                    STRING_AFTER_LAST(v->curr_func_iden, '.'), n->iden);
1042
1043                symbol_table_node_insert_unknown_var(v->rec_body_sym_node,
1044                    v->sym_table,
1045                    n->iden,
1046                    ast_node_get_file_location(AST_NODE_OF(n)));
1047                v->prev_sym_type = symbol_type_unknown_alloc(v->sym_table);
1048            } else {

```

```

1049         v->prev_sym_type = tmp_sym->resolved_type;
1050     }
1051
1052     } else {
1053         if (!tmp_sym)
1054             symbol_table_node_insert_unknown_var(sym_node, v->sym_table,
1055             n->iden, ast_node_get_file_location(ast_node));
1056         else
1057             tmp_sym->resolved_type = symbol_type_unknown_alloc(v->sym_table);
1058         report_error_location(ast_node_get_file_location(ast_node),
1059         S("undeclared %svariable " QFY("%S") "\n"),
1060         v->dot_ref_sym_node ? "record " : "",
1061         n->iden);
1062         v->prev_sym_type = symbol_type_unknown_alloc(v->sym_table);
1063     }
1064 }
1065
1066 static bool type_check_rec_construct_valid_access(Ast_Visitor_Type_Check *v,
1067 Symbol *lookup_sym, File_Location *loc)
1068 {
1069     if (!lookup_sym || !v->rec_body_sym_node || !v->in_beginning_of_ctor)
1070         return true;
1071
1072     if (lookup_sym->sym_node->scope_id == v->rec_body_sym_node->scope_id) {
1073         v->prev_sym_type = symbol_type_unknown_alloc(v->sym_table);
1074         report_error_location(loc,
1075         S("record fields inaccessible before invocation of "
1076         "constructors\n"));
1077         return false;
1078     }
1079
1080     return true;
1081 }
1082
1083 static void symbol_property_var_iden(Symbol_Table_Node *sym_node,
1084 Ast_Variable_Iden *n, Ast_Visitor_Type_Check *v,
1085 Symbol_Table_Node *dot_ref_node)
1086 {
1087     Symbol *tmp_sym;
1088     if (!dot_ref_node)
1089         tmp_sym = symbol_table_node_lookup_var(sym_node, n->iden,
1090         ast_node_get_file_location(AST_NODE_OF(n)));
1091     else
1092         tmp_sym = symbol_table_node_get(sym_node, n->iden,
1093         string_hash_code(n->iden), SYMBOL_PROPERTY_VAR);
1094
1095     DEBUGT(def,
1096     if (tmp_sym)
1097         assert(tmp_sym->resolved_type->methods->get_type() !=
1098         SYMBOL_TYPE_CYCLE);
1099     );
1100
1101     if (!type_check_rec_construct_valid_access(v, tmp_sym,
1102     ast_node_get_file_location(AST_NODE_OF(n))))
1103         goto out;
1104
1105     symbol_property_var_iden_report(sym_node, tmp_sym, n, v);
1106 out:;
1107 }
1108
1109 /* Will really only insert an unknown type if the symbol was not found.
1110 * This is because the error has been reported by symbol table. */
1111 static void symbol_property_type_def_iden_report(Symbol *tmp_sym,
1112 Ast_Visitor_Type_Check *v)
1113 {
1114     if (tmp_sym)
1115         v->prev_sym_type = tmp_sym->resolved_type;
1116     else
1117         v->prev_sym_type = symbol_type_unknown_alloc(v->sym_table);
1118 }
1119
1120 static void symbol_property_type_def_iden(Symbol_Table_Node *sym_node,
1121 String iden, Ast_Visitor_Type_Check *v)
1122 {

```

```

1123     Symbol *tmp_sym = symbol_table_node_lookup(sym_node, iden,
1124         SYMBOL_PROPERTY_TYPE_DEF);
1125     symbol_property_type_def_iden_report(tmp_sym, v);
1126 }
1127
1128 static void symbol_property_func_iden_report (
1129     Symbol_Table_Node *sym_node,
1130     Symbol_Func_Map *func_map,
1131     Ast_Variable_Iden *n,
1132     Ast_Visitor_Type_Check *v)
1133 {
1134     Ast_Node *ast_node = AST_NODE_OF(n);
1135
1136     if (func_map) {
1137         if (type_visitor_record_field_invalid(v, func_map->sym_node)) {
1138             report_error_location(ast_node_get_file_location(ast_node),
1139                 S("record function " QFY("%S") " unable to access "
1140                     "function " QFY("%S") "\n"),
1141                 STRING_AFTER_LAST(v->curr_func_iden, '.'), n->iden);
1142
1143             symbol_table_node_insert_unknown_func(v->rec_body_sym_node,
1144                 v->sym_table, n->iden,
1145                 ast_node_get_file_location(AST_NODE_OF(n)));
1146             v->prev_sym_type = symbol_type_unknown_alloc(v->sym_table);
1147             v->prev_func_map = NULL;
1148         } else {
1149             v->prev_sym_type = NULL;
1150             v->prev_func_map = func_map;
1151         }
1152     } else {
1153         symbol_table_node_insert_unknown_func(sym_node, v->sym_table,
1154             n->iden, ast_node_get_file_location(ast_node));
1155         report_error_location(ast_node_get_file_location(ast_node),
1156             S("undefined %sfunction " QFY("%S") "\n"),
1157             v->dot_ref_sym_node ? "record " : "",
1158             STRING_AFTER_DOT(n->iden));
1159         v->prev_sym_type = symbol_type_unknown_alloc(v->sym_table);
1160         v->prev_func_map = NULL;
1161     }
1162 }
1163
1164 static void symbol_property_func_iden(Symbol_Table_Node *sym_node,
1165     Ast_Variable_Iden *n, Ast_Visitor_Type_Check *v,
1166     Symbol_Table_Node *dot_ref_node)
1167 {
1168     Symbol_Func_Map *func_map;
1169     if (!dot_ref_node)
1170         func_map = symbol_table_node_lookup_func_map(sym_node, n->iden);
1171     else
1172         func_map = symbol_table_node_get_func_map(sym_node, n->iden,
1173             string_hash_code(n->iden));
1174
1175     if (func_map) {
1176         File_Location *loc = ast_node_get_file_location(AST_NODE_OF(n));
1177         bool maps_ctor = symbol_func_maps_ctor(func_map);
1178         if (!v->in_beginning_of_ctor && maps_ctor && !v->in_allocate_stmt) {
1179             report_error_location(loc,
1180                 S("call to constructor not allowed here\n"));
1181             v->prev_sym_type = symbol_type_unknown_alloc(v->sym_table);
1182             goto out;
1183         } else if (!maps_ctor) {
1184             Symbol *tmp_sym = vector_get(&func_map->overload_idens, 0);
1185             if (!type_check_rec_construct_valid_access(v, tmp_sym, loc))
1186                 goto out;
1187         }
1188     }
1189
1190     symbol_property_func_iden_report(sym_node, func_map, n, v);
1191
1192 out:;
1193 }
1194
1195 ASTVF_TC_BEGIN(AST_VARIABLE_IDEN, Ast_Visitor_Type_Check, v,
1196     Ast_Variable_Iden, n)

```



```

1197     Ast_Node *ast_node = AST_NODE_OF(n);
1198
1199     Symbol_Table_Node *sym_node = v->dot_ref_sym_node ? v->dot_ref_sym_node :
1200         ast_node_get_symbol_table_node(ast_node);
1201
1202     DEBUGT(def, v->prev_func_map = NULL);
1203
1204     switch (v->next_property) {
1205     case SYMBOL_PROPERTY_VAR:
1206         symbol_property_var_iden(sym_node, n, v, v->dot_ref_sym_node);
1207         v->prev_variable_sym_type = v->prev_sym_type;
1208         break;
1209
1210     case SYMBOL_PROPERTY_TYPE_DEF:
1211         assert(!v->dot_ref_sym_node);
1212         symbol_property_type_def_iden(sym_node, n->iden, v);
1213         break;
1214
1215     case SYMBOL_PROPERTY_FUNC:
1216         if (!v->next_func_iden_is_decl)
1217             symbol_property_func_iden(sym_node, n, v, v->dot_ref_sym_node);
1218         break;
1219     }
1220
1221     Symbol_Table_Node *ref_node;
1222     if (v->next_property == SYMBOL_PROPERTY_FUNC && v->next_func_iden_is_decl)
1223         ref_node = NULL;
1224     else if (v->dot_ref_sym_node)
1225         ref_node = v->dot_ref_sym_node;
1226     else if (v->rec_body_sym_node)
1227         ref_node = v->rec_body_sym_node;
1228     else
1229         ref_node = NULL;
1230
1231     if (ref_node) {
1232         assert(ref_node->node_rec);
1233         Symbol_Type_Rec_Ambiguous_Ref *r =
1234             symbol_type_rec_get_ambiguous_ref(ref_node->node_rec, n->iden);
1235         if (r) {
1236             STRING(locations, "");
1237             File_Location *loc;
1238             VECTOR_FOR_EACH_ENTRY(&r->sym_locations, loc)
1239                 string_append_format(locations, S("\t\tF\n"), loc);
1240
1241             if (ref_node->node_rec->rec_name) {
1242                 report_error_location(ast_node_get_file_location(ast_node),
1243                     S("referencing field " QFY("%2$S")
1244                       " of record " QFY("%1$S")
1245                       " is ambiguous, " QFY("%2$S")
1246                       " is declared here:\n%3$S"),
1247                     STRING_AFTER_DOT(ref_node->node_rec->rec_name),
1248                     n->iden,
1249                     locations);
1250             } else {
1251                 report_error_location(ast_node_get_file_location(ast_node),
1252                     S("referencing field " QFY("%1$S")
1253                       " is ambiguous, " QFY("%1$S")
1254                       " is declared here:\n%2$S"),
1255                     n->iden,
1256                     locations);
1257             }
1258
1259             string_clear(locations);
1260         }
1261     }
1262
1263     v->prev_iden = n->iden;
1264     v->dot_ref_sym_node = NULL;
1265     /* v->lookup_rec = NULL; */
1266     if (v->prev_sym_type) {
1267         assert(!v->prev_func_map);
1268         v->prev_expr_type = symbol_type_to_expr_type(v->prev_sym_type);
1269         ast_variable_iden_set_expr_type(n, v->prev_expr_type);
1270     } else {

```

```

1271     DEBUGT(def,
1272             if (!v->next_func_iden_is_decl)
1273                 assert(v->prev_func_map)
1274         );
1275     }
1276     ASTVF_TC_END(AST_VARIABLE_IDEN, v)
1277
1278     ASTVF_BEGIN(AST_REC_SELF_PTR, Ast_Visitor_Type_Check, v, Ast_Empty, n)
1279     if (!v->rec_body_sym_node) {
1280         report_error_location(ast_node_get_file_location(AST_NODE_OF(n)),
1281                               S("unexpected " QFY("record") " self-reference "
1282                                 "outside of record declaration\n"));
1283         v->prev_sym_type = symbol_type_unknown_alloc(v->sym_table);
1284         v->prev_expr_type = AST_EXPR_TYPE_UNKNOWN;
1285     } else if (v->in_beginning_of_ctor) {
1286         report_error_location(ast_node_get_file_location(AST_NODE_OF(n)),
1287                               S(QFY("record") " self-reference inaccessible "
1288                                 "before invocation of constructors\n"));
1289         v->prev_sym_type = symbol_type_unknown_alloc(v->sym_table);
1290         v->prev_expr_type = AST_EXPR_TYPE_UNKNOWN;
1291     } else {
1292         Symbol *sym = symbol_table_get_from_location(
1293             ast_node_get_symbol_table_node(AST_NODE_OF(n)),
1294             ast_node_get_file_location(AST_NODE_OF(n)));
1295         assert(sym);
1296         v->prev_sym_type = sym->resolved_type;
1297         v->prev_expr_type = symbol_type_to_expr_type(v->prev_sym_type);
1298         if (v->prev_expr_type == AST_EXPR_TYPE_UNKNOWN)
1299             v->prev_variable_sym_type = v->prev_sym_type;
1300         else
1301             v->prev_variable_sym_type = NULL;
1302     }
1303     ASTVF_END
1304
1305     static inline void variable_unexpected_type(Ast_Node *n, Const_String type,
1306         Const_String expected)
1307     {
1308         report_error_location(ast_node_get_file_location(n),
1309                               S("expected " QFY("%S")
1310                                 ", expression evaluates into " QFY("%S") "\n"),
1311                               expected, type);
1312     }
1313
1314     ASTVF_TC_BEGIN(AST_EXPR_ARY_REF, Ast_Visitor_Type_Check, v, Ast_Node_Binary, n)
1315     Symbol_Type_Ary *ary;
1316     Ast_Expr_Type lhs_t;
1317     Symbol_Type_Struct *lhs_s;
1318     n->lhs->accept_visitor(n->lhs, AST_VISITOR_OF(v));
1319     lhs_s = v->prev_variable_sym_type;
1320
1321     lhs_t = v->prev_expr_type;
1322     if (lhs_t == AST_EXPR_TYPE_UNKNOWN)
1323         goto out;
1324
1325     if (lhs_t != AST_EXPR_TYPE_ARY && lhs_t != AST_EXPR_TYPE_STRING) {
1326         report_error_location(ast_node_get_file_location(AST_NODE_OF(n)),
1327                               S("expected " QFY("array") " or " QFY("string")
1328                                 " expression before " QFY("[") " ", "
1329                                 " found " QFY("%S") "\n"),
1330                               ast_expr_type_to_string(v->prev_expr_type));
1331         v->prev_sym_type = symbol_type_unknown_alloc(v->sym_table);
1332         goto out;
1333     }
1334
1335     n->rhs->accept_visitor(n->rhs, AST_VISITOR_OF(v));
1336     switch (v->prev_expr_type) {
1337     case AST_EXPR_TYPE_BOOL:
1338         warn_implicit_cast(ast_node_get_file_location(n->rhs),
1339                             AST_EXPR_TYPE_BOOL, AST_EXPR_TYPE_INT);
1340         /* Fall through. */
1341     case AST_EXPR_TYPE_INT:
1342         /* Fall through. */
1343     case AST_EXPR_TYPE_CHAR:
1344         switch (lhs_s->methods->get_type()) {

```

```

1345     case SYMBOL_TYPE_STRING:
1346         v->prev_sym_type = symbol_type_char_alloc(v->sym_table);
1347         v->lhs_of_assign_is_string_ref = true;
1348         break;
1349     case SYMBOL_TYPE_FUNC:
1350         lhs_s = SYMBOL_TYPE_STRUCT_CONTAINER(lhs_s,
1351             Symbol_Type_Func->return_type;
1352         /* Fall through. */
1353     case SYMBOL_TYPE_ARY:
1354         assert(lhs_s->methods->get_type() == SYMBOL_TYPE_ARY);
1355         ary = SYMBOL_TYPE_STRUCT_CONTAINER(lhs_s, Symbol_Type_Ary);
1356         v->prev_sym_type = ary->ary_type;
1357         break;
1358     default:
1359         assert(false);
1360         break;
1361 }
1362 break;
1363
1364 case AST_EXPR_TYPE_UNKNOWN:
1365     goto out;
1366
1367 default:
1368     report_error_location(ast_node_get_file_location(n->rhs),
1369         S("referencing " QFY("%S") " with " QFY("%S")
1370         " expression, expected integral expression\n"),
1371         ast_expr_type_to_string(lhs_t),
1372         ast_expr_type_to_string(v->prev_expr_type));
1373     v->prev_sym_type = symbol_type_unknown_alloc(v->sym_table);
1374     goto out;
1375 }
1376
1377 out:
1378     v->prev_expr_type = symbol_type_to_expr_type(v->prev_sym_type);
1379     v->prev_variable_sym_type = v->prev_sym_type;
1380     ast_expr_binary_set_expr_type(AST_EXPR_BINARY_OF(n), v->prev_expr_type);
1381 ASTVF_TC_END(AST_EXPR_ARY_REF, v)
1382
1383 static Int func_call_get_compatibility_score(Vector *arg_struct_expr_pairs,
1384     Vector *param_structs)
1385 {
1386     Symbol_Type_Struct *arg, *param;
1387     Uns num_args = vector_size(param_structs);
1388
1389     if (vector_size(arg_struct_expr_pairs) / 2 != num_args)
1390         return 0;
1391
1392     Uns score = 1;
1393
1394     Uns exact = num_args + 1;
1395     Uns rec_cast = num_args;
1396
1397     for (Uns i = 0; i < num_args; i++) {
1398         arg = vector_get(arg_struct_expr_pairs, i * 2);
1399         param = vector_get(param_structs, i);
1400
1401         Ast_Expr_Type arg_t = PTR_TO_INT(
1402             vector_get(arg_struct_expr_pairs, i * 2 + 1));
1403         Ast_Expr_Type par_t = symbol_type_to_expr_type(param);
1404
1405         Assign_Comatibility_Type assign_t =
1406             types_are_assignment_compatible(param, par_t, arg, arg_t);
1407
1408         switch (assign_t) {
1409             case ASSIGN_COMPATIBILITY_TRUE:
1410                 switch (arg_t) {
1411                     case AST_EXPR_TYPE_REC:
1412                         if (par_t == AST_EXPR_TYPE_NULL ||
1413                             param->methods->same_type(param, arg))
1414                             score += exact;
1415                         else
1416                             score += rec_cast;
1417                         break;

```

```

1419         case AST_EXPR_TYPE_STRING:
1420             /* Fall through. */
1421         case AST_EXPR_TYPE_ARY:
1422             if (par_t == AST_EXPR_TYPE_NULL) {
1423                 score += exact;
1424                 break;
1425             }
1426             /* Fall through. */
1427         default:
1428             if (par_t == arg_t)
1429                 score += rec_cast;
1430             break;
1431     }
1432     break;
1433
1434     case ASSIGN_COMPATIBILITY_FALSE:
1435         return 0;
1436
1437     case ASSIGN_COMPATIBILITY_AMBIGUOUS:
1438         if (arg_t == par_t && par_t == AST_EXPR_TYPE_REC)
1439             score += rec_cast;
1440         break;
1441
1442     case ASSIGN_COMPATIBILITY_IMPLICIT:
1443         break;
1444     }
1445 }
1446
1447 return score;
1448 }
1449
1450 // Return score of best match
1451 static Uns func_call_append_best_match(Vector *func_symbols,
1452     Vector *arg_struct_expr_pairs, Vector *result)
1453 {
1454     Uns current_score = 0;
1455
1456     Symbol *fsym;
1457     VECTOR_FOR_EACH_ENTRY(func_symbols, fsym) {
1458         Symbol_Type func_symbol_type =
1459             fsym->resolved_type->methods->get_type();
1460         assert(func_symbol_type == SYMBOL_TYPE_FUNC ||
1461             func_symbol_type == SYMBOL_TYPE_UNKNOWN);
1462
1463         if (func_symbol_type == SYMBOL_TYPE_UNKNOWN) {
1464             vector_clear(result);
1465             current_score = UNSIGNED_MAX;
1466             goto out;
1467         }
1468
1469         Symbol_Type_Func *func = SYMBOL_TYPE_STRUCT_CONTAINER(
1470             fsym->resolved_type, Symbol_Type_Func);
1471         Vector *params = &func->param_types;
1472
1473         Uns score =
1474             func_call_get_compatibility_score(arg_struct_expr_pairs, params);
1475
1476         if (score) {
1477             if (score > current_score) {
1478                 vector_clear(result);
1479                 vector_append(result, fsym);
1480                 current_score = score;
1481             } else if (score == current_score) {
1482                 vector_append(result, fsym);
1483             }
1484         }
1485     }
1486
1487 out:
1488     return current_score;
1489 }
1490
1491 static Vector *func_call_get_best_match(Ast_Visitor_Type_Check *v,
1492     Symbol_Func_Map *curr_map, Vector *arg_struct_expr_pairs)

```

```

1493 {
1494     extern void symbol_func_map_compare(Symbol_Func_Map *func_map,
1495         Symbol_Table *t);
1496
1497     Vector *ret = vector_alloc();
1498     VECTOR(tmp);
1499
1500     VECTOR(func_symbols);
1501
1502     Uns best_score = 0;
1503
1504     do {
1505         symbol_func_map_compare(curr_map, v->sym_table);
1506
1507         Symbol *fsym = NULL;
1508         VECTOR_FOR_EACH_ENTRY(&curr_map->overload_idens, fsym)
1509             vector_append(&func_symbols, fsym);
1510
1511         Uns score = func_call_append_best_match(&func_symbols,
1512             arg_struct_expr_pairs, &tmp);
1513         vector_clear(&func_symbols);
1514
1515         if (score == UNSIGNED_MAX) {
1516             vector_destroy(ret, NULL);
1517             ret = NULL;
1518             break;
1519         }
1520
1521         if (score > best_score) {
1522             best_score = score;
1523             vector_clear(ret);
1524             VECTOR_FOR_EACH_ENTRY(&tmp, fsym)
1525                 vector_append(ret, fsym);
1526         }
1527
1528         Symbol *sym = vector_get(&curr_map->overload_idens, 0);
1529         assert(sym->resolved_type->methods->get_type() == SYMBOL_TYPE_FUNC);
1530
1531         Symbol_Type_Func *func = SYMBOL_TYPE_STRUCT_CONTAINER(
1532             sym->resolved_type, Symbol_Type_Func);
1533         Symbol_Table_Node *real_parent = func->body_sym_node->parent;
1534         while (real_parent && real_parent->type == SYMBOL_TABLE_NODE_IMPORT)
1535             real_parent = real_parent->parent;
1536         Symbol_Table_Node *next_node;
1537         if (real_parent)
1538             next_node = real_parent->parent;
1539         else
1540             next_node = func->body_sym_node->parent->parent;
1541
1542         if (next_node)
1543             curr_map = symbol_table_node_lookup_func_map(next_node,
1544                 curr_map->func_iden);
1545         else
1546             curr_map = NULL;
1547
1548         vector_clear(&tmp);
1549     } while (curr_map);
1550
1551     vector_clear(&tmp);
1552
1553     return ret;
1554 }
1555
1556 ASTVF_TC_BEGIN(AST_EXPR_FUNC_CALL, Ast_Visitor_Type_Check, v,
1557     Ast_Expr_Func_Call, n)
1558     Uns num_args;
1559     Symbol_Type_Struct *tmp_type;
1560     Ast_Node *arg;
1561     String func_iden;
1562     Symbol_Type_Func *func = NULL;
1563
1564     Symbol_Property saved_property = v->next_property;
1565     v->next_property = SYMBOL_PROPERTY_FUNC;
1566     n->identifier->accept_visitor(n->identifier, AST_VISITOR_OF(v));

```

```

1567     v->next_property = saved_property;
1568     func_iden = v->prev_iden;
1569
1570     Vector *sym_results = NULL;
1571     VECTOR(arg_struct_expr_pairs);
1572     VECTOR(arg_locations);
1573
1574     if (v->prev_sym_type) {
1575         assert(v->prev_expr_type == AST_EXPR_TYPE_UNKNOWN);
1576         goto out;
1577     }
1578     assert(v->prev_func_map);
1579     Symbol_Func_Map *func_map = v->prev_func_map;
1580
1581     bool args_are_valid = true;
1582     VECTOR_FOR_EACH_ENTRY(n->arguments, arg) {
1583         arg->accept_visitor(arg, AST_VISITOR_OF(v));
1584         if (v->prev_expr_type == AST_EXPR_TYPE_VOID) {
1585             args_are_valid = false;
1586             report_error_location(ast_node_get_file_location(arg),
1587                 S("invalid " QFY("void") " function argument\n"));
1588         }
1589         if (v->prev_expr_type == AST_EXPR_TYPE_UNKNOWN) {
1590             v->prev_sym_type = symbol_type_unknown_alloc(v->sym_table);
1591             goto out;
1592         }
1593         vector_append(&arg_struct_expr_pairs, v->prev_sym_type);
1594         vector_append(&arg_struct_expr_pairs, INT_TO_PTR(v->prev_expr_type));
1595         vector_append(&arg_locations, ast_node_get_file_location(arg));
1596     }
1597
1598     if (args_are_valid)
1599         sym_results =
1600             func_call_get_best_match(v, func_map, &arg_struct_expr_pairs);
1601
1602     if (!sym_results) {
1603         v->prev_sym_type = symbol_type_unknown_alloc(v->sym_table);
1604         goto out;
1605     }
1606
1607     Symbol *fsym = vector_is_empty(sym_results) ?
1608         NULL : vector_get(sym_results, 0);
1609
1610     switch (vector_size(sym_results)) {
1611     case 0:
1612         report_error_location(ast_node_get_file_location(AST_NODE_OF(n)),
1613             S("unable to locate function " QFY("%S")
1614                 " with parameter list that matches arguments\n"),
1615             func_iden);
1616         v->prev_sym_type = symbol_type_unknown_alloc(v->sym_table);
1617         goto out;
1618     case 1:
1619         break;
1620     default:;
1621
1622     bool all_extern_c = true;
1623     Symbol *tmp_sym;
1624     VECTOR_FOR_EACH_ENTRY(sym_results, tmp_sym) {
1625         Symbol_Type_Func *tmp_func = SYMBOL_TYPE_STRUCT_CONTAINER(
1626             tmp_sym->resolved_type, Symbol_Type_Func);
1627         if (!tmp_func->is_extern_c) {
1628             all_extern_c = false;
1629             break;
1630         }
1631     }
1632     if (all_extern_c)
1633         break;
1634
1635     STRING(loc, "");
1636     if (symbol_get_symbol_table_node_type(fsym) ==
1637         SYMBOL_TABLE_NODE_IMPORT) {
1638
1639         VECTOR(imp_modules);
1640         VECTOR_FOR_EACH_ENTRY(sym_results, fsym) {

```

```

1641     String tmp = string_between_alloc(fsym->unique_name, '.');
1642     if (vector_contains(&imp_modules,
1643         (Vector_Comparator)string_compare, tmp)) {
1644         string_destroy(tmp);
1645     } else {
1646         string_append_format(loc, S("\t%S\n"), tmp);
1647         vector_append(&imp_modules, tmp);
1648     }
1649 }
1650 vector_for_each_destroy(&imp_modules,
1651     (Vector_Destructor)string_destroy);
1652
1653 report_error_location(ast_node_get_file_location(AST_NODE_OF(n)),
1654     S("call of function " QFY("%1$S") " is ambiguous"
1655     ", equally matching declarations imported from:\n%2$S"),
1656     func_iden, loc);
1657 } else {
1658
1659     int num_unique = 0;
1660     VECTOR(func_locations);
1661     VECTOR_FOR_EACH_ENTRY(sym_results, fsym) {
1662         String tmp = fsym->unique_name;
1663         if (vector_contains(&func_locations,
1664             (Vector_Comparator)string_compare, tmp)) {
1665         } else {
1666             ++num_unique;
1667             string_append_format(loc, S("\t%F\n"), fsym->location);
1668             vector_append(&func_locations, tmp);
1669         }
1670     }
1671     vector_clear(&func_locations);
1672
1673 #if 0
1674     VECTOR_FOR_EACH_ENTRY(sym_results, fsym)
1675         string_append_format(loc, S("\t%F\n"), fsym->location);
1676 #endif
1677
1678     report_error_location(ast_node_get_file_location(AST_NODE_OF(n)),
1679         S("call of function " QFY("%S") " is ambiguous"
1680         ", equally matching declarations located here:\n%S"),
1681         func_iden, loc);
1682 }
1683 string_clear(loc);
1684 v->prev_sym_type = symbol_type_unknown_alloc(v->sym_table);
1685 goto out;
1686 }
1687
1688 func = SYMBOL_TYPE_STRUCT_CONTAINER(fsym->resolved_type, Symbol_Type_Func);
1689
1690 num_args = vector_size(&func->param_types);
1691 assert(vector_size(&arg_struct_expr_pairs) / 2 == num_args);
1692
1693 for (Uns i = 0; i < num_args; i++) {
1694     Symbol_Type_Struct *arg_struct = vector_get(
1695         &arg_struct_expr_pairs, i * 2);
1696     tmp_type = vector_get(&func->param_types, i);
1697
1698     Ast_Expr_Type par_t = symbol_type_to_expr_type(tmp_type);
1699     Ast_Expr_Type arg_t = PTR_TO_INT(
1700         vector_get(&arg_struct_expr_pairs, i * 2 + 1));
1701     Assign_Compatibility_Type assign_t =
1702         types_are_assignment_compatible(tmp_type, par_t,
1703         arg_struct, arg_t);
1704
1705     switch (assign_t) {
1706     case ASSIGN_COMPATIBILITY_TRUE:
1707         break;
1708
1709     case ASSIGN_COMPATIBILITY_FALSE:
1710         assert(false);
1711         break;
1712
1713     case ASSIGN_COMPATIBILITY_AMBIGUOUS:
1714         report_error_location(

```

```

1715         vector_get(&arg_locations, i),
1716         S("passing argument %U to function " QFY("%S")
1717           " is ambiguous, record is extending multiple "
1718           "records with the required structure\n"),
1719         i + 1, func_iden);
1720     v->prev_sym_type = symbol_type_unknown_alloc(v->sym_table);
1721     goto out;
1722
1723     case ASSIGN_COMPATIBILITY_IMPLICIT:
1724         /* Implicit cast from par_t to arg_t. */
1725         warn_implicit_cast(vector_get(&arg_locations, i), arg_t, par_t);
1726         break;
1727     }
1728 }
1729 Ast_Variable_Iden *iden_node =
1730     AST_CONTAINER_OF(n->identifier, Ast_Variable_Iden);
1731 string_destroy(iden_node->iden);
1732 iden_node->iden = string_duplicate(fsym->identifier);
1733 func_iden = fsym->identifier;
1734
1735 n->func = func;
1736 v->prev_sym_type = func->return_type;
1737 out:
1738 v->prev_expr_type = symbol_type_to_expr_type(v->prev_sym_type);
1739 v->prev_func_call_iden = func_iden;
1740
1741 if (v->prev_expr_type != AST_EXPR_TYPE_UNKNOWN)
1742     v->prev_variable_sym_type = symbol_table_node_lookup(
1743         func->body_sym_node->parent,
1744         func_iden,
1745         SYMBOL_PROPERTY_FUNC)->resolved_type;
1746
1747 vector_clear(&arg_locations);
1748 vector_clear(&arg_struct_expr_pairs);
1749 if (sym_results)
1750     vector_destroy(sym_results, NULL);
1751
1752 ast_expr_func_call_set_expr_type(n, v->prev_expr_type);
1753 ASTVF_TC_END(AST_EXPR_FUNC_CALL, v)
1754
1755
1756 ASTVF_TC_BEGIN(AST_EXPR_DOT_REF, Ast_Visitor_Type_Check, v,
1757     Ast_Node_Binary, n)
1758     Symbol_Type_Rec *rec;
1759     n->lhs->accept_visitor(n->lhs, AST_VISITOR_OF(v));
1760     if (v->prev_expr_type == AST_EXPR_TYPE_UNKNOWN)
1761         goto out;
1762
1763     if (v->prev_expr_type != AST_EXPR_TYPE_REC) {
1764         variable_unexpected_type(AST_NODE_OF(n),
1765             ast_expr_type_to_string(v->prev_expr_type), S("record"));
1766         v->prev_sym_type = symbol_type_unknown_alloc(v->sym_table);
1767         goto out;
1768     }
1769
1770     rec = SYMBOL_TYPE_STRUCT_CONTAINER(v->prev_sym_type, Symbol_Type_Rec);
1771
1772     /* v->lookup_rec = rec; */
1773     v->dot_ref_sym_node = rec->rec_sym_node;
1774     n->rhs->accept_visitor(n->rhs, AST_VISITOR_OF(v));
1775
1776 out:
1777     v->prev_expr_type = symbol_type_to_expr_type(v->prev_sym_type);
1778     ast_expr_binary_set_expr_type(AST_EXPR_BINARY_OF(n), v->prev_expr_type);
1779 ASTVF_TC_END(AST_EXPR_DOT_REF, v)
1780
1781 /* Defined in symbol table. */
1782 bool __type_def_symbol_types_equal(Symbol_Type_Struct *lhs,
1783     Symbol_Type_Struct *rhs);
1784
1785 static bool type_check_rec_extends(Symbol_Type_Rec *rec,
1786     Symbol_Type_Struct *base)
1787 {
1788     Symbol_Type_Struct *ext;

```



```

1789     VECTOR_FOR_EACH_ENTRY(&rec->extended_types, ext) {
1790         if (___type_def_symbol_types_equal(ext, base))
1791             return true;
1792     }
1793     return false;
1794 }
1795
1796 ASTVF_TC_BEGIN(AST_EXPR_DIRECT_REF, Ast_Visitor_Type_Check, v,
1797     Ast_Node_Binary, n)
1798     Symbol_Type_Struct *ref_rec_result =
1799         symbol_type_unknown_alloc(v->sym_table);
1800     Symbol_Type_Struct *ref_func_result =
1801         symbol_type_unknown_alloc(v->sym_table);
1802
1803     Symbol_Type_Rec *ref_rec;
1804     Symbol_Type_Rec *curr_rec;
1805
1806     if (!v->rec_body_sym_node) {
1807         report_error_location(ast_node_get_file_location(AST_NODE_OF(n)),
1808             S("direct record reference outside record scope\n"));
1809         v->prev_sym_type = symbol_type_unknown_alloc(v->sym_table);
1810         goto out;
1811     }
1812     curr_rec = v->rec_body_sym_node->node_rec;
1813     assert(curr_rec);
1814
1815     if (n->lhs) {
1816         n->lhs->accept_visitor(n->lhs, AST_VISITOR_OF(v));
1817         if (v->prev_expr_type == AST_EXPR_TYPE_UNKNOWN)
1818             goto out;
1819
1820         if (v->prev_expr_type != AST_EXPR_TYPE_REC) {
1821             report_error_location(ast_node_get_file_location(AST_NODE_OF(n)),
1822                 S("expected type " QFY("record")
1823                     ", found type " QFY("%S") "\n"),
1824                 ast_expr_type_to_string(v->prev_expr_type));
1825             v->prev_sym_type = symbol_type_unknown_alloc(v->sym_table);
1826             goto out;
1827         }
1828
1829         ref_rec = SYMBOL_TYPE_STRUCT_CONTAINER(v->prev_sym_type, Symbol_Type_Rec);
1830     } else {
1831         ref_rec = v->rec_body_sym_node->node_rec;
1832     }
1833
1834     Symbol_Type_Struct *ref_struct =
1835         SYMBOL_TYPE_STRUCT_OF_CONTAINER(ref_rec);
1836     Symbol_Type_Struct *curr_struct =
1837         SYMBOL_TYPE_STRUCT_OF_CONTAINER(curr_rec);
1838
1839     bool compatible;
1840     if (ref_struct->methods->same_type(ref_struct, curr_struct))
1841         compatible = ___type_def_symbol_types_equal(ref_struct,
1842             curr_struct);
1843     else
1844         compatible = type_check_rec_extends(
1845             curr_rec, ref_struct);
1846
1847     if (!compatible) {
1848         String ref_str;
1849         if (ref_rec->rec_name)
1850             ref_str = string_from_format(S("record " QFY("%S")),
1851                 STRING_AFTER_DOT(ref_rec->rec_name));
1852         else
1853             ref_str = string_alloc(S("the given record type"));
1854
1855         String curr_str;
1856         if (curr_rec->rec_name)
1857             curr_str = string_from_format(S("record " QFY("%S")),
1858                 STRING_AFTER_DOT(curr_rec->rec_name));
1859         else
1860             curr_str = string_alloc(S("current record"));
1861
1862         report_error_location(ast_node_get_file_location(AST_NODE_OF(n)),

```

```

1863         S("%S cannot directly reference %S\n"),
1864         curr_str, ref_str);
1865     string_destroy(curr_str);
1866     string_destroy(ref_str);
1867     v->prev_sym_type = symbol_type_unknown_alloc(v->sym_table);
1868     goto out;
1869 }
1870
1871 bool in_ctor_list = v->in_beginning_of_ctor;
1872 if (n->rhs->accept_visitor ==
1873     AST_NODE_ACCEPT_VISITOR_FUNC(AST_EXPR_FUNC_CALL)) {
1874     Ast_Expr_Func_Call *call =
1875         AST_CONTAINER_OF(n->rhs, Ast_Expr_Func_Call);
1876     Ast_Variable_Iden *iden = AST_CONTAINER_OF(call->identifier,
1877         Ast_Variable_Iden);
1878     if (!string_is_ctor(iden->iden))
1879         v->in_beginning_of_ctor = false;
1880 } else {
1881     v->in_beginning_of_ctor = false;
1882 }
1883
1884 /* v->lookup_rec = ref_rec; */
1885 v->dot_ref_sym_node = ref_rec->rec_sym_node;
1886 n->rhs->accept_visitor(n->rhs, AST_VISITOR_OF(v));
1887
1888 v->in_beginning_of_ctor = in_ctor_list;
1889
1890 if (n->rhs->accept_visitor ==
1891     AST_NODE_ACCEPT_VISITOR_FUNC(AST_EXPR_FUNC_CALL)) {
1892     Ast_Expr_Func_Call *fnode =
1893         AST_CONTAINER_OF(n->rhs, Ast_Expr_Func_Call);
1894     Symbol_Type_Func *func = fnode->func;
1895     if (func)
1896         ref_func_result = SYMBOL_TYPE_STRUCT_OF_CONTAINER(func);
1897 }
1898 ref_rec_result = SYMBOL_TYPE_STRUCT_OF_CONTAINER(ref_rec);
1899
1900 out:
1901 v->prev_direct_ref_rec = ref_rec_result;
1902 v->prev_direct_ref_func = ref_func_result;
1903 v->prev_expr_type = symbol_type_to_expr_type(v->prev_sym_type);
1904 ast_expr_binary_set_expr_type(AST_EXPR_BINARY_OF(n), v->prev_expr_type);
1905 ASTVF_TC_END(AST_EXPR_DIRECT_REF, v)
1906
1907 ASTVF_TC_BEGIN(AST_SIMPLE_TYPE_INT, Ast_Visitor_Type_Check, v, Ast_Empty, n)
1908     (void)n;
1909     v->prev_expr_type = AST_EXPR_TYPE_INT;
1910     ASTVF_TC_END(AST_SIMPLE_TYPE_INT, v)
1911
1912 ASTVF_TC_BEGIN(AST_SIMPLE_TYPE_BOOL, Ast_Visitor_Type_Check, v, Ast_Empty, n)
1913     (void)n;
1914     v->prev_expr_type = AST_EXPR_TYPE_BOOL;
1915     ASTVF_TC_END(AST_SIMPLE_TYPE_BOOL, v)
1916
1917 ASTVF_TC_BEGIN(AST_TYPE_IDEN, Ast_Visitor_Type_Check, v, Ast_Type_Iden, n)
1918     Ast_Node *ast_node = AST_NODE_OF(n);
1919
1920     Symbol_Table_Node *sym_node = ast_node_get_symbol_table_node(ast_node);
1921
1922     symbol_property_type_def_iden(sym_node, n->iden, v);
1923
1924     v->prev_iden = n->iden;
1925     v->prev_expr_type = symbol_type_to_expr_type(v->prev_sym_type);
1926     ASTVF_TC_END(AST_TYPE_IDEN, v)
1927
1928 ASTVF_TC_BEGIN(AST_VAR_DECL, Ast_Visitor_Type_Check, v, Ast_Node_Binary, n)
1929     Symbol_Property saved_property = v->next_property;
1930     v->next_property = SYMBOL_PROPERTY_VAR;
1931     DEBUGT(def, v->prev_iden = NULL);
1932     n->lhs->accept_visitor(n->lhs, AST_VISITOR_OF(v));
1933     assert(v->prev_iden);
1934     String iden = v->prev_iden;
1935     v->next_property = saved_property;
1936     /* Identifiers should be resolved.

```

```

1937     * So no need to modify v->prev_property. */
1938     n->rhs->accept_visitor(n->rhs, AST_VISITOR_OF(v));
1939     v->prev_var_decl_iden = iden;
1940     ASTVF_TC_END(AST_VAR_DECL, v)
1941
1942     ASTVF_TC_BEGIN(AST_TYPE_DEF, Ast_Visitor_Type_Check, v, Ast_Node_Binary, n)
1943     Symbol_Property saved_property = v->next_property;
1944     v->next_property = SYMBOL_PROPERTY_TYPE_DEF;
1945     n->lhs->accept_visitor(n->lhs, AST_VISITOR_OF(v));
1946     v->next_property = saved_property;
1947     /* Identifiers should be resolved.
1948     * So no need to modify v->prev_property. */
1949     n->rhs->accept_visitor(n->rhs, AST_VISITOR_OF(v));
1950     ASTVF_TC_END(AST_TYPE_DEF, v)
1951
1952     ASTVF_TC_BEGIN(AST_TYPE_ARY, Ast_Visitor_Type_Check, v, Ast_Type, n)
1953     n->type->accept_visitor(n->type, AST_VISITOR_OF(v));
1954     v->prev_expr_type = AST_EXPR_TYPE_ARY;
1955     ASTVF_TC_END(AST_TYPE_ARY, v)
1956
1957     static void type_check_rec_default_constructor(Symbol_Type_Rec *rec,
1958           File_Location *loc);
1959
1960     ASTVF_TC_BEGIN(AST_TYPE_REC, Ast_Visitor_Type_Check, v, Ast_Type_Rec, n)
1961     Symbol_Table_Node *prev_body_node;
1962     Ast_Node *arg;
1963     Vector *vargs = n->extend_list;
1964     VECTOR_FOR_EACH_ENTRY(vargs, arg)
1965     arg->accept_visitor(arg, AST_VISITOR_OF(v));
1966
1967     type_check_rec_default_constructor(n->body_node->node_rec,
1968           ast_node_get_file_location(AST_NODE_OF(n)));
1969
1970     vargs = n->body;
1971     if (!vector_is_empty(vargs)) {
1972         prev_body_node = v->rec_body_sym_node;
1973
1974         arg = vector_get(vargs, 0);
1975         v->rec_body_sym_node = ast_node_get_symbol_table_node(arg);
1976         type_check_rec_default_constructor(v->rec_body_sym_node->node_rec,
1977               ast_node_get_file_location(AST_NODE_OF(n)));
1978
1979         arg->accept_visitor(arg, AST_VISITOR_OF(v));
1980
1981         for (Uns i = 1; i < vector_size(vargs); i++) {
1982             arg = vector_get(vargs, i);
1983             arg->accept_visitor(arg, AST_VISITOR_OF(v));
1984         }
1985
1986         v->rec_body_sym_node = prev_body_node;
1987     }
1988
1989     v->prev_expr_type = AST_EXPR_TYPE_REC;
1990     ASTVF_TC_END(AST_TYPE_REC, v)
1991
1992     static void stmt_list_action(Ast_Visitor_Type_Check *v, Ast_Stmt_List *n)
1993     {
1994         ++v->stmt_list_nest;
1995         Ast_Node *stmt;
1996         Vector *statements = n->statements;
1997         VECTOR_FOR_EACH_ENTRY(statements, stmt)
1998             stmt->accept_visitor(stmt, AST_VISITOR_OF(v));
1999         --v->stmt_list_nest;
2000     }
2001
2002     ASTVF_TC_BEGIN(AST_STMT_LIST, Ast_Visitor_Type_Check, v, Ast_Stmt_List, n)
2003     stmt_list_action(v, n);
2004     ASTVF_TC_END(AST_STMT_LIST, v)
2005
2006     ASTVF_TC_BEGIN(AST_FIN_STMT_LIST, Ast_Visitor_Type_Check, v, Ast_Stmt_List, n)
2007     if (v->stmt_list_nest > 1) {
2008         report_error_location(ast_node_get_file_location(AST_NODE_OF(n)),
2009               S(QFY("finalize") " statement only valid in "
2010                     "outermost scope\n"));

```

```

2011     }
2012     stmt_list_action(v, n);
2013     ASTVF_TC_END(AST_STMT_LIST, v)
2014
2015     static void func_type_check(Ast_Visitor_Type_Check *v, Ast_Func_Def *n,
2016                               Symbol_Table_Node *rec_body_sym_node, Type_Check_Func_Type tt_ftype)
2017     {
2018         Ast_Node *p;
2019         Symbol_Type_Struct *saved_return_sym;
2020         Ast_Expr_Type saved_return_type;
2021         String func_iden;
2022         String prev_func_iden;
2023
2024         Type_Check_Func_Type saved_func_type = v->current_func_type;
2025         v->current_func_type = tt_ftype;
2026
2027         Vector *vec = n->parameters;
2028
2029         Symbol_Property saved_property = v->next_property;
2030         v->next_property = SYMBOL_PROPERTY_FUNC;
2031         v->next_func_iden_is_decl = true;
2032         n->iden->accept_visitor(n->iden, AST_VISITOR_OF(v));
2033         v->next_func_iden_is_decl = false;
2034         func_iden = v->prev_iden;
2035         v->next_property = saved_property;
2036
2037         n->return_type->accept_visitor(n->return_type, AST_VISITOR_OF(v));
2038
2039         Symbol *sym = symbol_table_node_get(
2040             ast_node_get_symbol_table_node(AST_NODE_OF(n)),
2041             func_iden,
2042             string_hash_code(func_iden),
2043             SYMBOL_PROPERTY_FUNC);
2044         assert(sym);
2045         Symbol_Type_Func *func = SYMBOL_TYPE_STRUCT_CONTAINER(
2046             sym->resolved_type, Symbol_Type_Func);
2047
2048         if (tt_ftype == TT_FUNC_TYPE_FINALIZE) {
2049             if (v->prev_expr_type != AST_EXPR_TYPE_VOID &&
2050                 v->prev_expr_type != AST_EXPR_TYPE_UNKNOWN) {
2051                 report_error_location(ast_node_get_file_location(n->return_type),
2052                     S(QFY("finalize") " function expected to have " QFY("void")
2053                         " return type, found " QFY("%S") "\n"),
2054                     ast_expr_type_to_string(v->prev_expr_type));
2055             }
2056             if (rec_body_sym_node->finalize_func_count == 1) {
2057                 report_error_location(ast_node_get_file_location(AST_NODE_OF(n)),
2058                     S("multiple definitions of " QFY("finalize")
2059                         " function inside record\n"));
2060             }
2061             ++rec_body_sym_node->finalize_func_count;
2062         } else if (tt_ftype == TT_FUNC_TYPE_RECORD) {
2063             if (v->prev_expr_type != AST_EXPR_TYPE_VOID &&
2064                 v->prev_expr_type != AST_EXPR_TYPE_UNKNOWN) {
2065                 report_error_location(ast_node_get_file_location(n->return_type),
2066                     S(QFY("record") " function expected to have " QFY("void")
2067                         " return type, found " QFY("%S") "\n"),
2068                     ast_expr_type_to_string(v->prev_expr_type));
2069             }
2070         }
2071
2072         bool func_is_main = false;
2073         if (!rec_body_sym_node && !string_compare(func_iden, MAIN_FUNC_STR)) {
2074             if (vector_size(vec) > 1 && !func->main_param_err_reported) {
2075                 report_error_location(ast_node_get_file_location(AST_NODE_OF(n)),
2076                     S(QFY(MAIN_FUNC_CSTR) " function expected to have at "
2077                         "most one parameter, " QFY("array of string") "\n"));
2078                 func->main_param_err_reported = true;
2079             }
2080             func_is_main = true;
2081         }
2082
2083         saved_return_sym = v->func_return_sym;
2084         saved_return_type = v->func_return_type;

```

```

2085
2086     if (sym->resolved_type->methods->get_type() == SYMBOL_TYPE_UNKNOWN) {
2087         v->func_return_sym = symbol_type_unknown_alloc(v->sym_table);
2088         v->func_return_type = AST_EXPR_TYPE_UNKNOWN;
2089     } else {
2090         if (func_is_main &&
2091             func->return_type->methods->get_type() != SYMBOL_TYPE_INT) {
2092             if (!func->main_err_int_reported) {
2093                 report_error_location(
2094                     ast_node_get_file_location(AST_NODE_OF(n)),
2095                     S(QFY(MAIN_FUNC_CSTR) " function must have "
2096                       QFY("int") " return type\n"));
2097                 func->main_err_int_reported = true;
2098             }
2099         }
2100     }
2101     v->func_return_sym = SYMBOL_TYPE_STRUCT_CONTAINER(sym->resolved_type,
2102     Symbol_Type_Func->return_type;
2103     v->func_return_type = symbol_type_to_expr_type(v->func_return_sym);
2104
2105     if (func_is_main && vector_size(vec)) {
2106         p = vector_get(vec, 0);
2107         p->accept_visitor(p, AST_VISITOR_OF(v));
2108
2109         Symbol_Type_Ary *ary;
2110         Symbol *par_sym = symbol_table_node_get(
2111             func->body_sym_node,
2112             v->prev_var_decl_iden,
2113             string_hash_code(v->prev_var_decl_iden),
2114             SYMBOL_PROPERTY_VAR);
2115         /* This might be so if there are multiple
2116          * definitions of main in the same scope. */
2117         if (par_sym) {
2118             ary = SYMBOL_TYPE_STRUCT_CONTAINER(par_sym->resolved_type,
2119             Symbol_Type_Ary);
2120             if ((v->prev_expr_type != AST_EXPR_TYPE_ARRAY ||
2121                 ary->ary_type->methods->get_type() !=
2122                 SYMBOL_TYPE_STRING) &&
2123                 !func->main_param_err_reported) {
2124                 report_error_location(ast_node_get_file_location(p),
2125                     S(QFY(MAIN_FUNC_CSTR) " function expected to have "
2126                       QFY("array of string") " parameter\n"));
2127                 func->main_param_err_reported = true;
2128             }
2129         }
2130     } else {
2131         VECTOR_FOR_EACH_ENTRY(vec, p)
2132             p->accept_visitor(p, AST_VISITOR_OF(v));
2133     }
2134
2135     if (n->statements) {
2136         prev_func_iden = v->curr_func_iden;
2137         v->curr_func_iden = func_iden;
2138         n->statements->accept_visitor(n->statements, AST_VISITOR_OF(v));
2139         v->curr_func_iden = prev_func_iden;
2140     }
2141
2142     v->func_return_sym = saved_return_sym;
2143     v->func_return_type = saved_return_type;
2144 }
2145 v->current_func_type = saved_func_type;
2146 }
2147
2148 ASTVF_BEGIN(AST_EXT_FUNC_DECL, Ast_Visitor_Type_Check, v, Ast_Func_Def, n)
2149     func_type_check(v, n, NULL, TT_FUNC_TYPE_NORMAL);
2150 ASTVF_END
2151
2152 ASTVF_TC_BEGIN(AST_FIN_FUNC_DEF, Ast_Visitor_Type_Check, v, Ast_Func_Def, n)
2153     Symbol_Table_Node *rec_sym_node = v->rec_body_sym_node;
2154     Type_Check_Func_Type tt;
2155     if (!rec_sym_node) {
2156         tt = TT_FUNC_TYPE_NORMAL;
2157         report_error_location(ast_node_get_file_location(AST_NODE_OF(n)),
2158             S("definition of " QFY("finalize")

```

```

2159         " function outside record scope\n"));
2160     } else if (ast_node_get_symbol_table_node(AST_NODE_OF(n))->scope_id !=
2161         rec_sym_node->scope_id) {
2162         tt = TT_FUNC_TYPE_NORMAL;
2163         report_error_location(ast_node_get_file_location(AST_NODE_OF(n)),
2164             S("nested " QFY("finalize") " function definition\n"));
2165     } else {
2166         tt = TT_FUNC_TYPE_FINALIZE;
2167         if (!vector_is_empty(n->parameters)) {
2168             report_error_location(ast_node_get_file_location(AST_NODE_OF(n)),
2169                 S(QFY("finalize")
2170                     " function expected to have empty parameter list\n"));
2171         }
2172     }
2173     func_type_check(v, n, rec_sym_node, tt);
2174     ASTVF_TC_END(AST_FIN_FUNC_DEF, v)
2175
2176     ASTVF_TC_BEGIN(AST_REC_FUNC_DEF, Ast_Visitor_Type_Check, v, Ast_Func_Def, n)
2177     Symbol_Table_Node *rec_sym_node = v->rec_body_sym_node;
2178     Type_Check_Func_Type tt;
2179     if (!rec_sym_node) {
2180         tt = TT_FUNC_TYPE_NORMAL;
2181         report_error_location(ast_node_get_file_location(AST_NODE_OF(n)),
2182             S("definition of " QFY("record")
2183                 " function outside record scope\n"));
2184     } else if (ast_node_get_symbol_table_node(AST_NODE_OF(n))->scope_id !=
2185         rec_sym_node->scope_id) {
2186         tt = TT_FUNC_TYPE_NORMAL;
2187         report_error_location(ast_node_get_file_location(AST_NODE_OF(n)),
2188             S("nested definition of function " QFY("record") "\n"));
2189     } else {
2190         tt = TT_FUNC_TYPE_RECORD;
2191     }
2192     func_type_check(v, n, rec_sym_node, tt);
2193     ASTVF_TC_END(AST_REC_FUNC_DEF, v)
2194
2195     ASTVF_TC_BEGIN(AST_FUNC_DEF, Ast_Visitor_Type_Check, v, Ast_Func_Def, n)
2196     func_type_check(v, n, NULL, TT_FUNC_TYPE_NORMAL);
2197     ASTVF_TC_END(AST_FUNC_DEF, v)
2198
2199     static void stmt_expr_error_report(Ast_Expr_Type expr_type,
2200         const char *stmt_name, Ast_Node *stmt_node)
2201     {
2202         if (expr_type == AST_EXPR_TYPE_VOID) {
2203             report_error_location(ast_node_get_file_location(stmt_node),
2204                 S("invalid " QFY("void") " type in "
2205                     QFY("%s") "-expression\n"),
2206                 stmt_name);
2207         } else if (expr_type != AST_EXPR_TYPE_BOOL &&
2208             expr_type != AST_EXPR_TYPE_UNKNOWN) {
2209             warn_implicit_expr_cast(stmt_node, expr_type,
2210                 AST_EXPR_TYPE_BOOL, "if");
2211         }
2212     }
2213
2214     ASTVF_TC_BEGIN(AST_IF_STMT, Ast_Visitor_Type_Check, v, Ast_Node_Binary, n)
2215     n->lhs->accept_visitor(n->lhs, AST_VISITOR_OF(v));
2216     stmt_expr_error_report(v->prev_expr_type, "if", AST_NODE_OF(n));
2217     n->rhs->accept_visitor(n->rhs, AST_VISITOR_OF(v));
2218     ASTVF_TC_END(AST_IF_STMT, v)
2219
2220     ASTVF_TC_BEGIN(AST_IF_ELSE_STMT, Ast_Visitor_Type_Check, v,
2221         Ast_Node_Ternary, n)
2222     n->lhs->accept_visitor(n->lhs, AST_VISITOR_OF(v));
2223     stmt_expr_error_report(v->prev_expr_type, "if", AST_NODE_OF(n));
2224     n->mid->accept_visitor(n->mid, AST_VISITOR_OF(v));
2225     n->rhs->accept_visitor(n->rhs, AST_VISITOR_OF(v));
2226     ASTVF_TC_END(AST_IF_ELSE_STMT, v)
2227
2228     ASTVF_TC_BEGIN(AST_ALLOC_ARY, Ast_Visitor_Type_Check, v, Ast_Node_Binary, n)
2229     n->lhs->accept_visitor(n->lhs, AST_VISITOR_OF(v));
2230     switch (v->prev_expr_type) {
2231     case AST_EXPR_TYPE_UNKNOWN:
2232         /* Fall through. */

```

```

2233     case AST_EXPR_TYPE_ARY:
2234         if (v->prev_variable_sym_type->methods->get_type() ==
2235             SYMBOL_TYPE_FUNC) {
2236             report_error_location(ast_node_get_file_location(n->lhs),
2237                 S("invalid " QFY("allocate") " of function result,"
2238                     " expected variable of type " QFY("array") "\n"));
2239         }
2240         break;
2241
2242     default:
2243         report_error_location(ast_node_get_file_location(n->lhs),
2244             S("expected variable of type " QFY("array")
2245                 ", found " QFY("%S") "\n"),
2246             ast_expr_type_to_string(v->prev_expr_type));
2247         break;
2248     }
2249
2250     n->rhs->accept_visitor(n->rhs, AST_VISITOR_OF(v));
2251     switch (v->prev_expr_type) {
2252     case AST_EXPR_TYPE_UNKNOWN:
2253         /* Fall through. */
2254     case AST_EXPR_TYPE_INT:
2255         /* Fall through. */
2256     case AST_EXPR_TYPE_CHAR:
2257         break;
2258     case AST_EXPR_TYPE_BOOL:
2259         warn_implicit_cast(ast_node_get_file_location(n->rhs),
2260             AST_EXPR_TYPE_BOOL, AST_EXPR_TYPE_INT);
2261         break;
2262
2263     default:
2264         report_error_location(ast_node_get_file_location(n->rhs),
2265             S("expression of integral type expected,"
2266                 " found " QFY("%S") "\n"),
2267             ast_expr_type_to_string(v->prev_expr_type));
2268         break;
2269     }
2270     ASTVF_TC_END(AST_ALLOC_ARY, v)
2271
2272     static bool type_check_rec_has_default_rec_func(Symbol_Type_Rec *rec);
2273
2274     static bool type_check_rec_alloc(Ast_Visitor_Type_Check *v, Ast_Node *rec)
2275     {
2276         rec->accept_visitor(rec, AST_VISITOR_OF(v));
2277         if (!v->prev_variable_sym_type) {
2278             report_error_location(ast_node_get_file_location(rec),
2279                 S("invalid " QFY("allocate") " of " QFY("record")
2280                     " self-reference\n"));
2281             return false;
2282         } else if (v->prev_expr_type != AST_EXPR_TYPE_REC &&
2283             v->prev_expr_type != AST_EXPR_TYPE_UNKNOWN) {
2284             report_error_location(ast_node_get_file_location(rec),
2285                 S("expected variable of type " QFY("record")
2286                     ", found " QFY("%S") "\n"),
2287             ast_expr_type_to_string(v->prev_expr_type));
2288             return false;
2289         } else if (v->prev_variable_sym_type->methods->get_type() ==
2290             SYMBOL_TYPE_FUNC) {
2291             report_error_location(ast_node_get_file_location(rec),
2292                 S("invalid " QFY("allocate")
2293                     " of expression returned from function,"
2294                     " expected variable of type " QFY("record") "\n"));
2295             return false;
2296         } else if (v->prev_variable_sym_type->methods->get_type() !=
2297             SYMBOL_TYPE_UNKNOWN) {
2298             assert(v->prev_variable_sym_type->methods->get_type() ==
2299                 SYMBOL_TYPE_REC);
2300             if (!symbol_type_rec_assignment_compatible(v->prev_variable_sym_type,
2301                 v->prev_sym_type)) {
2302                 report_error_location(ast_node_get_file_location(rec),
2303                     S("type cast to invalid record "
2304                         "type in " QFY("allocate") " statement\n"));
2305                 return false;
2306             } else if (!v->prev_sym_type->methods->same_type(v->prev_sym_type,

```

```

2307         v->prev_variable_sym_type)) {
2308     #if 0
2309         Symbol_Type_Rec *var_rec = SYMBOL_TYPE_STRUCT_CONTAINER(
2310             v->prev_variable_sym_type, Symbol_Type_Rec);
2311         if (cmdopts.warn_no_finalize &
2312             !symbol_table_node_get_func_map(var_rec->rec_sym_node,
2313             (String)S("finalize"), string_hash_code(S("finalize")))) {
2314
2315             report_warning_location(ast_node_get_file_location(rec),
2316             S("type cast of record without " QFY("finalize")
2317             " function in " QFY("allocate") " statement\n"));
2318             /* Only warning. */
2319         }
2320     #endif
2321 }
2322 }
2323
2324     return true;
2325 }
2326
2327 ASTVF_TC_BEGIN(AST_ALLOC_REC, Ast_Visitor_Type_Check, v, Ast_Node_Unary, n)
2328     if (!type_check_rec_alloc(v, n->expr))
2329         goto out;
2330     if (v->prev_sym_type->methods->get_type() == SYMBOL_TYPE_UNKNOWN)
2331         goto out;
2332
2333     assert(v->prev_sym_type->methods->get_type() == SYMBOL_TYPE_REC);
2334
2335     Symbol_Type_Rec *rec_sym = SYMBOL_TYPE_STRUCT_CONTAINER(
2336         v->prev_sym_type, Symbol_Type_Rec);
2337     if (!type_check_rec_has_default_rec_func(rec_sym)) {
2338         if (rec_sym->rec_name)
2339             report_error_location(ast_node_get_file_location(n->expr),
2340             S("record " QFY("%S") " does not have a default "
2341             "constructor\n"),
2342             STRING_AFTER_DOT(rec_sym->rec_name));
2343         else
2344             report_error_location(ast_node_get_file_location(n->expr),
2345             S("record does not have a default constructor\n"));
2346     }
2347 out:;
2348 ASTVF_TC_END(AST_ALLOC_REC, v)
2349
2350 ASTVF_TC_BEGIN(AST_ALLOC_REC_CALL, Ast_Visitor_Type_Check, v,
2351     Ast_Node_Binary, n)
2352     if (!type_check_rec_alloc(v, n->lhs))
2353         goto out;
2354     if (v->prev_expr_type == AST_EXPR_TYPE_UNKNOWN)
2355         goto out;
2356     assert(v->prev_expr_type == AST_EXPR_TYPE_REC);
2357
2358     bool prev = v->in_allocate_stmt;
2359     v->in_allocate_stmt = true;
2360
2361     Symbol_Type_Rec *rec = SYMBOL_TYPE_STRUCT_CONTAINER(v->prev_sym_type,
2362     Symbol_Type_Rec);
2363     v->dot_ref_sym_node = rec->rec_sym_node;
2364     n->rhs->accept_visitor(n->rhs, AST_VISITOR_OF(v));
2365     v->in_allocate_stmt = prev;
2366 out:;
2367 ASTVF_TC_END(AST_ALLOC_REC, v)
2368
2369 ASTVF_TC_BEGIN(AST_DELETE, Ast_Visitor_Type_Check, v, Ast_Node_Unary, n)
2370     n->expr->accept_visitor(n->expr, AST_VISITOR_OF(v));
2371     if (v->prev_expr_type != AST_EXPR_TYPE_REC &&
2372         v->prev_expr_type != AST_EXPR_TYPE_ARY &&
2373         v->prev_expr_type != AST_EXPR_TYPE_UNKNOWN) {
2374         report_error_location(ast_node_get_file_location(n->expr),
2375         S("expected variable of type " QFY("record")
2376         " or " QFY("array") " , found " QFY("%S") "\n"),
2377         ast_expr_type_to_string(v->prev_expr_type));
2378     }
2379 ASTVF_TC_END(AST_DELETE, v)
2380

```



```

2381 ASTVF_TC_BEGIN(AST_WHILE_STMT, Ast_Visitor_Type_Check, v, Ast_Node_Binary, n)
2382   n->lhs->accept_visitor(n->lhs, AST_VISITOR_OF(v));
2383   stmt_expr_error_report(v->prev_expr_type, "while", AST_NODE_OF(n));
2384   n->rhs->accept_visitor(n->rhs, AST_VISITOR_OF(v));
2385 ASTVF_TC_END(AST_WHILE_STMT, v)
2386
2387 static bool type_check_void_return(Ast_Visitor_Type_Check *v, Ast_Node *n)
2388 {
2389     if (v->prev_expr_type == AST_EXPR_TYPE_VOID) {
2390         if (v->func_return_type != AST_EXPR_TYPE_VOID &&
2391             v->func_return_type != AST_EXPR_TYPE_UNKNOWN)
2392             report_error_location(ast_node_get_file_location(n),
2393                 S("returning " QFY("void") " from " QFY("%S")
2394                   " function\n"),
2395                 ast_expr_type_to_string(v->func_return_type));
2396         return true;
2397     } else {
2398         if (v->func_return_type == AST_EXPR_TYPE_VOID) {
2399             report_error_location(ast_node_get_file_location(n),
2400                 S("returning " QFY("%S") " from " QFY("void")
2401                   " function\n"),
2402                 ast_expr_type_to_string(v->prev_expr_type));
2403             return true;
2404         }
2405     }
2406     return false;
2407 }
2408
2409 ASTVF_TC_BEGIN(AST_RETURN_STMT, Ast_Visitor_Type_Check, v, Ast_Node_Unary, n)
2410   n->expr->accept_visitor(n->expr, AST_VISITOR_OF(v));
2411
2412   Ast_Node *ast_node = AST_NODE_OF(n);
2413   Symbol_Table_Node *sym_node = ast_node_get_symbol_table_node(ast_node);
2414
2415   if (v->sym_table) {
2416       if (sym_node->type == SYMBOL_TABLE_NODE_GLOBAL) {
2417           report_error_location(ast_node_get_file_location(ast_node),
2418               S("invalid " QFY("return")
2419                 " statement in global scope\n"));
2420           return;
2421       }
2422   }
2423
2424   if (type_check_void_return(v, AST_NODE_OF(n)))
2425       return;
2426
2427   Assign_Comatibility_Type assign_t =
2428       types_are_assignment_compatible(
2429           v->func_return_sym, v->func_return_type,
2430           v->prev_sym_type, v->prev_expr_type);
2431
2432   switch (assign_t) {
2433       case ASSIGN_COMPATIBILITY_TRUE:
2434           break;
2435       case ASSIGN_COMPATIBILITY_FALSE:
2436           if (v->prev_expr_type != v->func_return_type)
2437               report_error_location(ast_node_get_file_location(n->expr),
2438                   S("return expression of incompatible type " QFY("%S")
2439                     ", expected " QFY("%S") "\n"),
2440                   ast_expr_type_to_string(v->prev_expr_type),
2441                   ast_expr_type_to_string(v->func_return_type));
2442           else
2443               report_error_location(ast_node_get_file_location(n->expr),
2444                   S("return expression of incompatible " QFY("%S")
2445                     " type\n"),
2446                   ast_expr_type_to_string(v->prev_expr_type));
2447           break;
2448       case ASSIGN_COMPATIBILITY_AMBIGUOUS:
2449           report_error_location(
2450               ast_node_get_file_location(n->expr),
2451               S("return expression of ambiguous record type, "
2452                 "record is directly or "

```

```

2455         "indirectly extending multiple records "
2456         "with the required structure\n"));
2457     break;
2458
2459     case ASSIGN_COMPATIBILITY_IMPLICIT:
2460         warn_implicit_cast(ast_node_get_file_location(n->expr),
2461             v->prev_expr_type, v->func_return_type);
2462         break;
2463     }
2464     ASTVF_TC_END(AST_RETURN_STMT, v)
2465
2466     ASTVF_TC_BEGIN(AST_WRITE_STMT, Ast_Visitor_Type_Check, v, Ast_Node_Unary, n)
2467     n->expr->accept_visitor(n->expr, AST_VISITOR_OF(v));
2468     if (v->prev_expr_type == AST_EXPR_TYPE_VOID) {
2469         report_error_location(ast_node_get_file_location(n->expr),
2470             S("invalid " QFY("write") " of "
2471             QFY("void") " expression\n"));
2472     }
2473     ASTVF_TC_END(AST_WRITE_STMT, v)
2474
2475     static inline bool type_check_prev_was_cast(Ast_Visitor_Type_Check *v)
2476     {
2477         return v->prev_method == AST_VISITOR_FUNC(AST_EXPR_CAST);
2478     }
2479
2480     ASTVF_TC_BEGIN(AST_ASSIGNMENT, Ast_Visitor_Type_Check, v, Ast_Node_Binary, n)
2481     Ast_Expr_Type lhs_t, rhs_t;
2482     Symbol_Type_Struct *lhs_struct, *rhs_struct;
2483     v->lhs_of_assign_is_string_ref = false;
2484     n->lhs->accept_visitor(n->lhs, AST_VISITOR_OF(v));
2485     if (!type_check_prev_was_cast(v)) {
2486         if (!v->prev_variable_sym_type) {
2487             report_error_location(ast_node_get_file_location(n->lhs),
2488                 S("invalid assignment of " QFY("record")
2489                 " self-reference\n"));
2490             lhs_struct = symbol_type_unknown_alloc(v->sym_table);
2491             lhs_t = symbol_type_to_expr_type(lhs_struct);
2492         } else if (v->prev_variable_sym_type->methods->get_type() ==
2493             SYMBOL_TYPE_FUNC) {
2494             report_error_location(ast_node_get_file_location(n->lhs),
2495                 S("invalid function call on left hand side of assignment\n"));
2496             lhs_struct = symbol_type_unknown_alloc(v->sym_table);
2497             lhs_t = symbol_type_to_expr_type(lhs_struct);
2498         } else if (v->lhs_of_assign_is_string_ref) {
2499             report_error_location(ast_node_get_file_location(n->lhs),
2500                 S("invalid assignment of immutable string element\n"));
2501             lhs_struct = symbol_type_unknown_alloc(v->sym_table);
2502             lhs_t = symbol_type_to_expr_type(lhs_struct);
2503         } else {
2504             lhs_struct = v->prev_sym_type;
2505             lhs_t = v->prev_expr_type;
2506         }
2507     } else {
2508         lhs_struct = symbol_type_unknown_alloc(v->sym_table);
2509         lhs_t = symbol_type_to_expr_type(lhs_struct);
2510         report_error_location(ast_node_get_file_location(n->lhs),
2511             S("invalid type cast on left hand side of assignment\n"));
2512     }
2513
2514     n->rhs->accept_visitor(n->rhs, AST_VISITOR_OF(v));
2515     rhs_struct = v->prev_sym_type;
2516     rhs_t = v->prev_expr_type;
2517
2518     Assign_Comatibility_Type assign_t =
2519         types_are_assignment_compatible(lhs_struct, lhs_t,
2520             rhs_struct, rhs_t);
2521     switch (assign_t) {
2522     case ASSIGN_COMPATIBILITY_TRUE:
2523         break;
2524
2525     case ASSIGN_COMPATIBILITY_FALSE:
2526         if (lhs_t != rhs_t)
2527             report_error_location(ast_node_get_file_location(AST_NODE_OF(n)),
2528                 S("incompatible assignment of " QFY("%S") " type from "

```

```

2529         QFY("%S") " type\n"),
2530         ast_expr_type_to_string(lhs_t),
2531         ast_expr_type_to_string(rhs_t));
2532     else
2533         report_error_location(ast_node_get_file_location(AST_NODE_OF(n)),
2534             S("assignment with incompatible " QFY("%S") " types\n"),
2535             ast_expr_type_to_string(lhs_t));
2536     break;
2537
2538     case ASSIGN_COMPATIBILITY_AMBIGUOUS:
2539         report_error_location(
2540             ast_node_get_file_location(AST_NODE_OF(n)),
2541             S("ambiguous record assignment, "
2542               "right hand side record is directly or "
2543               "indirectly extending multiple records "
2544               "with the same structure as the left hand side\n"));
2545     break;
2546
2547     case ASSIGN_COMPATIBILITY_IMPLICIT:
2548         /* Implicit cast from rhs to lhs. */
2549         warn_implicit_cast(ast_node_get_file_location(AST_NODE_OF(n)),
2550             rhs_t, lhs_t);
2551     break;
2552 }
2553 ASTVF_TC_END(AST_ASSIGNMENT, v)
2554
2555 static void type_check_func_body(Ast_Visitor_Type_Check *v,
2556     Vector *statements, Uns start_idx)
2557 {
2558     Ast_Node *stmt;
2559     for (; start_idx < vector_size(statements); start_idx++) {
2560         stmt = vector_get(statements, start_idx);
2561         stmt->accept_visitor(stmt, AST_VISITOR_OF(v));
2562     }
2563 }
2564
2565 typedef struct Tt_Extend_Mapping {
2566     Symbol_Type_Struct *extended_rec;
2567     Uns call_count;
2568     bool constructor_called;
2569 } Tt_Extend_Mapping;
2570
2571 static UNUSED bool type_check_extend_maps_all_called(Vector *ext_maps)
2572 {
2573     Tt_Extend_Mapping *m;
2574     VECTOR_FOR_EACH_ENTRY(ext_maps, m) {
2575         if (!m->constructor_called)
2576             return false;
2577     }
2578     return true;
2579 }
2580
2581 static bool type_check_rec_has_default_rec_func(Symbol_Type_Rec *rec)
2582 {
2583     Symbol_Func_Map *map = symbol_table_node_get_func_map(rec->rec_sym_node,
2584         (String)S("record"), string_hash_code(S("record")));
2585     if (!map)
2586         return true;
2587
2588     Symbol_Type_Func *func;
2589     Symbol *sym;
2590     VECTOR_FOR_EACH_ENTRY(&map->overload_idens, sym) {
2591         if (sym->resolved_type->methods->get_type() == SYMBOL_TYPE_UNKNOWN)
2592             return true;
2593         assert(sym->resolved_type->methods->get_type() == SYMBOL_TYPE_FUNC);
2594
2595         func = SYMBOL_TYPE_STRUCT_CONTAINER(
2596             sym->resolved_type, Symbol_Type_Func);
2597         if (!vector_size(&func->param_identifiers))
2598             return true;
2599     }
2600
2601     return false;
2602 }

```

```

2603
2604 static inline void type_check_init_extend_maps(Symbol_Type_Rec *rec,
2605 Vector *ext_maps)
2606 {
2607     Symbol_Type_Struct *ext_s;
2608     Symbol_Type_Rec *ext_r;
2609     VECTOR_FOR_EACH_ENTRY(&rec->extended_types, ext_s) {
2610         if (ext_s->methods->get_type() == SYMBOL_TYPE_UNKNOWN)
2611             continue;
2612         assert(ext_s->methods->get_type() == SYMBOL_TYPE_REC);
2613         ext_r = SYMBOL_TYPE_STRUCT_CONTAINER(ext_s, Symbol_Type_Rec);
2614         if (ext_r->rec_sym_node->has_record_func) {
2615             Tt_Extend_Mapping *m = ALLOC_NEW(Tt_Extend_Mapping);
2616             m->extended_rec = ext_s;
2617             m->call_count = 0;
2618             m->constructor_called = type_check_rec_has_default_rec_func(ext_r);
2619             vector_append(ext_maps, m);
2620         }
2621     }
2622 }
2623
2624 static inline void type_check_extend_maps_report(Vector *ext_maps,
2625 Uns num_unnamed_bases, File_Location *loc)
2626 {
2627     if (num_unnamed_bases) {
2628         report_error_location(loc, S("missing call to base record "
2629 "constructor%s\n"),
2630 num_unnamed_bases > 1 ? "s" : "");
2631         return;
2632     }
2633
2634     Symbol_Type_Rec *rec;
2635     Tt_Extend_Mapping *m;
2636     VECTOR_FOR_EACH_ENTRY(ext_maps, m) {
2637         if (m->constructor_called)
2638             continue;
2639         rec = SYMBOL_TYPE_STRUCT_CONTAINER(m->extended_rec, Symbol_Type_Rec);
2640         report_error_location(loc, S("missing call to constructor"
2641 " of base record " QFY("%S") "\n"),
2642 STRING_AFTER_DOT(rec->rec_name));
2643     }
2644 }
2645
2646 static inline void type_check_finalize_extend_map_report(Vector *ext_maps,
2647 File_Location *final_loc)
2648 {
2649     Uns num_unnamed = 0;
2650     bool has_uncalled_ctor = false;
2651
2652     Symbol_Type_Rec *rec;
2653     Tt_Extend_Mapping *m;
2654     VECTOR_FOR_EACH_ENTRY(ext_maps, m) {
2655         rec = SYMBOL_TYPE_STRUCT_CONTAINER(m->extended_rec, Symbol_Type_Rec);
2656         if (!m->constructor_called) {
2657             if (!rec->rec_name)
2658                 ++num_unnamed;
2659             has_uncalled_ctor = true;
2660         }
2661     }
2662
2663     if (has_uncalled_ctor)
2664         type_check_extend_maps_report(ext_maps, num_unnamed, final_loc);
2665 }
2666
2667 static bool type_check_extend_maps_mark_called(Vector *ext_maps,
2668 Symbol_Type_Struct *base, File_Location *loc)
2669 {
2670     assert(base->methods->get_type() == SYMBOL_TYPE_REC);
2671
2672     bool ret = false;
2673     Tt_Extend_Mapping *m;
2674     VECTOR_FOR_EACH_ENTRY(ext_maps, m) {
2675         if (__type_def_symbol_types_equal(base, m->extended_rec)) {
2676             ret = true;

```

```

2677         ++m->call_count;
2678         m->constructor_called = true;
2679         break;
2680     }
2681 }
2682
2683 if (ret && m->call_count == 2) {
2684     Symbol_Type_Rec *base_r = SYMBOL_TYPE_STRUCT_CONTAINER(base,
2685         Symbol_Type_Rec);
2686     Const_String rec_name;
2687     if (base_r->rec_name)
2688         rec_name = base_r->rec_name;
2689     else
2690         rec_name = S("record");
2691     report_error_location(loc, S("multiple calls to constructor of base "
2692         QFY("%S") "\n"),
2693         rec_name);
2694 }
2695
2696 return ret;
2697 }
2698
2699 enum {
2700     TC_NOT_RECORD_CALL,
2701     TC_BASE_RECORD_CALL,
2702     TC_FORWARD_RECORD_CALL
2703 };
2704
2705 static inline int type_check_extend_maps_is_base_call(Vector *ext_maps,
2706     Symbol_Type_Struct *rec, Symbol_Type_Struct *base,
2707     Const_String func_iden, File_Location *loc)
2708 {
2709     if (string_compare(String_After_Dot(func_iden), S("record")))
2710         return TC_NOT_RECORD_CALL;
2711
2712     if (type_check_extend_maps_mark_called(ext_maps, base, loc))
2713         return TC_BASE_RECORD_CALL;
2714
2715     if (___type_def_symbol_types_equal(rec, base))
2716         return TC_FORWARD_RECORD_CALL;
2717
2718     return TC_NOT_RECORD_CALL;
2719 }
2720
2721 static void type_check_direct_ctor_error(Symbol_Type_Rec *rec,
2722     File_Location *loc)
2723 {
2724     if (rec->rec_name)
2725         report_error_location(loc, S("forwarding call to constructor of "
2726             "record " QFY("%S")
2727             " should be first statement of function\n"),
2728             rec->rec_name);
2729     else
2730         report_error_location(loc, S("forwarding call to constructor of "
2731             "record should be first statement of function\n"),
2732             rec->rec_name);
2733 }
2734
2735 static Uns type_check_func_constructors(Ast_Visitor_Type_Check *v,
2736     Vector *statements, Ast_Stmt_List *n)
2737 {
2738     Ast_Node *stmt;
2739     Uns idx = 0;
2740
2741     VECTOR(ext_maps);
2742
2743     if (!v->rec_body_sym_node)
2744         goto unknown_out;
2745
2746     File_Location *loc = ast_node_get_file_location(AST_NODE_OF(n));
2747
2748     Symbol_Type_Struct *curr_rec = SYMBOL_TYPE_STRUCT_CONTAINER(
2749         v->rec_body_sym_node->node_rec);
2750

```

```

2751     bool do_finalize = true;
2752
2753     type_check_init_extend_maps(v->rec_body_sym_node->node_rec, &ext_maps);
2754
2755     while (idx < vector_size(statements)) {
2756         stmt = vector_get(statements, idx);
2757         loc = ast_node_get_file_location(stmt);
2758
2759         if (stmt->accept_visitor !=
2760             AST_NODE_ACCEPT_VISITOR_FUNC(AST_EXPR_DIRECT_REF))
2761             break;
2762
2763         v->prev_direct_ref_func = NULL;
2764         v->prev_direct_ref_rec = NULL;
2765         stmt->accept_visitor(stmt, AST_VISITOR_OF(v));
2766         ++idx;
2767
2768         if (v->prev_direct_ref_func && v->prev_direct_ref_rec) {
2769             Symbol_Type_Struct *base = v->prev_direct_ref_rec;
2770             Symbol_Type_Struct *func = v->prev_direct_ref_func;
2771             if (base->methods->get_type() == SYMBOL_TYPE_UNKNOWN ||
2772                 func->methods->get_type() == SYMBOL_TYPE_UNKNOWN) {
2773                 goto unknown_out;
2774             }
2775             int res = type_check_extend_maps_is_base_call(&ext_maps,
2776                                                         curr_rec, base, v->prev_func_call_iden, loc);
2777             if (res == TC_NOT_RECORD_CALL)
2778                 break;
2779             if (res == TC_FORWARD_RECORD_CALL) {
2780                 if (idx != 1) {
2781                     type_check_direct_ctor_error(SYMBOL_TYPE_STRUCT_CONTAINER(
2782                                                         base, Symbol_Type_Rec), loc);
2783                 } else {
2784                     do_finalize = false;
2785                     break;
2786                 }
2787             }
2788             } else {
2789                 break;
2790             }
2791     }
2792
2793     n->num_rec_ctor_stmts = idx;
2794
2795     if (do_finalize)
2796         type_check_finalize_extend_map_report(&ext_maps, loc);
2797
2798     unknown_out:
2799     vector_for_each_destroy(&ext_maps, free_mem);
2800     return idx;
2801 }
2802
2803 static void type_check_rec_default_constructor(Symbol_Type_Rec *rec,
2804                                               File_Location *loc)
2805 {
2806     if (!rec->missing_record_func_name)
2807         return;
2808
2809     VECTOR(ext_maps);
2810     type_check_init_extend_maps(rec, &ext_maps);
2811     type_check_finalize_extend_map_report(&ext_maps, loc);
2812     vector_for_each_destroy(&ext_maps, free_mem);
2813 }
2814
2815 ASTVF_TC_BEGIN(AST_FUNC_BODY, Ast_Visitor_Type_Check, v, Ast_Stmt_List, n)
2816     Vector *statements = n->statements;
2817     Uns idx;
2818
2819     bool prev = v->in_beginning_of_ctor;
2820     v->in_beginning_of_ctor = true;
2821     if (v->current_func_type == TT_FUNC_TYPE_RECORD)
2822         idx = type_check_func_constructors(v, statements, n);
2823     else
2824         idx = 0;

```

```

2825     v->in_beginning_of_ctor = prev;
2826     type_check_func_body(v, statements, idx);
2827     ASTVF_TC_END(AST_FUNC_BODY, v)
2828
2829     ASTVF_TC_BEGIN(AST_EXPR_CHAR, Ast_Visitor_Type_Check, v, Ast_Expr_Char, n)
2830     (void)n;
2831     v->prev_expr_type = AST_EXPR_TYPE_CHAR;
2832     ASTVF_TC_END(AST_EXPR_CHAR, v)
2833
2834     ASTVF_TC_BEGIN(AST_EXPR_STRING, Ast_Visitor_Type_Check, v,
2835         Ast_Expr_String, n)
2836     (void)n;
2837     v->prev_expr_type = AST_EXPR_TYPE_STRING;
2838     ASTVF_TC_END(AST_EXPR_STRING, v)
2839
2840     ASTVF_TC_BEGIN(AST_IMPORT_STRING, Ast_Visitor_Type_Check, v,
2841         Ast_Module_String, n)
2842     (void)n; (void)v;
2843     ASTVF_TC_END(AST_IMPORT_STRING, v)
2844
2845     ASTVF_TC_BEGIN(AST_PACKAGE_STRING, Ast_Visitor_Type_Check, v,
2846         Ast_Module_String, n)
2847     (void)n; (void)v;
2848     ASTVF_TC_END(AST_IMPORT_STRING, v)
2849
2850     ASTVF_TC_BEGIN(AST_SIMPLE_TYPE_CHAR, Ast_Visitor_Type_Check, v, Ast_Empty, n)
2851     (void)n;
2852     v->prev_expr_type = AST_EXPR_TYPE_CHAR;
2853     ASTVF_TC_END(AST_SIMPLE_TYPE_CHAR, v)
2854
2855     ASTVF_TC_BEGIN(AST_SIMPLE_TYPE_VOID, Ast_Visitor_Type_Check, v, Ast_Empty, n)
2856     (void)n;
2857     v->prev_expr_type = AST_EXPR_TYPE_VOID;
2858     ASTVF_TC_END(AST_SIMPLE_TYPE_CHAR, v)
2859
2860     ASTVF_TC_BEGIN(AST_SIMPLE_TYPE_STRING, Ast_Visitor_Type_Check, v, Ast_Empty, n)
2861     (void)n;
2862     v->prev_expr_type = AST_EXPR_TYPE_STRING;
2863     ASTVF_TC_END(AST_SIMPLE_TYPE_STRING, v)
2864
2865     static Ast_Visitor_Type_Check type_check_visitor = {
2866         .AST_VISITOR_FIELD = AST_VISITOR_STATIC_INIT(),
2867         .next_property = SYMBOL_PROPERTY_VAR
2868     };
2869
2870     void ast_visitor_type_check(Ast *ast)
2871     {
2872         Ast_Node *root;
2873         if (ast_is_valid(ast)) {
2874             type_check_visitor.sym_table = ast_get_symbol_table(ast);
2875             type_check_visitor.prev_sym_type = NULL;
2876             root = ast_get_root(ast);
2877             root->accept_visitor(root, AST_VISITOR_OF(&type_check_visitor));
2878             assert(!type_check_visitor.rec_body_sym_node);
2879             assert(type_check_visitor.next_property == SYMBOL_PROPERTY_VAR);
2880         }
2881     }

```

:

A.5.2 src/ast/ast_visitor_type_check.h

```

1  #ifndef AST_VISITOR_TYPE_CHECK_H
2  #define AST_VISITOR_TYPE_CHECK_H
3
4  #include "ast_visitor.h"
5
6  void ast_visitor_type_check(Ast *ast);
7
8  #endif // AST_VISITOR_TYPE_CHECK_H

```

A.6 Generate AIA

:

A.6.1 src/ast/ast_visitor_aia.c

```

1  #include "ast_visitor_aia.h"
2  #include "ast_visitor_dependency.h"
3  #include "symbol_table.h"
4  #include <main.h>
5  #include <debug.h>
6  #include <hash_map.h>
7  #include <aia/aia.h>
8
9  /* let DEBUG=aia-gen:aia-gen-full
10   * for more debug info. */
11 #undef DEBUG_TYPE
12 #define DEBUG_TYPE aia-gen
13
14 #define BYTE_ALIGNMENT ((Uns)1)
15 #define LONG_ALIGNMENT ((Uns)4)
16 #define LONG_ALIGN_MASK (LONG_ALIGNMENT - 1)
17
18 typedef enum Aia_Expr_Type {
19     AIA_EXPR_DEBUG,
20     AIA_EXPR_SHORT_CIRCUIT,
21     AIA_EXPR_STANDARD
22 } Aia_Expr_Type;
23
24 typedef enum Aia_Visitor_Func_Type {
25     AIAV_FUNC_TYPE_NORMAL,
26     AIAV_FUNC_TYPE_FINALIZE,
27     AIAV_FUNC_TYPE_RECORD
28 } Aia_Visitor_Func_Type;
29
30 AST_VISITOR_STRUCT_BEGIN(Ast_Visitor_Aia)
31     Symbol_Table *sym_table;
32     Aia *aia;
33     Aia_Operand *prev_result;
34     Aia_Operand *func_call_selfptr;
35     Aia_Operand *prev_variable_result;
36     Aia_Operand *bool_true_label;
37     Aia_Operand *bool_false_label;
38     Symbol_Type_Struct *prev_sym_type;
39     Symbol_Type_Struct *func_ret_sym_type;
40     Symbol_Type_Rec *dot_ref_record;
41     Symbol_Type_Rec *curr_record_selfptr;
42     Symbol_Type_Rec *alloc_record_selfptr;
43     Symbol_Type_Rec *prev_dot_ref_record;
44     Stn_Display_Preserve *curr_display;
45     Symbol *curr_symbol;
46     String prev_iden;
47     Hash_Map record_map;
48     Hash_Map vmt_map;
49     Hash_Map trampolines;
50     Symbol_Property next_property;
51     File_Location null_location;
52     File_Location *curr_loc;
53     Uns stmt_list_nest;
54     Ast_Expr_Type prev_expr_type;
55     Aia_Expr_Type aia_expr_type;
56     Aia_Visitor_Func_Type curr_func_type;
57     int32_t allocated_record_offset;
58     /* Used to determine whether a variable identifier is used as a variable
59      * or whether it is just declared. */
60     bool next_variable_iden_is_operand;
61     bool dot_ref_is_direct;
62     bool next_var_decl_is_param;
63     bool allocating_record;
64 AST_VISITOR_STRUCT_END(Ast_Visitor_Aia)
65

```



```

66 typedef enum Vit_Field_Type {
67     VIT_RECORD_FIELD_BYTE,
68     VIT_RECORD_FIELD_LONG,
69     VIT_RECORD_FIELD_RECORD
70 } Vit_Field_Type;
71
72 typedef struct Vit_Record_Field {
73     String field_name;
74     Hash_Map_Slot hash_slot;
75     Int field_offset;
76     Vit_Field_Type field_type;
77     String field_type_name;
78     Symbol_Type sym_type;
79 } Vit_Record_Field;
80
81 typedef struct Vit_Vmt_Trampoline {
82     String tramp_name;
83     String func_name;
84     Int rec_offset;
85 } Vit_Vmt_Trampoline;
86
87 #define VIT_RECORD_FIELD_OF(slot) \
88     HASH_MAP_ENTRY(slot, Vit_Record_Field, hash_slot);
89
90 typedef struct Vit_Vmt_Trampoline_Entry {
91     Hash_Map_Slot hash_slot;
92     Vector trampolines;
93 } Vit_Vmt_Trampoline_Entry;
94
95 #define VIT_VMT_TRAMPOLINE_ENTRY_OF(slot) \
96     HASH_MAP_ENTRY(slot, Vit_Vmt_Trampoline_Entry, hash_slot)
97
98 typedef struct Vit_Func_Off_Entry {
99     String func_name;
100     Hash_Map_Slot hash_slot;
101     Uns func_offset;
102     Uns cast_offset;
103 } Vit_Func_Off_Entry;
104
105 #define VIT_FUNC_OFF_ENTRY_OF(slot) \
106     HASH_MAP_ENTRY(slot, Vit_Func_Off_Entry, hash_slot)
107
108 typedef struct Vit_Record_Initializer {
109     String initializer_name;
110     /* Vector containing record offsets. See comment for vmt_offsets.*/
111     Vector rec_offsets;
112     /* Vector containing offsets into this VMT.
113      * rec_offset[1] contains the offset to record with name 'record_name'
114      * which should get initialized with vmt: this VMT + vmt_offsets[1]. */
115     Vector vmt_offsets;
116 } Vit_Record_Initializer;
117
118 typedef struct Vit_Vmt {
119     /* Name of the record containing this VMT. */
120     String record_name;
121     /* Names of functions and function trampolines in the VMT. */
122     Vector func_names;
123     /* Offsets to functions in the VMT from the owning record's perspective. */
124     Hash_Map func_off_map;
125     /* struct to generate function to initialize record VMT's. */
126     Vit_Record_Initializer initializer;
127     Hash_Map_Slot hash_slot;
128     /* Offset to next function which will get inserted.
129      * After all funcs are inserted it can be used to
130      * determine the size if the vmt. */
131     Uns current_func_offset;
132     bool is_imported;
133 } Vit_Vmt;
134
135 #define VIT_VMT_OF(slot) HASH_MAP_ENTRY(slot, Vit_Vmt, hash_slot)
136
137 typedef struct Vit_Record {
138     String identifier;
139     Const_String initializer_name;

```

```

140     Symbol_Type_Rec *rec;
141     Vector extended_sym_types;
142     Vector field_vector;
143     Hash_Map field_map;
144     Hash_Map_Slot hash_slot;
145     Uns byte_size;
146     Uns alignment;
147     bool is_c_header_printed;
148 } Vit_Record;
149
150 #define VIT_RECORD_OF(slot) HASH_MAP_ENTRY(slot, Vit_Record, hash_slot)
151
152 typedef struct Vit_Record_Func_Entry {
153     String func_name;
154     String long_func_name;
155     Hash_Map_Slot hash_slot;
156     bool is_inserted;
157     Symbol_Type_Rec *owner;
158 } Vit_Record_Func_Entry;
159
160 #define VIT_RECORD_FUNC_ENTRY_OF(slot) \
161     HASH_MAP_ENTRY(slot, Vit_Record_Func_Entry, hash_slot)
162
163 static Const_String vit_field_type_strings[] = {
164     [VIT_RECORD_FIELD_BYTE] = S(".byte"),
165     [VIT_RECORD_FIELD_LONG] = S(".long"),
166     [VIT_RECORD_FIELD_RECORD] = NULL
167 };
168
169 static NORETURN void aia_visitor_unexpected_symbol()
170 {
171     fatal_error(S("AIA code generator encountered unexpected "
172                 "symbol type. Aborting...\n"));
173 }
174
175 static inline Const_String vit_simple_field_type_to_string(Vit_Field_Type t)
176 {
177     Const_String ret = vit_field_type_strings[t];
178     assert(ret);
179     return ret;
180 }
181
182 static inline Vit_Record_Field *vit_record_get_field(Vit_Record *r,
183     String field_name)
184 {
185     Hash_Map_Slot *s = hash_map_get(&r->field_map, field_name,
186         string_hash_code(field_name));
187     if (s)
188         return VIT_RECORD_FIELD_OF(s);
189     return NULL;
190 }
191
192 static inline Const_String vit_record_field_get_type_name(Vit_Record_Field *f)
193 {
194     if (f->field_type == VIT_RECORD_FIELD_RECORD)
195         return f->field_type_name;
196     return vit_simple_field_type_to_string(f->field_type);
197 }
198
199 static bool vit_record_field_comparator(String search_name,
200     Hash_Map_Slot *map_slot)
201 {
202     Vit_Record_Field *f = VIT_RECORD_FIELD_OF(map_slot);
203     return !string_compare(search_name, f->field_name);
204 }
205
206 static Vit_Record_Field *__vit_record_field_alloc(
207     String field_type_name, Vit_Field_Type t,
208     Symbol_Type sym_type, Const_String field_name)
209 {
210     Vit_Record_Field *f = ALLOC_NEW(Vit_Record_Field);
211     f->field_type_name = field_type_name;
212     f->field_name = string_duplicate(field_name);
213     f->field_type = t;

```

```

214     f->sym_type = sym_type;
215     return f;
216 }
217
218 static void vit_record_field_destroy(Vit_Record_Field *f)
219 {
220     string_destroy(f->field_name);
221     free_mem(f);
222 }
223
224 static Vit_Record_Field *vit_record_field_alloc_simple(Vit_Field_Type t,
225     Symbol_Type sym_type, Const_String field_name)
226 {
227     return __vit_record_field_alloc(NULL, t, sym_type, field_name);
228 }
229
230 static Vit_Record_Field *vit_record_field_alloc_record(String type_name,
231     Const_String field_name)
232 {
233     return __vit_record_field_alloc(type_name, VIT_RECORD_FIELD_RECORD,
234     SYMBOL_TYPE_IDEN, field_name);
235 }
236
237 static inline Vit_Vmt_Trampoline *vit_vmt_trampoline_alloc(String func_name,
238     Const_String unique_rec_name, Int rec_offset)
239 {
240     Vit_Vmt_Trampoline *t = ALLOC_NEW(Vit_Vmt_Trampoline);
241     t->func_name = func_name;
242     t->tramp_name = string_from_format(S("%S%S"), unique_rec_name,
243     func_name);
244     t->rec_offset = rec_offset;
245     return t;
246 }
247
248 static inline void vit_vmt_trampoline_destroy(Vit_Vmt_Trampoline *t)
249 {
250     string_destroy(t->tramp_name);
251     free_mem(t);
252 }
253
254 static UNUSED void vit_vmt_trampoline_dump(Vit_Vmt_Trampoline *t)
255 {
256     print_message(S("trampoline %S subtracts %U from thisptr\n\n"),
257     t->tramp_name, t->rec_offset);
258 }
259
260 static inline Vit_Vmt_Trampoline_Entry *vit_vmt_trampoline_entry_alloc()
261 {
262     Vit_Vmt_Trampoline_Entry *e = ALLOC_NEW(Vit_Vmt_Trampoline_Entry);
263     e->trampolines = VECTOR_INIT_SIZE(4);
264     return e;
265 }
266
267 static inline void vit_vmt_trampoline_entry_destroy(
268     Vit_Vmt_Trampoline_Entry *e)
269 {
270     vector_for_each_destroy(&e->trampolines,
271     (Vector_Destructor) vit_vmt_trampoline_destroy);
272     free_mem(e);
273 }
274
275 static void vit_vmt_trampoline_entry_hash_destroy(Hash_Map_Slot *s)
276 {
277     vit_vmt_trampoline_entry_destroy(VIT_VMT_TRAMPOLINE_ENTRY_OF(s));
278 }
279
280 static bool vit_vmt_trampoline_entry_comparator(String search_func,
281     Hash_Map_Slot *map_slot)
282 {
283     Vit_Vmt_Trampoline_Entry *e = VIT_VMT_TRAMPOLINE_ENTRY_OF(map_slot);
284     Vit_Vmt_Trampoline *t = vector_get(&e->trampolines, 0);
285     return !string_compare(t->func_name, search_func);
286 }
287

```

```

288 /* Does not matter whether the trampoline is already inserted.
289 * This function will detect it.
290 * Returns the name of the trampoline. */
291 static inline String aia_visitor_insert_trampoline(Ast_Visitor_Aia *v,
292           Vit_Vmt_Trampoline *t)
293 {
294     Uns hash = string_hash_code(t->func_name);
295     Hash_Map_Slot *s = hash_map_get(&v->trampolines, t->func_name, hash);
296     Vit_Vmt_Trampoline_Entry *e;
297     if (!s) {
298         e = vit_vmt_trampoline_entry_alloc();
299         hash_map_insert(&v->trampolines, &e->hash_slot,
300                       string_hash_code(t->func_name));
301     } else {
302         e = VIT_VMT_TRAMPOLINE_ENTRY_OF(s);
303         Vector *tramps = &e->trampolines;
304         Vit_Vmt_Trampoline *existing;
305         VECTOR_FOR_EACH_ENTRY(tramps, existing) {
306             if (!string_compare(existing->tramp_name, t->tramp_name)) {
307                 vit_vmt_trampoline_destroy(t);
308                 return existing->tramp_name;
309             }
310         }
311     }
312     DEBUGT(aia-gen-full, vit_vmt_trampoline_dump(t));
313     vector_append(&e->trampolines, t);
314     return t->tramp_name;
315 }
316
317 static inline Vit_Func_Off_Entry *vit_func_off_entry_alloc(String func_name,
318           Uns func_off, Uns cast_off)
319 {
320     Vit_Func_Off_Entry *e = ALLOC_NEW(Vit_Func_Off_Entry);
321     e->func_name = func_name;
322     e->func_offset = func_off;
323     e->cast_offset = cast_off;
324     return e;
325 }
326
327 static inline void vit_func_off_entry_destroy(Vit_Func_Off_Entry *e)
328 {
329     free_mem(e);
330 }
331
332 static void vit_func_off_entry_hash_destroy(Hash_Map_Slot *s)
333 {
334     vit_func_off_entry_destroy(VIT_FUNC_OFF_ENTRY_OF(s));
335 }
336
337 static bool vit_func_off_entry_comparator(String search_func,
338           Hash_Map_Slot *map_slot)
339 {
340     Vit_Func_Off_Entry *f = VIT_FUNC_OFF_ENTRY_OF(map_slot);
341     return !string_compare(search_func, f->func_name);
342 }
343
344 static UNUSED void vit_vmt_dump(Vit_Vmt *vmt)
345 {
346     print_message(S("initializer %S {\n"), vmt->initializer.initializer_name);
347     for (Uns i = 0; i < vector_size(&vmt->initializer.rec_offsets); i++) {
348         Uns roff = PTR_TO_INT(vector_get(&vmt->initializer.rec_offsets, i));
349         Uns voff = PTR_TO_INT(vector_get(&vmt->initializer.vmt_offsets, i));
350         print_message(S("\trecord offset %U is initialized with "
351           "vmt offset %U\n"),
352           roff, voff);
353     }
354     print_message(S("}\n\n"));
355     print_message(S("vmt %S {\n"), vmt->record_name);
356     Hash_Map_Slot *slot;
357     HASH_MAP_FOR_EACH(&vmt->func_off_map, slot) {
358         Vit_Func_Off_Entry *e = VIT_FUNC_OFF_ENTRY_OF(slot);
359         print_message(S("\tfunc %S has offset %U add thisptr "

```

```

362         "with %U before call\n"),
363         e->func_name, e->func_offset, e->cast_offset);
364     }
365     print_message(S("\n"));
366
367     String fname;
368     Uns off = 0;
369     VECTOR_FOR_EACH_ENTRY(&vmt->func_names, fname) {
370         print_message(S("\t%U: .long %S\n"), off, fname);
371         off += LONG_ALIGNMENT;
372     }
373
374     print_message(S("{}\n\n"));
375 }
376
377 static inline void vit_vmt_append_rec_vmt_offset(Vit_Vmt *v,
378         Uns rec_off, Uns vmt_off)
379 {
380     if (vector_is_empty(&v->initializer.rec_offsets) ||
381         PTR_TO_UINT(vector_peek_last(&v->initializer.rec_offsets)) !=
382         rec_off) {
383         vector_append(&v->initializer.rec_offsets, INT_TO_PTR(rec_off));
384         vector_append(&v->initializer.vmt_offsets, INT_TO_PTR(vmt_off));
385     }
386 }
387
388 static inline bool rec_has_vmt_func(Symbol_Type_Rec *rec);
389
390 static inline Vit_Vmt *vit_vmt_alloc(Symbol_Type_Rec *rec)
391 {
392     if (!rec_has_vmt_func(rec))
393         return NULL;
394
395     Vit_Vmt *v = ALLOC_NEW(Vit_Vmt);
396     v->record_name = rec->unique_name;
397     v->func_names = VECTOR_INIT_SIZE(4);
398     v->initializer.rec_offsets = VECTOR_INIT_SIZE(4);
399     v->initializer.vmt_offsets = VECTOR_INIT_SIZE(4);
400     v->initializer.initializer_name = string_from_format(S("%S.init"),
401         rec->unique_name);
402     v->func_off_map = HASH_MAP_INIT_SIZE(HASH_MAP_SIZE_11,
403         (Hash_Map_Comparator) vit_func_off_entry_comparator);
404     v->current_func_offset = 0;
405     v->is_imported = rec->rec_sym_node->parent->type == SYMBOL_TABLE_NODE_IMPORT;
406     return v;
407 }
408
409 static inline void vit_vmt_destroy(Vit_Vmt *v)
410 {
411     vector_clear(&v->func_names);
412     vector_clear(&v->initializer.rec_offsets);
413     vector_clear(&v->initializer.vmt_offsets);
414     string_destroy(v->initializer.initializer_name);
415     hash_map_for_each_destroy(&v->func_off_map,
416         vit_func_off_entry_hash_destroy);
417     free_mem(v);
418 }
419
420 /* Does not matter whether the func is already appended. */
421 static inline void vit_vmt_append_func(Vit_Vmt *v,
422     String unique_func_name, String short_func_name, Uns rec_offset)
423 {
424     vector_append(&v->func_names, unique_func_name);
425     Uns hash = string_hash_code(short_func_name);
426     if (!hash_map_contains(&v->func_off_map, short_func_name, hash)) {
427         Vit_Func_Off_Entry *e = vit_func_off_entry_alloc(short_func_name,
428             v->current_func_offset, rec_offset);
429         hash_map_insert(&v->func_off_map, &e->hash_slot, hash);
430     }
431     v->current_func_offset += LONG_ALIGNMENT;
432 }
433
434 static bool vit_vmt_comparator(String search_rec, Hash_Map_Slot *map_slot)

```

```

436 {
437     Vit_Vmt *v = VIT_VMT_OF(map_slot);
438     return !string_compare(search_rec, v->record_name);
439 }
440
441 static void vit_vmt_hash_destroy(Hash_Map_Slot *s)
442 {
443     vit_vmt_destroy(VIT_VMT_OF(s));
444 }
445
446 static UNUSED void vit_record_dump(Vit_Record *r)
447 {
448     Vit_Record_Field *f;
449     print_message(S("record %S {\n"), r->identifier);
450     VECTOR_FOR_EACH_ENTRY(&r->field_vector, f)
451         print_message(S("\t%U: %S of type %S\n"), f->field_offset,
452             f->field_name, vit_record_field_get_type_name(f));
453     if (r->initializer_name)
454         print_message(S("} of size %U, initializer: %S\n\n"), r->byte_size,
455             r->initializer_name);
456     else
457         print_message(S("} of size %U, initializer: (null)\n\n"),
458             r->byte_size);
459 }
460
461 static inline Vit_Record *vit_record_alloc(Symbol_Type_Rec *rec)
462 {
463     Vit_Record *vrec = ALLOC_NEW(Vit_Record);
464     vrec->identifier = rec->unique_name;
465     vrec->initializer_name = NULL;
466     vrec->rec = rec;
467     vrec->extended_sym_types = VECTOR_INIT_SIZE(2);
468     vrec->field_vector = VECTOR_INIT();
469     vrec->field_map = HASH_MAP_INIT_SIZE(HASH_MAP_SIZE_23,
470         (Hash_Map_Comparator) vit_record_field_comparator);
471     vrec->byte_size = 0;
472     vrec->is_c_header_printed = false;
473     return vrec;
474 }
475
476 static inline void vit_record_destroy(Vit_Record *r)
477 {
478     vector_clear(&r->extended_sym_types);
479     vector_for_each_destroy(&r->field_vector,
480         (Vector_Destructor) vit_record_field_destroy);
481     hash_map_clear(&r->field_map);
482     free_mem(r);
483 }
484
485 static inline void vit_record_hash_destroy(Hash_Map_Slot *s)
486 {
487     vit_record_destroy(VIT_RECORD_OF(s));
488 }
489
490 static inline Vit_Record *aia_visitor_get_record(Ast_Visitor_Aia *v,
491     String iden);
492
493 static bool aia_visitor_get_rec_base_offset(Ast_Visitor_Aia *v, Vit_Record *r,
494     Symbol_Type_Struct *base_rec, int32_t *result)
495 {
496     Vit_Record_Field *f;
497     Symbol_Type_Struct *ext_s;
498     Vector *ext_vec = &r->extended_sym_types;
499     VECTOR_FOR_EACH_ENTRY(ext_vec, ext_s) {
500         Symbol_Type_Rec *ext_r = SYMBOL_TYPE_STRUCT_CONTAINER(ext_s,
501             Symbol_Type_Rec);
502         f = vit_record_get_field(r, ext_r->unique_name);
503         if (base_rec->methods->same_type(base_rec, ext_s)) {
504             *result = f->field_offset;
505             return true;
506         }
507     }
508     Vit_Record *base = aia_visitor_get_record(v, ext_r->unique_name);
509     assert(base);
510     if (aia_visitor_get_rec_base_offset(v, base, base_rec, result)) {

```

```

510         *result += f->field_offset;
511         return true;
512     }
513 }
514 return false;
515 }
516
517 /* Assumes it is possible to cast record with name 'rec_from' to the record
518  * with name 'rec_to'. */
519 static int32_t aia_visitor_get_rec_cast_offset(Ast_Visitor_Aia *v,
520     Symbol_Type_Struct *from, Symbol_Type_Struct *to)
521 {
522     int32_t result;
523
524     Symbol_Type_Rec *rec_from, *rec_to;
525     rec_from = SYMBOL_TYPE_STRUCT_CONTAINER(from, Symbol_Type_Rec);
526     rec_to = SYMBOL_TYPE_STRUCT_CONTAINER(to, Symbol_Type_Rec);
527
528     if (from->methods->same_type(from, to)) {
529         result = 0;
530         goto out;
531     }
532
533     Vit_Record *r = aia_visitor_get_record(v, rec_from->unique_name);
534     assert(r);
535     if (aia_visitor_get_rec_base_offset(v, r, to, &result))
536         goto out;
537     r = aia_visitor_get_record(v, rec_to->unique_name);
538     assert(r);
539     bool ret = aia_visitor_get_rec_base_offset(v, r, from, &result);
540     (void)ret;
541     assert(ret);
542     result = -result;
543
544 out:
545     return result;
546 }
547
548 static int32_t aia_visitor_get_rec_field_offset(Ast_Visitor_Aia *v,
549     Symbol_Type_Rec *rec, String field_name)
550 {
551     if (!symbol_table_node_get(rec->rec_sym_node, field_name,
552         string_hash_code(field_name), SYMBOL_PROPERTY_VAR))
553         return -1;
554
555     Vit_Record *r = aia_visitor_get_record(v, rec->unique_name);
556     assert(r);
557     Vit_Record_Field *f = vit_record_get_field(r, field_name);
558     if (f)
559         return f->field_offset;
560
561     Vector *ext = &rec->extended_types;
562     Symbol_Type_Struct *tmp;
563     VECTOR_FOR_EACH_ENTRY(ext, tmp) {
564         Symbol_Type_Rec *tmp_r = SYMBOL_TYPE_STRUCT_CONTAINER(tmp,
565             Symbol_Type_Rec);
566         int32_t ret = aia_visitor_get_rec_field_offset(v, tmp_r, field_name);
567         if (ret > -1) {
568             f = vit_record_get_field(r, tmp_r->unique_name);
569             assert(f);
570             return ret + f->field_offset;
571         }
572     }
573
574     fatal_error(S("Unexpected query for record field '%S'. Aborting...\n"));
575 }
576
577 static inline Uns vit_record_get_record_padding(Uns current_offset,
578     Vit_Record *oth_rec)
579 {
580     assert(oth_rec->alignment);
581     if (oth_rec->alignment == BYTE_ALIGNMENT) {
582         return 0;
583     } else { // alignment == LONG_ALIGNMENT

```

```

584     if (current_offset & LONG_ALIGN_MASK)
585         return LONG_ALIGNMENT - (current_offset & LONG_ALIGN_MASK);
586     else
587         return 0;
588 }
589 }
590
591 static inline void vit_record_insert_field(Vit_Record *r, Vit_Record_Field *f,
592     Ast_Visitor_Aia *v)
593 {
594     Vit_Record *oth;
595     Uns hash = string_hash_code(f->field_name);
596     assert(!hash_map_contains(&r->field_map, f->field_name, hash));
597     hash_map_insert(&r->field_map, &f->hash_slot, hash);
598     vector_append(&r->field_vector, f);
599     switch (f->field_type) {
600     case VIT_RECORD_FIELD_BYTE:
601         f->field_offset = r->byte_size;
602         r->byte_size += BYTE_ALIGNMENT;
603         break;
604     case VIT_RECORD_FIELD_LONG:
605         if (r->byte_size & LONG_ALIGN_MASK)
606             f->field_offset =
607                 (r->byte_size & ~LONG_ALIGN_MASK) + LONG_ALIGNMENT;
608         else
609             f->field_offset = r->byte_size;
610         r->byte_size = f->field_offset + LONG_ALIGNMENT;
611         break;
612     case VIT_RECORD_FIELD_RECORD:
613         oth = aia_visitor_get_record(v, f->field_type_name);
614         assert(oth);
615         Uns padding = vit_record_get_record_padding(r->byte_size, oth);
616         f->field_offset = r->byte_size + padding;
617         r->byte_size = f->field_offset + oth->byte_size;
618         break;
619     }
620 }
621
622 static bool vit_record_comparator(String search_rec_name,
623     Hash_Map_Slot *map_slot)
624 {
625     Vit_Record *map_rec = VIT_RECORD_OF(map_slot);
626     return !string_compare(search_rec_name, map_rec->identifier);
627 }
628
629 static inline void aia_visitor_insert_record(Ast_Visitor_Aia *v, Vit_Record *r,
630     Hash_Map *map)
631 {
632     Vit_Record_Field *tmp;
633     Vit_Record *ext;
634     Uns hash = string_hash_code(r->identifier);
635     assert(!hash_map_contains(map, r->identifier, hash));
636     if (vector_is_empty(&r->field_vector)) {
637         r->alignment = BYTE_ALIGNMENT;
638     } else {
639         tmp = vector_get(&r->field_vector, 0);
640         switch (tmp->field_type) {
641         case VIT_RECORD_FIELD_BYTE:
642             r->alignment = BYTE_ALIGNMENT;
643             break;
644         case VIT_RECORD_FIELD_LONG:
645             r->alignment = LONG_ALIGNMENT;
646             break;
647         case VIT_RECORD_FIELD_RECORD:
648             ext = aia_visitor_get_record(v, tmp->field_type_name);
649             r->alignment = ext->alignment;
650             break;
651         }
652     }
653     hash_map_insert(map, &r->hash_slot, hash);
654 }
655
656 static inline Vit_Record *aia_visitor_get_record(Ast_Visitor_Aia *v,
657     String iden)

```



```

658 {
659     Uns hash = string_hash_code(iden);
660     Hash_Map_Slot *slot;
661     slot = hash_map_get(&v->record_map, iden, hash);
662     if (slot)
663         return VIT_RECORD_OF(slot);
664     return NULL;
665 }
666
667 bool aia_struct_comparator(Vit_Record *search_struct, Hash_Map_Slot *map_slot);
668
669 static void rec_generate_record(Symbol_Type_Rec *rec,
670     Ast_Visitor_Aia *v);
671
672 static void rec_generate_extended_records(Symbol_Type_Rec *rec,
673     Ast_Visitor_Aia *v)
674 {
675     Symbol_Type_Rec *ext_r;
676     Symbol_Type_Struct *ext_s;
677     Vector *extended = &rec->extended_types;
678     VECTOR_FOR_EACH_ENTRY(extended, ext_s) {
679         ext_r = SYMBOL_TYPE_STRUCT_CONTAINER(ext_s, Symbol_Type_Rec);
680         rec_generate_record(ext_r, v);
681     }
682 }
683
684 static inline bool rec_has_vmt_func(Symbol_Type_Rec *rec)
685 {
686     Const_String iden;
687     VECTOR_FOR_EACH_ENTRY(&rec->func_identifiers, iden) {
688         if (!string_is_ctor(iden))
689             return true;
690     }
691
692     Vector *ext = &rec->extended_types;
693     Symbol_Type_Struct *tmp;
694     VECTOR_FOR_EACH_ENTRY(ext, tmp) {
695         if (rec_has_vmt_func(
696             SYMBOL_TYPE_STRUCT_CONTAINER(tmp, Symbol_Type_Rec)))
697             return true;
698     }
699     return false;
700 }
701
702 static inline bool rec_has_own_vmt(Symbol_Type_Rec *rec)
703 {
704     Symbol_Type_Struct *tmp;
705     Vector *ext = &rec->extended_types;
706     VECTOR_FOR_EACH_ENTRY(ext, tmp) {
707         if (rec_has_vmt_func(
708             SYMBOL_TYPE_STRUCT_CONTAINER(tmp, Symbol_Type_Rec)))
709             return false;
710     }
711
712     Const_String iden;
713     VECTOR_FOR_EACH_ENTRY(&rec->func_identifiers, iden) {
714         if (!string_is_ctor(iden))
715             return true;
716     }
717
718     return false;
719 }
720
721 /* Called to make sure that if some extended record is defining a function then
722  * the first extended record is defining a function. */
723 static bool rec_order_extended(Symbol_Type_Rec *rec)
724 {
725     Uns size;
726     Symbol_Type_Rec *ext_r;
727     Symbol_Type_Struct *ext_s;
728     Vector *extended = &rec->extended_types;
729     size = vector_size(extended);
730     for (Uns i = 0; i < size; i++) {
731         ext_s = vector_get(extended, i);

```

```

732     ext_r = SYMBOL_TYPE_STRUCT_CONTAINER(ext_s, Symbol_Type_Rec);
733     if (rec_has_vmt_func(ext_r)) {
734         vector_swap(extended, 0, i);
735         return true;
736     }
737 }
738 return false;
739 }
740
741 static inline void rec_insert_vmt(Symbol_Type_Rec *rec, Vit_Record *r,
742     Ast_Visitor_Aia *v)
743 {
744     if (rec_has_own_vmt(rec)) {
745         Vit_Record_Field *field =
746             vit_record_field_alloc_simple(VIT_RECORD_FIELD_LONG,
747                 SYMBOL_TYPE_REC, VMT_STR);
748         vit_record_insert_field(r, field, v);
749     }
750 }
751
752 static void rec_insert_extended(Symbol_Type_Rec *rec, Vit_Record *r,
753     Ast_Visitor_Aia *v)
754 {
755     Vit_Record_Field *f;
756     Symbol_Type_Rec *ext_r;
757     Symbol_Type_Struct *ext_s;
758     Vector *extended = &rec->extended_types;
759     VECTOR_FOR_EACH_ENTRY(extended, ext_s) {
760         ext_r = SYMBOL_TYPE_STRUCT_CONTAINER(ext_s, Symbol_Type_Rec);
761         f = vit_record_field_alloc_record(ext_r->unique_name,
762             ext_r->unique_name);
763         vit_record_insert_field(r, f, v);
764         vector_append(&r->extended_sym_types, ext_s);
765     }
766 }
767
768 static void rec_insert_field(String iden, Symbol_Type_Struct *s,
769     Vit_Record *r, Ast_Visitor_Aia *v)
770 {
771     Vit_Record_Field *f;
772     Symbol_Type t = s->methods->get_type();
773     switch (t) {
774     case SYMBOL_TYPE_INT:
775     case SYMBOL_TYPE_STRING:
776     case SYMBOL_TYPE_ARY:
777     case SYMBOL_TYPE_REC:
778         f = vit_record_field_alloc_simple(VIT_RECORD_FIELD_LONG, t, iden);
779         break;
780     case SYMBOL_TYPE_BOOL:
781     case SYMBOL_TYPE_CHAR:
782         f = vit_record_field_alloc_simple(VIT_RECORD_FIELD_BYTE, t, iden);
783         break;
784     default:
785         aia_visitor_unexpected_symbol();
786     }
787     vit_record_insert_field(r, f, v);
788 }
789
790 static void rec_insert_fields(Symbol_Type_Rec *rec, Vit_Record *r,
791     Ast_Visitor_Aia *v)
792 {
793     Vector *ftype_vec = &rec->var_types;
794     Vector *fname_vec = &rec->var_identifiers;
795     Uns size = vector_size(fname_vec);
796     for (Uns i = 0; i < size; i++)
797         rec_insert_field(vector_get(fname_vec, i),
798             vector_get(ftype_vec, i), r, v);
799 }
800
801 static inline String get_func_long_name(Symbol_Table_Node *n, String func_iden)
802 {
803     Symbol *sym = symbol_table_node_lookup(n, func_iden, SYMBOL_PROPERTY_FUNC);
804     assert(sym);
805     return sym->unique_name;

```

```

806 }
807
808 static Vit_Record_Func_Entry *vit_record_func_entry_alloc(
809     Symbol_Type_Rec *rec, String func_name)
810 {
811     Vit_Record_Func_Entry *e = ALLOC_NEW(Vit_Record_Func_Entry);
812     e->func_name = func_name;
813     e->long_func_name = get_func_long_name(rec->rec_sym_node, func_name);
814     e->is_inserted = false;
815     e->owner = rec;
816     return e;
817 }
818
819 static inline void vit_record_func_entry_destroy(Vit_Record_Func_Entry *e)
820 {
821     free_mem(e);
822 }
823
824 static bool vit_record_func_entry_comparator(String search_func,
825     Hash_Map_Slot *map_slot)
826 {
827     Vit_Record_Func_Entry *fentry = VIT_RECORD_FUNC_ENTRY_OF(map_slot);
828     return !string_compare(fentry->func_name, search_func);
829 }
830
831 static inline String rec_get_func_iden(Symbol_Table_Node *orig_node,
832     String first_iden)
833 {
834     DLOG("\nAIA get func symbol: %S from node %p\n", first_iden, orig_node);
835     DEBUG(
836         Hash_Map_Slot *slot;
837         HASH_MAP_FOR_EACH(&orig_node->symbol_maps[SYMBOL_PROPERTY_FUNC],
838             slot) {
839             Symbol *func_sym = SYMBOL_OF_SLOT(slot);
840             DLOG("existing sym: %S\n", func_sym->identifier);
841         }
842     );
843     Symbol *func_sym = symbol_table_node_get(
844         orig_node,
845         first_iden,
846         string_hash_code(first_iden),
847         SYMBOL_PROPERTY_FUNC);
848     assert(func_sym);
849     return func_sym->identifier;
850 }
851
852 static void rec_insert_missing_vmt_funcs(Vit_Vmt *vmt,
853     Symbol_Table_Node *orig_node,
854     Hash_Map *func_map,
855     Symbol_Type_Rec *rec,
856     Ast_Visitor_Aia *v,
857     Uns byte_offset,
858     bool is_first_extended)
859 {
860     DLOG("insert missing vmts in %S\n", rec->rec_name);
861     Hash_Map_Slot *slot;
862     Vit_Record_Func_Entry *e;
863     String func_iden;
864     Vector *funcs = &rec->func_identifiers;
865     VECTOR_FOR_EACH_ENTRY(funcs, func_iden) {
866         if (string_is_ctor(func_iden))
867             continue;
868         func_iden = rec_get_func_iden(orig_node, func_iden);
869         slot = hash_map_get(func_map, func_iden, string_hash_code(func_iden));
870         assert(slot);
871         e = VIT_RECORD_FUNC_ENTRY_OF(slot);
872         if (is_first_extended) {
873             if (e->is_inserted)
874                 continue;
875             vit_vmt_append_func(vmt, e->long_func_name, e->func_name,
876                 byte_offset);
877             e->is_inserted = true;
878         }
879     }

```

```

880     } else if (e->owner != rec) {
881         Vit_Vmt_Trampoline *tramp = vit_vmt_trampoline_alloc(
882             e->long_func_name, rec->unique_name, byte_offset);
883         String tname = aia_visitor_insert_trampoline(v, tramp);
884         vit_vmt_append_func(vmt, tname, e->func_name, byte_offset);
885     } else {
886         vit_vmt_append_func(vmt, e->long_func_name, e->func_name,
887             byte_offset);
888         e->is_inserted = true;
889     }
890 }
891 }
892
893 static void rec_append_vmt(Vit_Vmt *vmt,
894     Symbol_Table_Node *orig_node,
895     Hash_Map *func_map,
896     Symbol_Type_Rec *rec,
897     Ast_Visitor_Aia *v,
898     Uns curr_offset,
899     bool is_first_extended)
900 {
901     Vector *vec;
902     String func_iden;
903     Uns hash;
904     Hash_Map_Slot *slot;
905
906     vec = &rec->func_identifiers;
907     VECTOR_FOR_EACH_ENTRY(vec, func_iden) {
908         if (string_is_ctor(func_iden))
909             continue;
910         func_iden = rec_get_func_iden(orig_node, func_iden);
911         hash = string_hash_code(func_iden);
912         if (hash_map_contains(func_map, func_iden, hash))
913             continue;
914         Vit_Record_Func_Entry *e = vit_record_func_entry_alloc(rec,
915             func_iden);
916         hash_map_insert(func_map, &e->hash_slot, hash);
917     }
918
919     vec = &rec->extended_types;
920     Uns size = vector_size(vec);
921     if (size) {
922         Symbol_Type_Rec *tmp = vector_get(vec, 0);
923
924         Vit_Record *tmp_vit = aia_visitor_get_record(v, tmp->unique_name);
925         curr_offset += vit_record_get_record_padding(curr_offset, tmp_vit);
926
927         vit_vmt_append_rec_vmt_offset(vmt, curr_offset,
928             vmt->current_func_offset);
929         rec_append_vmt(vmt, orig_node, func_map, tmp, v,
930             curr_offset, is_first_extended);
931         rec_insert_missing_vmt_funcs(vmt, orig_node, func_map, rec, v,
932             curr_offset, is_first_extended);
933
934         curr_offset += tmp_vit->byte_size;
935         for (Uns i = 1; i < size; i++) {
936             tmp = vector_get(vec, i);
937
938             tmp_vit = aia_visitor_get_record(v, tmp->unique_name);
939             curr_offset += vit_record_get_record_padding(curr_offset, tmp_vit);
940
941             if (rec_has_vmt_func(tmp))
942                 vit_vmt_append_rec_vmt_offset(vmt, curr_offset,
943                     vmt->current_func_offset);
944             rec_append_vmt(vmt, orig_node, func_map, tmp, v,
945                 curr_offset, false);
946             curr_offset += tmp_vit->byte_size;
947         }
948     } else {
949         Vit_Record *tmp_vit = aia_visitor_get_record(v, rec->unique_name);
950         curr_offset += vit_record_get_record_padding(curr_offset, tmp_vit);
951
952         if (rec_has_vmt_func(rec))
953             vit_vmt_append_rec_vmt_offset(vmt, curr_offset,

```

```

954         vmt->current_func_offset);
955     rec_insert_missing_vmt_funcs(vmt, orig_node, func_map, rec, v,
956         curr_offset, is_first_extended);
957 }
958
959 vec = &rec->func_identifiers;
960 VECTOR_FOR_EACH_ENTRY(vec, func_iden) {
961     if (string_is_ctor(func_iden))
962         continue;
963     func_iden = rec_get_func_iden(orig_node, func_iden);
964     hash = string_hash_code(func_iden);
965     slot = hash_map_get(func_map, func_iden, hash);
966     assert(slot);
967     Vit_Record_Func_Entry *e = VIT_RECORD_FUNC_ENTRY_OF(slot);
968     if (e->owner == rec) {
969         hash_map_remove(func_map, func_iden, hash);
970         vit_record_func_entry_destroy(e);
971     }
972 }
973 }
974
975 static void rec_generate_vmt(Symbol_Type_Rec *rec, Ast_Visitor_Aia *v,
976     Vit_Record *r)
977 {
978     if (vector_is_empty(&rec->func_types)) {
979         if (vector_is_empty(&rec->extended_types))
980             goto out;
981         Symbol_Type_Rec *tmp = SYMBOL_TYPE_STRUCT_CONTAINER(
982             vector_get(&rec->extended_types, 0), Symbol_Type_Rec);
983         if (vector_is_empty(&tmp->func_types))
984             goto out;
985     }
986
987     Vit_Vmt *vmt = vit_vmt_alloc(rec);
988     if (!vmt)
989         goto out;
990
991     HASH_MAP(func_map, (Hash_Map_Comparator) vit_record_func_entry_comparator);
992     rec_append_vmt(vmt, rec->rec_sym_node, &func_map, rec, v, 0, true);
993     Uns hash = string_hash_code(vmt->record_name);
994     assert(!hash_map_contains(&v->vmt_map, vmt->record_name, hash));
995     hash_map_insert(&v->vmt_map, &vmt->hash_slot, hash);
996
997     DEBUGT(aia-gen-full, vit_vmt_dump(vmt));
998
999     hash_map_clear(&func_map);
1000
1001     r->initializer_name = vmt->initializer.initializer_name;
1002 out:;
1003 }
1004
1005 static void rec_generate_record(Symbol_Type_Rec *rec, Ast_Visitor_Aia *v)
1006 {
1007     Vit_Record *r;
1008     UNUSED Symbol_Type_Rec *ext_r;
1009
1010     /* Needed to handle case where the SAME record is declared and imported */
1011     if ((r = aia_visitor_get_record(v, rec->unique_name)) {
1012         if (rec->rec_sym_node->parent->type == SYMBOL_TABLE_NODE_IMPORT)
1013             return;
1014         Hash_Map_Slot *slot = hash_map_get(&v->vmt_map, rec->unique_name,
1015             string_hash_code(rec->unique_name));
1016         if (slot)
1017             VIT_VMT_OF(slot)->is_imported = false;
1018
1019         return;
1020     }
1021
1022     bool insert_vmt_field;
1023     if (rec_order_extended(rec))
1024         insert_vmt_field = false;
1025     else
1026         insert_vmt_field = true;
1027

```

```

1028
1029     rec_generate_extended_records(rec, v);
1030
1031     r = vit_record_alloc(rec);
1032     if (insert_vmt_field)
1033         rec_insert_vmt(rec, r, v);
1034 #if 0
1035     if (vector_is_empty(&rec->extended_types)) {
1036         rec_insert_vmt(rec, r, v);
1037     } else {
1038         ext_r = SYMBOL_TYPE_STRUCT_CONTAINER(
1039             vector_get(&rec->extended_types, 0), Symbol_Type_Rec);
1040         if (vector_is_empty(&ext_r->func_types))
1041             rec_insert_vmt(rec, r, v);
1042     }
1043 #endif
1044
1045     rec_insert_extended(rec, r, v);
1046     rec_insert_fields(rec, r, v);
1047
1048     aia_visitor_insert_record(v, r, &v->record_map);
1049
1050     rec_generate_vmt(rec, v, r);
1051
1052     DEBUGT(aia-gen-full, vit_record_dump(r));
1053 }
1054
1055 static Symbol_Type_Struct *on_symbol_type_ary_get_struct(
1056     Symbol_Type_Struct *ary)
1057 {
1058     while (ary->methods->get_type() == SYMBOL_TYPE_ARY)
1059         ary = SYMBOL_TYPE_STRUCT_CONTAINER(ary, Symbol_Type_Ary)->ary_type;
1060     return ary;
1061 }
1062
1063 static inline void on_symrec_gen_vitrec(Symbol_Type_Struct *sym_s,
1064     Ast_Visitor_Aia *v)
1065 {
1066     sym_s = on_symbol_type_ary_get_struct(sym_s);
1067
1068     Symbol_Type_Rec *rec;
1069     if (sym_s->methods->get_type() == SYMBOL_TYPE_REC) {
1070         rec = SYMBOL_TYPE_STRUCT_CONTAINER(sym_s, Symbol_Type_Rec);
1071         rec_generate_record(rec, v);
1072     }
1073 }
1074
1075 static inline uint8_t ast_expr_type_to_aia_operand_size(Ast_Expr_Type t)
1076 {
1077     switch (t) {
1078     case AST_EXPR_TYPE_CHAR:
1079     case AST_EXPR_TYPE_BOOL:
1080         return AIA_BYTE;
1081     default:
1082         return AIA_LONG;
1083     }
1084 }
1085
1086 static inline uint8_t symbol_type_struct_to_aia_operand_size(
1087     Symbol_Type_Struct *s)
1088 {
1089     switch (s->methods->get_type()) {
1090     case SYMBOL_TYPE_CHAR:
1091     case SYMBOL_TYPE_BOOL:
1092         return AIA_BYTE;
1093     default:
1094         return AIA_LONG;
1095     }
1096 }
1097
1098 static inline Aia_Operand *get_tmp_op(Ast_Visitor_Aia *v)
1099 {
1100     return aia_operand_tmp_reg_alloc(v->aia);
1101 }

```

```

1102
1103 static void mov_instr(Ast_Visitor_Aia *v, uint16_t instr_t, Aia_Operand *dest,
1104     Aia_Operand *src, uint8_t dest_type, uint8_t src_type)
1105 {
1106     Aia_Instr *instr = __aia_lop_instr(v->aia, instr_t, dest_type, src_type,
1107         v->curr_loc);
1108     aia_instr_set_src_op(instr, 0, src);
1109     aia_instr_set_dest_op(instr, dest);
1110 }
1111
1112 static Aia_Operand *movs_tmp_instr(Ast_Visitor_Aia *v, Aia_Operand *op)
1113 {
1114     Aia_Operand *tmp_op = get_tmp_op(v);
1115     mov_instr(v, AIA_MOVS, tmp_op, op, AIA_LONG, AIA_BYTE);
1116     return tmp_op;
1117 }
1118
1119 static Aia_Operand *movz_tmp_instr(Ast_Visitor_Aia *v, Aia_Operand *op)
1120 {
1121     Aia_Operand *tmp_op = get_tmp_op(v);
1122     mov_instr(v, AIA_MOVZ, tmp_op, op, AIA_LONG, AIA_BYTE);
1123     return tmp_op;
1124 }
1125
1126 #define EXPRESSION_SETUP(v) \
1127     bool __saved_next_iden_is_op = (v)->next_variable_iden_is_operand; \
1128     (v)->next_variable_iden_is_operand = true;
1129
1130 #define EXPRESSION_END(v) \
1131     (v)->next_variable_iden_is_operand = __saved_next_iden_is_op;
1132
1133 #define EXPRESSION_ACCEPT_VISITOR(node, v) \
1134     DEBUG(v->prev_result = NULL); \
1135     DEBUG(v->aia_expr_type = AIA_EXPR_DEBUG); \
1136     (node)->accept_visitor((node), AST_VISITOR_OF(v)); \
1137     assert(v->aia_expr_type != AIA_EXPR_DEBUG); \
1138     assert(v->prev_result || v->aia_expr_type == AIA_EXPR_SHORT_CIRCUIT);
1139
1140 /* Possible operand types are int or char. */
1141 static void arith_binop(Ast_Visitor_Aia *v, Ast_Node_Binary *n,
1142     uint16_t op_type)
1143 {
1144     v->curr_loc = ast_node_get_file_location(AST_NODE_OF(n));
1145
1146     Aia_Operand *lhs_r, *rhs_r;
1147
1148     EXPRESSION_SETUP(v);
1149
1150     Ast_Expr_Type res_type = ast_expr_binary_get_expr_type(
1151         AST_EXPR_BINARY_OF(n));
1152     DEBUG(
1153         switch (res_type) {
1154             case AST_EXPR_TYPE_INT:
1155             case AST_EXPR_TYPE_CHAR:
1156                 break;
1157             default:
1158                 assert(false);
1159         }
1160     );
1161
1162     EXPRESSION_ACCEPT_VISITOR(n->lhs, v);
1163     lhs_r = v->prev_result;
1164     if (v->prev_expr_type == AST_EXPR_TYPE_CHAR)
1165         lhs_r = movs_tmp_instr(v, lhs_r);
1166     else if (v->prev_expr_type == AST_EXPR_TYPE_BOOL)
1167         lhs_r = movz_tmp_instr(v, lhs_r);
1168
1169     EXPRESSION_ACCEPT_VISITOR(n->rhs, v);
1170     rhs_r = v->prev_result;
1171     if (v->prev_expr_type == AST_EXPR_TYPE_CHAR)
1172         rhs_r = movs_tmp_instr(v, rhs_r);
1173     else if (v->prev_expr_type == AST_EXPR_TYPE_BOOL)
1174         rhs_r = movz_tmp_instr(v, rhs_r);
1175

```

```

1176     uint8_t op_sizes = ast_expr_type_to_aia_operand_size(res_type);
1177
1178     Aia_Operand *tmp_op = get_tmp_op(v);
1179     Aia_Instr *add_instr = __aia_2op_instr(v->aia, op_type,
1180         op_sizes, op_sizes, v->curr_loc);
1181     aia_instr_set_src_op(add_instr, 0, lhs_r);
1182     aia_instr_set_src_op(add_instr, 1, rhs_r);
1183     aia_instr_set_dest_op(add_instr, tmp_op);
1184
1185     v->prev_result = tmp_op;
1186     v->prev_expr_type = res_type;
1187     v->aia_expr_type = AIA_EXPR_STANDARD;
1188
1189     EXPRESSION_END(v);
1190 }
1191
1192 /* Get index of the function display with index_display_node as
1193  * body symbol table node.
1194  * func_node is the symbol table node of the function
1195  * which sets up the display. */
1196 static Int stn_display_index(Symbol_Table_Node *func_node,
1197     Symbol_Table_Node *index_display_node,
1198     Const_String *display_func_name_out)
1199 {
1200     while (func_node->type != SYMBOL_TABLE_NODE_FUNC) {
1201         assert(func_node->type == SYMBOL_TABLE_NODE_INTERMEDIATE ||
1202             func_node->type == SYMBOL_TABLE_NODE_REC);
1203         func_node = func_node->parent;
1204     }
1205
1206     Stn_Display_Preserve *p = stn_display_preserve_get(func_node);
1207
1208     Vector *v = &p->stn_displays;
1209     Int size = (Int)vector_size(v);
1210     for (Int i = 0; i < size; i++) {
1211         if (vector_get(v, i) == index_display_node) {
1212             Stn_Display_Preserve *tmp = stn_display_preserve_get(
1213                 index_display_node);
1214             assert(tmp);
1215             assert(tmp->func_name);
1216             *display_func_name_out = tmp->func_name;
1217             return i;
1218         }
1219     }
1220     assert(false);
1221     *display_func_name_out = NULL;
1222     return -1;
1223 }
1224
1225 static inline String generate_display_str(Uns display_idx)
1226 {
1227     return string_from_format(S(DISP_CSTR_PREFIX "%U"), display_idx);
1228 }
1229
1230 static inline void __type_cast_byte_to_long(Ast_Visitor_Aia *v,
1231     Aia_Operand *char_op, Ast_Expr_Type long_type, uint32_t mov_type)
1232 {
1233     assert(long_type != AST_EXPR_TYPE_CHAR && long_type != AST_EXPR_TYPE_BOOL);
1234     Aia_Operand *tmp_op = get_tmp_op(v);
1235     mov_instr(v, mov_type, tmp_op, char_op, AIA_LONG, AIA_BYTE);
1236     v->prev_result = tmp_op;
1237     v->prev_expr_type = long_type;
1238     v->aia_expr_type = AIA_EXPR_STANDARD;
1239 }
1240
1241 static inline void type_cast_char_to_long(Ast_Visitor_Aia *v,
1242     Aia_Operand *char_op, Ast_Expr_Type long_type)
1243 {
1244     __type_cast_byte_to_long(v, char_op, long_type, AIA_MOVS);
1245 }
1246
1247 static inline void type_cast_prev_char_to_long(Ast_Visitor_Aia *v,
1248     Ast_Expr_Type long_type)
1249 {

```



```

1250     type_cast_char_to_long(v, v->prev_result, long_type);
1251 }
1252
1253 static inline void type_cast_bool_to_long(Ast_Visitor_Aia *v,
1254     Aia_Operand *bool_op, Ast_Expr_Type long_type)
1255 {
1256     __type_cast_byte_to_long(v, bool_op, long_type, AIA_MOVZ);
1257 }
1258
1259 static inline void type_cast_prev_bool_to_long(Ast_Visitor_Aia *v,
1260     Ast_Expr_Type long_type)
1261 {
1262     type_cast_bool_to_long(v, v->prev_result, long_type);
1263 }
1264
1265 static inline void type_cast_prev_long_to_char(Ast_Visitor_Aia *v)
1266 {
1267     assert(v->prev_expr_type != AST_EXPR_TYPE_CHAR &&
1268         v->prev_expr_type != AST_EXPR_TYPE_BOOL);
1269
1270     Aia_Operand *tmp_op = get_tmp_op(v);
1271     mov_instr(v, AIA_MOV, tmp_op, v->prev_result,
1272         AIA_BYTE, AIA_BYTE);
1273     v->prev_result = tmp_op;
1274     v->prev_expr_type = AST_EXPR_TYPE_CHAR;
1275     v->aia_expr_type = AIA_EXPR_STANDARD;
1276 }
1277
1278 static inline void type_cast_prev_to_bool(Ast_Visitor_Aia *v,
1279     uint8_t prev_op_size)
1280 {
1281     assert(v->prev_expr_type != AST_EXPR_TYPE_BOOL);
1282
1283     Aia_Instr *cmp_in = __aia_2op_instr(v->aia, AIA_CMP, 0, prev_op_size,
1284         v->curr_loc);
1285     Aia_Operand *iconst0 = aia_operand_const_int_alloc(v->aia, 0);
1286     aia_instr_set_src_op(cmp_in, 0, v->prev_result);
1287     aia_instr_set_src_op(cmp_in, 1, iconst0);
1288
1289     Aia_Operand *tmp = get_tmp_op(v);
1290     Aia_Instr *setne_in = __aia_0op_instr(v->aia, AIA_SETNE, AIA_BYTE,
1291         v->curr_loc);
1292     aia_instr_set_dest_op(setne_in, tmp);
1293
1294     v->prev_result = tmp;
1295     v->prev_expr_type = AST_EXPR_TYPE_BOOL;
1296     v->aia_expr_type = AIA_EXPR_STANDARD;
1297 }
1298
1299 static void __if_stmt_expr(Ast_Visitor_Aia *v, uint8_t cmp_size,
1300     Aia_Operand *then_lbl, Aia_Operand *skip_lbl);
1301
1302 /* Assumes v->prev_result contains from record operand.
1303  * If opt_dest != NULL the result is moved to opt_dest operand. */
1304 static inline void type_cast_rec_rec(Ast_Visitor_Aia *v,
1305     Symbol_Type_Struct *to_s, Symbol_Type_Struct *from_s,
1306     Aia_Operand *opt_dest, bool null_check)
1307 {
1308     int32_t off = aia_visitor_get_rec_cast_offset(v, from_s, to_s);
1309     if (v->allocating_record) {
1310         v->allocated_record_offset += off;
1311         return;
1312     }
1313
1314     if (!off) {
1315         if (opt_dest)
1316             mov_instr(v, AIA_MOV, opt_dest, v->prev_result,
1317                 AIA_LONG, AIA_LONG);
1318         return;
1319     }
1320
1321     Aia_Operand *result;
1322     if (!opt_dest)
1323         result = get_tmp_op(v);

```

```

1324     else
1325         result = opt_dest;
1326
1327     if (null_check) {
1328         /* AIA block structure:
1329          *      ( )
1330          *      / \
1331          *      (n) (d)
1332          *      \ /
1333          *      (e) */
1334
1335         Aia_Operand *do_cast =
1336             aia_operand_tmp_label_alloc(v->aia, S("CAST.do"));
1337         Aia_Operand *is_null =
1338             aia_operand_tmp_label_alloc(v->aia, S("CAST.null"));
1339         Aia_Operand *end_lbl =
1340             aia_operand_tmp_label_alloc(v->aia, S("CAST.end"));
1341
1342         __if_stmt_expr(v, AIA_LONG, do_cast, is_null);
1343
1344         __aia_block(v->aia);
1345         __aia_insert_jump_label_instr(v->aia,
1346             aia_operand_label_get_name(do_cast));
1347
1348         Aia_Instr *add_in = __aia_2op_instr(v->aia, AIA_ADD,
1349             AIA_LONG, AIA_LONG, v->curr_loc);
1350         aia_instr_set_dest_op(add_in, result);
1351         aia_instr_set_src_op(add_in, 0, v->prev_result);
1352         aia_instr_set_src_op(add_in, 1, aia_operand_const_int_alloc(v->aia, off));
1353
1354         Aia_Instr *jmp = __aia_lop_instr(v->aia, __AIA_JMP, -1, AIA_LONG,
1355             v->curr_loc);
1356         aia_instr_set_src_op(jmp, 0, end_lbl);
1357
1358         __aia_block(v->aia);
1359         __aia_insert_jump_label_instr(v->aia,
1360             aia_operand_label_get_name(is_null));
1361
1362         Aia_Operand *iconst0 = aia_operand_const_int_alloc(v->aia, 0);
1363         mov_instr(v, AIA_MOV, result, iconst0, AIA_LONG, AIA_LONG);
1364
1365         jmp = __aia_lop_instr(v->aia, __AIA_JMP, -1, AIA_LONG, v->curr_loc);
1366         aia_instr_set_src_op(jmp, 0, end_lbl);
1367
1368         __aia_block(v->aia);
1369         __aia_insert_jump_label_instr(v->aia,
1370             aia_operand_label_get_name(end_lbl));
1371     }
1372 } else {
1373     Aia_Instr *add_in = __aia_2op_instr(v->aia, AIA_ADD,
1374         AIA_LONG, AIA_LONG, v->curr_loc);
1375
1376     aia_instr_set_dest_op(add_in, result);
1377     aia_instr_set_src_op(add_in, 0, v->prev_result);
1378     aia_instr_set_src_op(add_in, 1,
1379         aia_operand_const_int_alloc(v->aia, off));
1380 }
1381
1382 v->prev_result = result;
1383 }
1384
1385 static inline void cmp_to_bool_result(Ast_Visitor_Aia *v, uint8_t src_op_size,
1386     Aia_Operand *lhs, Aia_Operand *rhs, Aia_Operand *result_op,
1387     uint32_t set_instr)
1388 {
1389     DEBUG(
1390         switch (set_instr) {
1391             case AIA_SETE:
1392             case AIA_SETNE:
1393             case AIA_SETL:
1394             case AIA_SETG:
1395             case AIA_SETLE:
1396             case AIA_SETGE:
1397                 break;

```

```

1398         default:
1399             fatal_error(S("Unexpectd cmp_to_bool set instruction\n"));
1400     };
1401 };
1402 Aia_Instr *cmp_in = __aia_2op_instr(v->aia, AIA_CMP, 0, src_op_size,
1403     v->curr_loc);
1404 aia_instr_set_src_op(cmp_in, 0, lhs);
1405 aia_instr_set_src_op(cmp_in, 1, rhs);
1406
1407 Aia_Instr *set_in = __aia_0op_instr(v->aia, set_instr, AIA_BYTE,
1408     v->curr_loc);
1409 aia_instr_set_dest_op(set_in, result_op);
1410 }
1411
1412 static inline void mov_prev_to_bool(Ast_Visitor_Aia *v,
1413     uint8_t prev_op_size, Aia_Operand *result_op)
1414 {
1415     Aia_Operand *iconst0 = aia_operand_const_int_alloc(v->aia, 0);
1416     cmp_to_bool_result(v, prev_op_size, v->prev_result, iconst0,
1417         result_op, AIA_SETNE);
1418 }
1419
1420 /* Default AIA block structure:
1421 *      (p)
1422 *      / \
1423 *      | (c)
1424 *      | / \
1425 *      (t) (f)
1426 *      \ /
1427 *      (e) */
1428 ASTVF_BEGIN(AST_EXPR_LOR, Ast_Visitor_Aia, v, Ast_Node_Binary, n)
1429     uint8_t src_size;
1430     Aia_Instr *test_instr;
1431     Aia_Instr *jmp_instr;
1432
1433     v->curr_loc = ast_node_get_file_location(AST_NODE_OF(n));
1434
1435     EXPRESSION_SETUP(v);
1436
1437     Aia_Operand *lort;
1438     Aia_Operand *lorf;
1439
1440     Aia_Operand *lor_rhs = aia_operand_tmp_label_alloc(v->aia, S("LOR.rhs"));
1441
1442     Aia_Operand *prev_true = v->bool_true_label;
1443     Aia_Operand *prev_false = v->bool_false_label;
1444
1445     if (v->bool_true_label) {
1446         assert(v->bool_false_label);
1447         lort = v->bool_true_label;
1448         lorf = v->bool_false_label;
1449     } else {
1450         lort = aia_operand_tmp_label_alloc(v->aia, S("LOR.true"));
1451         lorf = aia_operand_tmp_label_alloc(v->aia, S("LOR.false"));
1452     }
1453
1454     v->bool_true_label = lort;
1455
1456     Aia_Operand *int_const0 = NULL;
1457
1458     v->bool_false_label = lor_rhs;
1459     EXPRESSION_ACCEPT_VISITOR(n->lhs, v);
1460     if (v->aia_expr_type == AIA_EXPR_STANDARD) {
1461         int_const0 = aia_operand_const_int_alloc(v->aia, 0);
1462         src_size = ast_expr_type_to_aia_operand_size(v->prev_expr_type);
1463
1464         test_instr = __aia_2op_instr(v->aia, AIA_CMP, 0, src_size, v->curr_loc);
1465         aia_instr_set_src_op(test_instr, 0, v->prev_result);
1466         aia_instr_set_src_op(test_instr, 1, int_const0);
1467         jmp_instr = __aia_2op_instr(v->aia, __AIA_JNE, -1, AIA_LONG,
1468             v->curr_loc);
1469         aia_instr_set_src_op(jmp_instr, 0, lort);
1470         aia_instr_set_src_op(jmp_instr, 1, lor_rhs);
1471

```

```

1472     }
1473
1474     __aia_block(v->aia);
1475     __aia_insert_jump_label_instr(v->aia,
1476         aia_operand_label_get_name(lor_rhs));
1477
1478     v->bool_false_label = lorf;
1479     EXPRESSION_ACCEPT_VISITOR(n->rhs, v);
1480     if (v->aia_expr_type == AIA_EXPR_STANDARD) {
1481         if (!int_const0)
1482             int_const0 = aia_operand_const_int_alloc(v->aia, 0);
1483         src_size = ast_expr_type_to_aia_operand_size(v->prev_expr_type);
1484
1485         test_instr = __aia_2op_instr(v->aia, AIA_CMP, 0, src_size, v->curr_loc);
1486         aia_instr_set_src_op(test_instr, 0, v->prev_result);
1487         aia_instr_set_src_op(test_instr, 1, int_const0);
1488
1489         jmp_instr = __aia_2op_instr(v->aia, __AIA_JE, -1,
1490             AIA_LONG, v->curr_loc);
1491         aia_instr_set_src_op(jmp_instr, 0, lorf);
1492         aia_instr_set_src_op(jmp_instr, 1, lort);
1493     }
1494
1495     Aia_Operand *result_op = NULL;
1496     if (!prev_true) {
1497         if (!int_const0)
1498             int_const0 = aia_operand_const_int_alloc(v->aia, 0);
1499         Aia_Operand *int_const1 = aia_operand_const_int_alloc(v->aia, 1);
1500         Aia_Operand *lore = aia_operand_tmp_label_alloc(v->aia, S("LOR.end"));
1501
1502         result_op = get_tmp_op(v);
1503
1504         __aia_block(v->aia);
1505         __aia_insert_jump_label_instr(v->aia,
1506             aia_operand_label_get_name(lort));
1507
1508         Aia_Instr *mov_instr = __aia_1op_instr(v->aia, AIA_MOV,
1509             AIA_BYTE, AIA_BYTE, v->curr_loc);
1510         aia_instr_set_src_op(mov_instr, 0, int_const1);
1511         aia_instr_set_dest_op(mov_instr, result_op);
1512         jmp_instr = __aia_1op_instr(v->aia, __AIA_JMP, -1, AIA_LONG,
1513             v->curr_loc);
1514         aia_instr_set_src_op(jmp_instr, 0, lore);
1515
1516         __aia_block(v->aia);
1517         __aia_insert_jump_label_instr(v->aia,
1518             aia_operand_label_get_name(lorf));
1519
1520         mov_instr = __aia_1op_instr(v->aia, AIA_MOV, AIA_BYTE, AIA_BYTE,
1521             v->curr_loc);
1522         aia_instr_set_src_op(mov_instr, 0, int_const0);
1523         aia_instr_set_dest_op(mov_instr, result_op);
1524         jmp_instr = __aia_1op_instr(v->aia, __AIA_JMP, -1, AIA_LONG,
1525             v->curr_loc);
1526         aia_instr_set_src_op(jmp_instr, 0, lore);
1527
1528         __aia_block(v->aia);
1529         __aia_insert_jump_label_instr(v->aia,
1530             aia_operand_label_get_name(lore));
1531     }
1532
1533     v->bool_true_label = prev_true;
1534     v->bool_false_label = prev_false;
1535
1536     v->prev_expr_type = AST_EXPR_TYPE_BOOL;
1537     v->aia_expr_type = AIA_EXPR_SHORT_CIRCUIT;
1538     v->prev_result = result_op;
1539
1540     EXPRESSION_END(v);
1541     ASTVF_END
1542
1543     /* Default AIA block structure:
1544     *     (p)
1545     *     / \

```

```

1546 * | (c)
1547 * | / \
1548 * (f) (t)
1549 * \ /
1550 * (e) */
1551 ASTVF_BEGIN(AST_EXPR_LAND, Ast_Visitor_Aia, v, Ast_Node_Binary, n)
1552     uint8_t src_size;
1553     Aia_Instr *test_instr;
1554     Aia_Instr *jmp_instr;
1555
1556     v->curr_loc = ast_node_get_file_location(AST_NODE_OF(n));
1557
1558     EXPRESSION_SETUP(v);
1559
1560     Aia_Operand *landf;
1561     Aia_Operand *landt;
1562
1563     Aia_Operand *land_rhs = aia_operand_tmp_label_alloc(v->aia, S("LAND.rhs"));
1564
1565     Aia_Operand *prev_true = v->bool_true_label;
1566     Aia_Operand *prev_false = v->bool_false_label;
1567
1568     if (v->bool_true_label) {
1569         assert(v->bool_false_label);
1570         landt = v->bool_true_label;
1571         landf = v->bool_false_label;
1572     } else {
1573         landt = aia_operand_tmp_label_alloc(v->aia, S("LAND.true"));
1574         landf = aia_operand_tmp_label_alloc(v->aia, S("LAND.false"));
1575     }
1576     v->bool_false_label = landf;
1577
1578     Aia_Operand *int_const0 = NULL;
1579
1580     v->bool_true_label = land_rhs;
1581     EXPRESSION_ACCEPT_VISITOR(n->lhs, v);
1582     if (v->aia_expr_type == AIA_EXPR_STANDARD) {
1583         int_const0 = aia_operand_const_int_alloc(v->aia, 0);
1584         src_size = ast_expr_type_to_aia_operand_size(v->prev_expr_type);
1585
1586         test_instr = __aia_2op_instr(v->aia, AIA_CMP, 0, src_size,
1587             v->curr_loc);
1588         aia_instr_set_src_op(test_instr, 0, v->prev_result);
1589         aia_instr_set_src_op(test_instr, 1, int_const0);
1590         jmp_instr = __aia_2op_instr(v->aia, __AIA_JE, -1, AIA_LONG,
1591             v->curr_loc);
1592         aia_instr_set_src_op(jmp_instr, 0, landf);
1593         aia_instr_set_src_op(jmp_instr, 1, land_rhs);
1594     }
1595
1596     __aia_block(v->aia);
1597     __aia_insert_jump_label_instr(v->aia,
1598         aia_operand_label_get_name(land_rhs));
1599
1600     v->bool_true_label = landt;
1601     EXPRESSION_ACCEPT_VISITOR(n->rhs, v);
1602     if (v->aia_expr_type == AIA_EXPR_STANDARD) {
1603         if (!int_const0)
1604             int_const0 = aia_operand_const_int_alloc(v->aia, 0);
1605         src_size = ast_expr_type_to_aia_operand_size(v->prev_expr_type);
1606
1607         test_instr = __aia_2op_instr(v->aia, AIA_CMP, 0, src_size,
1608             v->curr_loc);
1609         aia_instr_set_src_op(test_instr, 0, v->prev_result);
1610         aia_instr_set_src_op(test_instr, 1, int_const0);
1611         jmp_instr = __aia_2op_instr(v->aia, __AIA_JNE, -1, AIA_LONG,
1612             v->curr_loc);
1613         aia_instr_set_src_op(jmp_instr, 0, landt);
1614         aia_instr_set_src_op(jmp_instr, 1, landf);
1615     }
1616
1617     Aia_Operand *result_op = NULL;
1618
1619     if (!prev_true) {

```

```

1620     if (!int_const0)
1621         int_const0 = aia_operand_const_int_alloc(v->aia, 0);
1622     Aia_Operand *int_const1 = aia_operand_const_int_alloc(v->aia, 1);
1623     Aia_Operand *lande = aia_operand_tmp_label_alloc(v->aia, S("LOR.end"));
1624
1625     result_op = get_tmp_op(v);
1626
1627     __aia_block(v->aia);
1628     __aia_insert_jump_label_instr(v->aia,
1629         aia_operand_label_get_name(landf));
1630
1631     Aia_Instr *mov_instr = __aia_lop_instr(v->aia, AIA_MOV,
1632         AIA_BYTE, AIA_BYTE, v->curr_loc);
1633     aia_instr_set_src_op(mov_instr, 0, int_const0);
1634     aia_instr_set_dest_op(mov_instr, result_op);
1635     jmp_instr = __aia_lop_instr(v->aia, __AIA_JMP, -1, AIA_LONG,
1636         v->curr_loc);
1637     aia_instr_set_src_op(jmp_instr, 0, lande);
1638
1639     __aia_block(v->aia);
1640     __aia_insert_jump_label_instr(v->aia,
1641         aia_operand_label_get_name(landt));
1642
1643     mov_instr = __aia_lop_instr(v->aia, AIA_MOV, AIA_BYTE, AIA_BYTE,
1644         v->curr_loc);
1645     aia_instr_set_src_op(mov_instr, 0, int_const1);
1646     aia_instr_set_dest_op(mov_instr, result_op);
1647     jmp_instr = __aia_lop_instr(v->aia, __AIA_JMP, -1, AIA_LONG,
1648         v->curr_loc);
1649     aia_instr_set_src_op(jmp_instr, 0, lande);
1650
1651     __aia_block(v->aia);
1652     __aia_insert_jump_label_instr(v->aia,
1653         aia_operand_label_get_name(lande));
1654 }
1655
1656 v->bool_true_label = prev_true;
1657 v->bool_false_label = prev_false;
1658
1659 v->prev_expr_type = AST_EXPR_TYPE_BOOL;
1660 v->aia_expr_type = AIA_EXPR_SHORT_CIRCUIT;
1661 v->prev_result = result_op;
1662
1663 EXPRESSION_END(v);
1664 ASTVF_END
1665
1666 static inline void comparison_binop(Ast_Visitor_Aia *v, Ast_Node_Binary *n,
1667     uint32_t set_instr)
1668 {
1669     v->curr_loc = ast_node_get_file_location(AST_NODE_OF(n));
1670
1671     EXPRESSION_SETUP(v);
1672     Ast_Expr_Type lhs_t, rhs_t;
1673     Aia_Operand *lhs_r, *rhs_r;
1674
1675     EXPRESSION_ACCEPT_VISITOR(n->lhs, v);
1676     lhs_t = v->prev_expr_type;
1677     lhs_r = v->prev_result;
1678
1679     EXPRESSION_ACCEPT_VISITOR(n->rhs, v);
1680     rhs_t = v->prev_expr_type;
1681     rhs_r = v->prev_result;
1682
1683     Aia_Operand *tmp = get_tmp_op(v);
1684     switch (lhs_t) {
1685     case AST_EXPR_TYPE_CHAR:
1686         switch (rhs_t) {
1687         case AST_EXPR_TYPE_BOOL:
1688         case AST_EXPR_TYPE_CHAR:
1689             cmp_to_bool_result(v, AIA_BYTE, lhs_r, rhs_r, tmp, set_instr);
1690             break;
1691         default:
1692             type_cast_char_to_long(v, lhs_r, rhs_t);
1693             cmp_to_bool_result(v, AIA_LONG, v->prev_result,

```

```

1694         rhs_r, tmp, set_instr);
1695     break;
1696 }
1697 break;
1698
1699 case AST_EXPR_TYPE_BOOL:
1700     switch (rhs_t) {
1701     case AST_EXPR_TYPE_CHAR:
1702         /* Fall through. */
1703     case AST_EXPR_TYPE_BOOL:
1704         cmp_to_bool_result(v, AIA_BYTE, lhs_r, rhs_r, tmp, set_instr);
1705         break;
1706     default:
1707         type_cast_bool_to_long(v, lhs_r, rhs_t);
1708         cmp_to_bool_result(v, AIA_LONG, v->prev_result,
1709             rhs_r, tmp, set_instr);
1710         break;
1711     }
1712     break;
1713
1714 default:
1715     switch (rhs_t) {
1716     case AST_EXPR_TYPE_BOOL:
1717         type_cast_bool_to_long(v, v->prev_result, lhs_t);
1718         cmp_to_bool_result(v, AIA_LONG, lhs_r,
1719             v->prev_result, tmp, set_instr);
1720         break;
1721     case AST_EXPR_TYPE_CHAR:
1722         type_cast_char_to_long(v, rhs_r, lhs_t);
1723         cmp_to_bool_result(v, AIA_LONG, lhs_r,
1724             v->prev_result, tmp, set_instr);
1725         break;
1726     default:
1727         cmp_to_bool_result(v, AIA_LONG, lhs_r, rhs_r, tmp, set_instr);
1728         break;
1729     }
1730     break;
1731 }
1732 v->prev_result = tmp;
1733
1734 EXPRESSION_END(v);
1735 v->prev_expr_type = ast_expr_binary_get_expr_type(AST_EXPR_BINARY_OF(n));
1736 v->aia_expr_type = AIA_EXPR_STANDARD;
1737 }
1738
1739 ASTVF_BEGIN(AST_EXPR_EQ, Ast_Visitor_Aia, v, Ast_Node_Binary, n)
1740     comparison_binop(v, n, AIA_SETE);
1741 ASTVF_END
1742
1743 ASTVF_BEGIN(AST_EXPR_NEQ, Ast_Visitor_Aia, v, Ast_Node_Binary, n)
1744     comparison_binop(v, n, AIA_SETNE);
1745 ASTVF_END
1746
1747 ASTVF_BEGIN(AST_EXPR_GT, Ast_Visitor_Aia, v, Ast_Node_Binary, n)
1748     comparison_binop(v, n, AIA_SETG);
1749 ASTVF_END
1750
1751 ASTVF_BEGIN(AST_EXPR_LT, Ast_Visitor_Aia, v, Ast_Node_Binary, n)
1752     comparison_binop(v, n, AIA_SETL);
1753 ASTVF_END
1754
1755 ASTVF_BEGIN(AST_EXPR_GTEQ, Ast_Visitor_Aia, v, Ast_Node_Binary, n)
1756     comparison_binop(v, n, AIA_SETGE);
1757 ASTVF_END
1758
1759 ASTVF_BEGIN(AST_EXPR_LTEQ, Ast_Visitor_Aia, v, Ast_Node_Binary, n)
1760     comparison_binop(v, n, AIA_SETLE);
1761 ASTVF_END
1762
1763 ASTVF_BEGIN(AST_EXPR_PLUS, Ast_Visitor_Aia, v, Ast_Node_Binary, n)
1764     arith_binop(v, n, AIA_ADD);
1765 ASTVF_END
1766
1767

```

```

1768 ASTVF_BEGIN(AST_EXPR_MINUS, Ast_Visitor_Aia, v, Ast_Node_Binary, n)
1769     arith_binop(v, n, AIA_SUB);
1770 ASTVF_END
1771
1772 ASTVF_BEGIN(AST_EXPR_MUL, Ast_Visitor_Aia, v, Ast_Node_Binary, n)
1773     arith_binop(v, n, AIA_IMUL);
1774 ASTVF_END
1775
1776 ASTVF_BEGIN(AST_EXPR_DIV, Ast_Visitor_Aia, v, Ast_Node_Binary, n)
1777     arith_binop(v, n, AIA_IDIV);
1778 ASTVF_END
1779
1780 static NORETURN void type_cast_unexpected()
1781 {
1782     fatal_error(S("Unexpected type cast. Aborting...\n"));
1783 }
1784
1785 ASTVF_BEGIN(AST_EXPR_CAST, Ast_Visitor_Aia, v, Ast_Node_Binary, n)
1786     v->curr_loc = ast_node_get_file_location(AST_NODE_OF(n));
1787
1788     Ast_Expr_Type lhs_t;
1789     Symbol_Type_Struct *lhs_s, *rhs_s;
1790
1791     Symbol *prev_curr_sym = v->curr_symbol;
1792     Symbol *lhs_sym = symbol_table_get_from_location(
1793         ast_node_get_symbol_table_node(AST_NODE_OF(n)),
1794         ast_node_get_file_location(AST_NODE_OF(n)));
1795     v->curr_symbol = lhs_sym;
1796
1797     Symbol_Property saved_property = v->next_property;
1798     v->next_property = SYMBOL_PROPERTY_TYPE_DEF;
1799     n->lhs->accept_visitor(n->lhs, AST_VISITOR_OF(v));
1800     v->next_property = saved_property;
1801
1802     lhs_s = lhs_sym->resolved_type;
1803     lhs_t = symbol_type_to_expr_type(lhs_s);
1804
1805     on_symrec_gen_vitrec(lhs_s, v);
1806
1807     EXPRESSION_SETUP(v);
1808     EXPRESSION_ACCEPT_VISITOR(n->rhs, v);
1809     rhs_s = v->prev_sym_type;
1810
1811     DEBUGT(def,
1812         if (v->allocating_record) {
1813             assert(v->prev_expr_type == AST_EXPR_TYPE_REC);
1814             assert(lhs_t == AST_EXPR_TYPE_REC);
1815         }
1816     );
1817
1818     switch (lhs_t) {
1819     case AST_EXPR_TYPE_INT:
1820         switch (v->prev_expr_type) {
1821         case AST_EXPR_TYPE_INT:
1822             break;
1823         case AST_EXPR_TYPE_CHAR:
1824             type_cast_prev_char_to_long(v, AST_EXPR_TYPE_INT);
1825             break;
1826         case AST_EXPR_TYPE_BOOL:
1827             type_cast_prev_bool_to_long(v, AST_EXPR_TYPE_INT);
1828             break;
1829         default:
1830             type_cast_unexpected();
1831         }
1832         break;
1833
1834     case AST_EXPR_TYPE_CHAR:
1835         switch (v->prev_expr_type) {
1836         case AST_EXPR_TYPE_INT:
1837             type_cast_prev_long_to_char(v);
1838             break;
1839         case AST_EXPR_TYPE_CHAR:
1840             break;
1841         case AST_EXPR_TYPE_BOOL:

```



```

1842         break;
1843     default:
1844         type_cast_unexpected();
1845     }
1846     break;
1847
1848     case AST_EXPR_TYPE_BOOL:
1849         switch (v->prev_expr_type) {
1850             case AST_EXPR_TYPE_CHAR:
1851                 type_cast_prev_to_bool(v, AIA_BYTE);
1852                 break;
1853             case AST_EXPR_TYPE_BOOL:
1854                 break;
1855             default:
1856                 type_cast_prev_to_bool(v, AIA_LONG);
1857                 break;
1858         }
1859         break;
1860
1861     case AST_EXPR_TYPE_STRING:
1862         switch (v->prev_expr_type) {
1863             case AST_EXPR_TYPE_NULL:
1864                 break;
1865             case AST_EXPR_TYPE_STRING:
1866                 break;
1867             case AST_EXPR_TYPE_ARY:
1868                 break;
1869             default:
1870                 type_cast_unexpected();
1871                 break;
1872         }
1873         break;
1874
1875     case AST_EXPR_TYPE_REC:
1876         switch (v->prev_expr_type) {
1877             case AST_EXPR_TYPE_NULL:
1878                 break;
1879             case AST_EXPR_TYPE_REC:
1880                 type_cast_rec_rec(v, lhs_s, rhs_s, NULL, true);
1881                 break;
1882             default:
1883                 type_cast_unexpected();
1884                 break;
1885         }
1886         break;
1887
1888     case AST_EXPR_TYPE_ARY:
1889         switch (v->prev_expr_type) {
1890             case AST_EXPR_TYPE_NULL:
1891                 break;
1892             case AST_EXPR_TYPE_ARY:
1893                 break;
1894             default:
1895                 type_cast_unexpected();
1896                 break;
1897         }
1898         break;
1899
1900     default:
1901         type_cast_unexpected();
1902     }
1903
1904     EXPRESSION_END(v);
1905     v->prev_expr_type = ast_expr_binary_get_expr_type(AST_EXPR_BINARY_OF(n));
1906     v->aia_expr_type = AIA_EXPR_STANDARD;
1907     v->prev_sym_type = lhs_s;
1908     v->curr_symbol = prev_curr_sym;
1909     ASTVF_END
1910
1911     ASTVF_BEGIN(AST_EXPR_LNOT, Ast_Visitor_Aia, v, Ast_Node_Unary, n)
1912         v->curr_loc = ast_node_get_file_location(AST_NODE_OF(n));
1913
1914         n->expr->accept_visitor(n->expr, AST_VISITOR_OF(v));
1915

```

```

1916     uint8_t src_size = ast_expr_type_to_aia_operand_size(v->prev_expr_type);
1917
1918     Aia_Instr *cmp_in = __aia_2op_instr(v->aia, AIA_CMP, 0, src_size,
1919         v->curr_loc);
1920     Aia_Operand *iconst0 = aia_operand_const_int_alloc(v->aia, 0);
1921     aia_instr_set_src_op(cmp_in, 0, v->prev_result);
1922     aia_instr_set_src_op(cmp_in, 1, iconst0);
1923
1924     Aia_Operand *tmp = get_tmp_op(v);
1925     Aia_Instr *sete_in = __aia_0op_instr(v->aia, AIA_SETE, AIA_BYTE,
1926         v->curr_loc);
1927     aia_instr_set_dest_op(sete_in, tmp);
1928
1929     v->prev_result = tmp;
1930     v->prev_expr_type = AST_EXPR_TYPE_BOOL;
1931     v->aia_expr_type = AIA_EXPR_STANDARD;
1932 ASTVF_END
1933
1934 static void cmp0_skip_on_false(Ast_Visitor_Aia *v, uint8_t cmp_size,
1935     Aia_Operand *then_lbl, Aia_Operand *skip_lbl, uint16_t jmp_instr_type)
1936 {
1937     Aia_Operand *iconst0 = aia_operand_const_int_alloc(v->aia, 0);
1938     Aia_Instr *cmp = __aia_2op_instr(v->aia, AIA_CMP, 0, cmp_size,
1939         v->curr_loc);
1940     aia_instr_set_src_op(cmp, 0, v->prev_result);
1941     aia_instr_set_src_op(cmp, 1, iconst0);
1942     Aia_Instr *jmp = __aia_2op_instr(v->aia, jmp_instr_type, -1, AIA_LONG,
1943         v->curr_loc);
1944     aia_instr_set_src_op(jmp, 0, skip_lbl);
1945     aia_instr_set_src_op(jmp, 1, then_lbl);
1946 }
1947
1948 static void long_integer_abs(Ast_Visitor_Aia *v)
1949 {
1950     Aia_Operand *result = get_tmp_op(v);
1951
1952     /* AIA block structure:
1953     *      ( )
1954     *      / \
1955     *      (n) (p)
1956     *      \ /
1957     *      (e) */
1958
1959     Aia_Operand *do_neg =
1960         aia_operand_tmp_label_alloc(v->aia, S("ABS.neg"));
1961     Aia_Operand *skip_neg =
1962         aia_operand_tmp_label_alloc(v->aia, S("ABS.pos"));
1963     Aia_Operand *end_lbl =
1964         aia_operand_tmp_label_alloc(v->aia, S("ABS.end"));
1965
1966     cmp0_skip_on_false(v, AIA_LONG, do_neg, skip_neg, __AIA_JGE);
1967
1968     __aia_block(v->aia);
1969     __aia_insert_jump_label_instr(v->aia,
1970         aia_operand_label_get_name(do_neg));
1971
1972     Aia_Instr *neg_in = __aia_1op_instr(v->aia, AIA_NEG,
1973         AIA_LONG, AIA_LONG, v->curr_loc);
1974     aia_instr_set_dest_op(neg_in, result);
1975     aia_instr_set_src_op(neg_in, 0, v->prev_result);
1976
1977     Aia_Instr *jmp = __aia_1op_instr(v->aia, __AIA_JMP, -1, AIA_LONG,
1978         v->curr_loc);
1979     aia_instr_set_src_op(jmp, 0, end_lbl);
1980
1981     __aia_block(v->aia);
1982     __aia_insert_jump_label_instr(v->aia,
1983         aia_operand_label_get_name(skip_neg));
1984
1985     mov_instr(v, AIA_MOV, result, v->prev_result, AIA_LONG, AIA_LONG);
1986
1987     jmp = __aia_1op_instr(v->aia, __AIA_JMP, -1, AIA_LONG, v->curr_loc);
1988     aia_instr_set_src_op(jmp, 0, end_lbl);
1989

```

```

1990     __aia_block(v->aia);
1991     __aia_insert_jmp_label_instr(v->aia,
1992         aia_operand_label_get_name(end_lbl));
1993
1994     v->prev_result = result;
1995 }
1996
1997 ASTVF_BEGIN(AST_EXPR_ABS, Ast_Visitor_Aia, v, Ast_Node_Unary, n)
1998     Aia_Operand *iconst;
1999     v->curr_loc = ast_node_get_file_location(AST_NODE_OF(n));
2000
2001     n->expr->accept_visitor(n->expr, AST_VISITOR_OF(v));
2002
2003     switch (v->prev_expr_type) {
2004     case AST_EXPR_TYPE_BOOL:
2005         v->prev_result = movz_tmp_instr(v, v->prev_result);
2006         long_integer_abs(v);
2007         break;
2008
2009     case AST_EXPR_TYPE_CHAR:
2010         v->prev_result = movs_tmp_instr(v, v->prev_result);
2011         /* Fall through. */
2012     case AST_EXPR_TYPE_INT:
2013         long_integer_abs(v);
2014         break;
2015
2016     case AST_EXPR_TYPE_ARY:
2017         iconst = aia_operand_const_int_alloc(v->aia, -4);
2018         v->prev_result = aia_operand_addr_ref_alloc(v->aia,
2019             NULL,
2020             iconst,
2021             v->prev_result,
2022             NULL,
2023             NULL);
2024         break;
2025     default:
2026         fatal_error(S("Unexpected " QFY("| |") "-operator operand\n"));
2027     }
2028
2029     v->prev_expr_type = AST_EXPR_TYPE_INT;
2030     v->aia_expr_type = AIA_EXPR_STANDARD;
2031 ASTVF_END
2032
2033 ASTVF_BEGIN(AST_EXPR_INT, Ast_Visitor_Aia, v, Ast_Expr_Int, n)
2034     v->curr_loc = ast_node_get_file_location(AST_NODE_OF(n));
2035     v->prev_result = aia_operand_const_int_alloc(v->aia, n->val);
2036     v->prev_expr_type = AST_EXPR_TYPE_INT;
2037     v->aia_expr_type = AIA_EXPR_STANDARD;
2038 ASTVF_END
2039
2040 ASTVF_BEGIN(AST_EXPR_BOOL, Ast_Visitor_Aia, v, Ast_Expr_Bool, n)
2041     v->curr_loc = ast_node_get_file_location(AST_NODE_OF(n));
2042     v->prev_result = aia_operand_const_int_alloc(v->aia, n->val);
2043     v->prev_expr_type = AST_EXPR_TYPE_BOOL;
2044     v->aia_expr_type = AIA_EXPR_STANDARD;
2045 ASTVF_END
2046
2047 ASTVF_BEGIN(AST_EXPR_NULL, Ast_Visitor_Aia, v, Ast_Expr_Null, n)
2048     v->curr_loc = ast_node_get_file_location(AST_NODE_OF(n));
2049     v->prev_result = aia_operand_const_int_alloc(v->aia, 0);
2050     v->prev_expr_type = AST_EXPR_TYPE_NULL;
2051     v->aia_expr_type = AIA_EXPR_STANDARD;
2052 ASTVF_END
2053
2054 static void variable_iden_record_field_ref(Ast_Visitor_Aia *v,
2055     String field_name, Symbol_Type_Rec *selfrec, Aia_Operand *selfop)
2056 {
2057     int32_t field_offset;
2058     Aia_Operand *off_op, *ref_op;
2059     field_offset = aia_visitor_get_rec_field_offset(v,
2060         selfrec, field_name);
2061     off_op = aia_operand_const_int_alloc(v->aia,
2062         field_offset);
2063     ref_op = aia_operand_addr_ref_alloc(v->aia,

```

```

2064     NULL, off_op, selfop, NULL, NULL);
2065     v->prev_result = ref_op;
2066 }
2067
2068 static void variable_iden_record_vmt_mov(Ast_Visitor_Aia *v,
2069     Aia_Operand *selfop)
2070 {
2071     Aia_Operand *ref_op, *tmp;
2072     tmp = get_tmp_op(v);
2073     ref_op = aia_operand_addr_ref_alloc(v->aia,
2074     NULL, NULL, selfop, NULL, NULL);
2075     mov_instr(v, AIA_MOV, tmp, ref_op, AIA_LONG, AIA_LONG);
2076     v->prev_result = tmp;
2077 }
2078
2079 static Aia_Operand *get_curr_display_selfptr(Ast_Visitor_Aia *v)
2080 {
2081     Aia_Operand *selfptr;
2082
2083     Symbol_Table_Node *tmp = v->curr_display->sym_node;
2084     while (tmp->parent->type != SYMBOL_TABLE_NODE_REC)
2085         tmp = tmp->parent;
2086
2087     if (tmp == v->curr_display->sym_node) {
2088         selfptr = v->aia->record_self_ptr;
2089     } else {
2090         Int display;
2091         Const_String display_func_name;
2092         display = stn_display_index(
2093             v->curr_display->sym_node->parent,
2094             tmp,
2095             &display_func_name);
2096
2097         Aia_Operand *tmp_disp = __aia_operand_local_ref_alloc(
2098             v->aia, generate_display_str(display) /*, AIA_LONG */);
2099         Aia_Operand *tmp_reg = aia_operand_tmp_reg_alloc(v->aia);
2100         mov_instr(v, AIA_MOV, tmp_reg, tmp_disp, AIA_LONG,
2101             AIA_LONG);
2102         selfptr = aia_operand_display_ref_alloc(v->aia,
2103             tmp_reg, SELF_STR, display_func_name /*, AIA_LONG */);
2104     }
2105
2106     return selfptr;
2107 }
2108
2109 ASTVF_BEGIN(AST_VARIABLE_IDEN, Ast_Visitor_Aia, v, Ast_Variable_Iden, n)
2110     v->curr_loc = ast_node_get_file_location(AST_NODE_OF(n));
2111
2112     Symbol_Table_Node *sym_node;
2113     if (v->dot_ref_record) {
2114         sym_node = v->dot_ref_record->rec_sym_node;
2115     } else {
2116         sym_node = ast_node_get_symbol_table_node(AST_NODE_OF(n));
2117     }
2118
2119     Symbol *prev_curr_sym = v->curr_symbol;
2120     Symbol *sym;
2121     if (v->next_property != SYMBOL_PROPERTY_VAR) {
2122         sym = symbol_table_node_lookup(sym_node, n->iden, v->next_property);
2123     } else {
2124         DLOG("var iden: %S from node %p\n", n->iden, sym_node);
2125         sym = symbol_table_node_lookup_var(sym_node, n->iden,
2126             ast_node_get_file_location(AST_NODE_OF(n)));
2127     }
2128     assert(sym);
2129     v->curr_symbol = sym;
2130
2131     on_symrec_gen_vitrec(sym->resolved_type, v);
2132
2133     if (v->next_variable_iden_is_operand) {
2134         Symbol_Table_Node *var_root = sym->sym_node;
2135         while (var_root->type == SYMBOL_TABLE_NODE_INTERMEDIATE)
2136             var_root = var_root->parent;
2137

```

```

2138     if (var_root->type == SYMBOL_TABLE_NODE_GLOBAL ||
2139         var_root->type == SYMBOL_TABLE_NODE_IMPORT) {
2140         assert(!v->dot_ref_record);
2141         v->prev_result = aia_operand_label_alloc(v->aia,
2142             sym->unique_name, 0);
2143     }
2144     } else if (var_root->type == SYMBOL_TABLE_NODE_FUNC) {
2145         assert(!v->dot_ref_record);
2146
2147         if (sym->resolved_type->methods->get_type() == SYMBOL_TYPE_FUNC) {
2148             v->prev_result = aia_operand_label_alloc(v->aia,
2149                 sym->unique_name, 0);
2150         } else {
2151             UNUSED uint8_t var_size = symbol_type_struct_to_aia_operand_size(
2152                 sym->resolved_type);
2153
2154             if (var_root == v->curr_display->sym_node) {
2155                 v->prev_result = aia_operand_local_ref_alloc(v->aia,
2156                     sym->unique_name /* , var_size */);
2157             } else {
2158                 Const_String func_name;
2159                 Int idx = stn_display_index(
2160                     v->curr_display->sym_node->parent,
2161                     var_root,
2162                     &func_name);
2163                 Aia_Operand *tmp_disp = __aia_operand_local_ref_alloc(
2164                     v->aia, generate_display_str(idx) /*, AIA_LONG */);
2165                 /*
2166                 Aia_Operand *tmp_reg = aia_operand_tmp_reg_alloc(v->aia);
2167                 mov_instr(v, AIA_MOV, tmp_reg, tmp_disp, AIA_LONG,
2168                     AIA_LONG);
2169                 */
2170                 v->prev_result = aia_operand_display_ref_alloc(v->aia,
2171                     tmp_disp, sym->unique_name, func_name
2172                     /*, var_size */);
2173             }
2174         }
2175     } else { // SYMBOL_TABLE_NODE_REC
2176         assert(v->dot_ref_record || v->curr_record_selfptr);
2177         assert(v->curr_record_selfptr || v->prev_result);
2178
2179         Symbol_Type_Struct *field = sym->resolved_type;
2180
2181         Aia_Operand *selfptr;
2182         Symbol_Type_Rec *selfrec;
2183
2184         bool is_direct_ref;
2185         if (v->dot_ref_record) {
2186             selfptr = v->prev_result;
2187             is_direct_ref = v->dot_ref_is_direct;
2188             if (is_direct_ref) {
2189                 if (v->alloc_record_selfptr) {
2190                     selfrec = v->alloc_record_selfptr;
2191                     v->alloc_record_selfptr = NULL;
2192                 } else {
2193                     selfrec = v->curr_record_selfptr;
2194                 }
2195             } else {
2196                 selfrec = v->dot_ref_record;
2197             }
2198         } else {
2199             is_direct_ref = false;
2200             selfptr = get_curr_display_selfptr(v);
2201             selfrec = v->curr_record_selfptr;
2202         }
2203
2204         switch (field->methods->get_type()) {
2205             case SYMBOL_TYPE_FUNC;
2206             int32_t cast_offset;
2207             if (!is_direct_ref) {
2208                 variable_iden_record_vmt_mov(v, selfptr);
2209                 // v->prev_result is pointing at VMT at the moment.
2210
2211                 Hash_Map_Slot *vmt_slot = hash_map_get(&v->vmt_map,

```

```

2212         selfrec->unique_name,
2213         string_hash_code(selfrec->unique_name));
2214     Vit_Vmt *vmt = VIT_VMT_OF(vmt_slot);
2215     Hash_Map_Slot *fe_slot = hash_map_get(&vmt->func_off_map,
2216     sym->identifier,
2217     string_hash_code(sym->identifier));
2218     Vit_Func_Off_Entry *fe = VIT_FUNC_OFF_ENTRY_OF(fe_slot);
2219
2220     Aia_Operand *disp = aia_operand_const_int_alloc(v->aia,
2221     fe->func_offset);
2222     v->prev_result = aia_operand_addr_ref_alloc(v->aia,
2223     NULL, disp, v->prev_result, NULL, NULL);
2224     cast_offset = (int32_t)fe->cast_offset;
2225 } else {
2226     v->prev_result = aia_operand_label_alloc(v->aia,
2227     sym->unique_name, 0);
2228     cast_offset = aia_visitor_get_rec_cast_offset(v,
2229     SYMBOL_TYPE_STRUCT_OF_CONTAINER(selfrec),
2230     SYMBOL_TYPE_STRUCT_OF_CONTAINER(v->dot_ref_record)
2231     );
2232 }
2233
2234     if (cast_offset) {
2235         v->func_call_selfptr = get_tmp_op(v);
2236         Aia_Instr *add_in = __aia_2op_instr(v->aia, AIA_ADD,
2237         AIA_LONG, AIA_LONG, v->curr_loc);
2238         aia_instr_set_dest_op(add_in, v->func_call_selfptr);
2239         aia_instr_set_src_op(add_in, 0,
2240         aia_operand_const_int_alloc(v->aia,
2241         cast_offset));
2242         aia_instr_set_src_op(add_in, 1, selfptr);
2243     } else {
2244         v->func_call_selfptr = selfptr;
2245     }
2246     break;
2247
2248     default:
2249         variable_iden_record_field_ref(v, sym->identifier,
2250         selfrec, selfptr);
2251         break;
2252 }
2253 }
2254
2255     v->dot_ref_record = NULL;
2256     v->prev_variable_result = v->prev_result;
2257     v->prev_sym_type = sym->resolved_type;
2258     v->prev_expr_type = ast_variable_iden_get_expr_type(n);
2259 } else {
2260     DEBUG(v->prev_sym_type = NULL; v->prev_result = NULL);
2261 }
2262
2263     v->aia_expr_type = AIA_EXPR_STANDARD;
2264     v->prev_iden = n->iden;
2265     v->curr_symbol = prev_curr_sym;
2266 ASTVF_END
2267
2268 ASTVF_BEGIN(AST_EXPR_ARY_REF, Ast_Visitor_Aia, v, Ast_Node_Binary, n)
2269     Symbol_Type_Ary *ary;
2270     Aia_Operand *ary_op;
2271     Ast_Expr_Type result_type;
2272
2273     v->curr_loc = ast_node_get_file_location(AST_NODE_OF(n));
2274
2275     EXPRESSION_SETUP(v);
2276
2277     EXPRESSION_ACCEPT_VISITOR(n->lhs, v);
2278     ary_op = v->prev_result;
2279     assert(v->prev_sym_type->methods->get_type() == SYMBOL_TYPE_ARY ||
2280     v->prev_sym_type->methods->get_type() == SYMBOL_TYPE_STRING);
2281
2282     if (v->prev_sym_type->methods->get_type() == SYMBOL_TYPE_ARY)
2283         ary = SYMBOL_TYPE_STRUCT_CONTAINER(v->prev_sym_type, Symbol_Type_Ary);
2284     else
2285         ary = NULL;

```

```

2286
2287     EXPRESSION_ACCEPT_VISITOR(n->rhs, v);
2288     if (v->prev_expr_type == AST_EXPR_TYPE_CHAR)
2289         type_cast_prev_char_to_long(v, AST_EXPR_TYPE_INT);
2290     else if (v->prev_expr_type == AST_EXPR_TYPE_BOOL)
2291         type_cast_prev_bool_to_long(v, AST_EXPR_TYPE_INT);
2292     // v->prev_result == long array index at the moment
2293
2294     result_type = ast_expr_binary_get_expr_type(AST_EXPR_BINARY_OF(n));
2295
2296     Aia_Operand *scale;
2297     if (ast_expr_type_to_aia_operand_size(result_type) == AIA_LONG)
2298         scale = aia_operand_const_int_alloc(v->aia, 4);
2299     else
2300         scale = NULL;
2301
2302     v->prev_result = aia_operand_addr_ref_alloc(v->aia,
2303         NULL, NULL, ary_op, v->prev_result, scale);
2304
2305     EXPRESSION_END(v);
2306     if (ary)
2307         v->prev_sym_type = ary->ary_type;
2308     v->prev_expr_type = result_type;
2309     v->aia_expr_type = AIA_EXPR_STANDARD;
2310 ASTVF_END
2311
2312 static NORETURN void func_call_unexpected_arg()
2313 {
2314     fatal_error(S("Unexpected function call argument. Aborting...\n"));
2315 }
2316
2317 static void func_call_set_prev_result(Ast_Visitor_Aia *v,
2318     Ast_Expr_Type param_type, Symbol_Type_Struct *param_s,
2319     Symbol_Type_Struct *arg_s)
2320 {
2321     switch (v->prev_expr_type) {
2322     case AST_EXPR_TYPE_INT:
2323         switch (param_type) {
2324         case AST_EXPR_TYPE_INT:
2325             break;
2326         case AST_EXPR_TYPE_BOOL:
2327             type_cast_prev_to_bool(v, AIA_LONG);
2328             type_cast_prev_bool_to_long(v, AST_EXPR_TYPE_INT);
2329             break;
2330         case AST_EXPR_TYPE_CHAR:
2331             break;
2332         default:
2333             func_call_unexpected_arg();
2334         }
2335         break;
2336
2337     case AST_EXPR_TYPE_BOOL:
2338         type_cast_prev_bool_to_long(v, AST_EXPR_TYPE_INT);
2339         break;
2340
2341     case AST_EXPR_TYPE_CHAR:
2342         switch (param_type) {
2343         case AST_EXPR_TYPE_BOOL:
2344             type_cast_prev_to_bool(v, AIA_BYTE);
2345             type_cast_prev_bool_to_long(v, AST_EXPR_TYPE_INT);
2346             break;
2347         case AST_EXPR_TYPE_CHAR:
2348             /* Fall. */
2349         case AST_EXPR_TYPE_INT:
2350             type_cast_prev_char_to_long(v, AST_EXPR_TYPE_INT);
2351             break;
2352         default:
2353             func_call_unexpected_arg();
2354         }
2355         break;
2356
2357     case AST_EXPR_TYPE_STRING:
2358         switch (param_type) {
2359         case AST_EXPR_TYPE_BOOL:

```

```

2360         type_cast_prev_to_bool(v, AIA_LONG);
2361         type_cast_prev_bool_to_long(v, AST_EXPR_TYPE_INT);
2362         break;
2363     case AST_EXPR_TYPE_STRING:
2364         break;
2365     default:
2366         func_call_unexpected_arg();
2367     }
2368     break;
2369
2370 case AST_EXPR_TYPE_ARY:
2371     switch (param_type) {
2372     case AST_EXPR_TYPE_BOOL:
2373         type_cast_prev_to_bool(v, AIA_LONG);
2374         type_cast_prev_bool_to_long(v, AST_EXPR_TYPE_INT);
2375         break;
2376     case AST_EXPR_TYPE_ARY:
2377         break;
2378     default:
2379         func_call_unexpected_arg();
2380     }
2381     break;
2382
2383 case AST_EXPR_TYPE_NULL:
2384     break;
2385
2386 case AST_EXPR_TYPE_REC:
2387     switch (param_type) {
2388     case AST_EXPR_TYPE_BOOL:
2389         type_cast_prev_to_bool(v, AIA_LONG);
2390         type_cast_prev_bool_to_long(v, AST_EXPR_TYPE_INT);
2391         break;
2392     case AST_EXPR_TYPE_REC:
2393         type_cast_rec_rec(v, param_s, arg_s, NULL, true);
2394         break;
2395     default:
2396         func_call_unexpected_arg();
2397     }
2398     break;
2399
2400 default:
2401     fatal_error(S("unexpected function call arg. Aborting...\n"));
2402 }
2403 }
2404
2405 ASTVF_BEGIN(AST_EXPR_FUNC_CALL, Ast_Visitor_Aia, v, Ast_Expr_Func_Call, n)
2406     v->curr_loc = ast_node_get_file_location(AST_NODE_OF(n));
2407
2408     Vector *arguments, *parameters;
2409     Ast_Node *arg_node;
2410     Symbol_Type_Struct *param;
2411     Aia_Operand *call_operand;
2412
2413     EXPRESSION_SETUP(v);
2414
2415     Symbol_Property saved_property = v->next_property;
2416     v->next_property = SYMBOL_PROPERTY_FUNC;
2417     // We visit VARIABLE_IDEN node now. It might need v->prev_result
2418     // so we don't use EXPRESSION_ACCEPT_VISITOR() here.
2419     n->identifier->accept_visitor(n->identifier, AST_VISITOR_OF(v));
2420     v->next_property = saved_property;
2421     call_operand = v->prev_result;
2422
2423     Symbol_Type_Func *func = ast_expr_func_call_get_func(n);
2424
2425     uint8_t ret_size = symbol_type_struct_to_aia_operand_size(
2426         func->return_type);
2427
2428     arguments = n->arguments;
2429     parameters = &func->param_types;
2430     Uns size = vector_size(parameters);
2431     assert(vector_size(arguments) == size);
2432
2433     VECTOR(tmp_arg_ops);

```



```

2434
2435     Aia_Operand *tmp_arg;
2436     if (v->func_call_selfptr) {
2437         tmp_arg = aia_operand_tmp_reg_alloc(v->aia);
2438         mov_instr(v, AIA_MOV, tmp_arg, v->func_call_selfptr,
2439             AIA_LONG, AIA_LONG);
2440         vector_append(&tmp_arg_ops, tmp_arg);
2441     }
2442
2443     Aia_Operand *func_self = v->func_call_selfptr;
2444
2445     v->func_call_selfptr = NULL;
2446     for (Uns i = 0; i < size; i++) {
2447         arg_node = vector_get(arguments, i);
2448         param = vector_get(parameters, i);
2449
2450         EXPRESSION_ACCEPT_VISITOR(arg_node, v);
2451         Symbol_Type_Struct *arg = v->prev_sym_type;
2452
2453         Ast_Expr_Type ptype = symbol_type_to_expr_type(param);
2454
2455         func_call_set_prev_result(v, ptype, param, arg);
2456         tmp_arg = aia_operand_tmp_reg_alloc(v->aia);
2457         mov_instr(v, AIA_MOV, tmp_arg, v->prev_result, AIA_LONG, AIA_LONG);
2458         vector_append(&tmp_arg_ops, tmp_arg);
2459     }
2460     v->func_call_selfptr = func_self;
2461
2462     int32_t arg_idx = 0;
2463     VECTOR_FOR_EACH_ENTRY(&tmp_arg_ops, tmp_arg) {
2464         Aia_Operand *arg_op = aia_operand_arg_alloc(v->aia, arg_idx++);
2465         mov_instr(v, AIA_MOV, arg_op, tmp_arg, AIA_LONG, AIA_LONG);
2466     }
2467     vector_clear(&tmp_arg_ops);
2468
2469     Aia_Instr *call_in = __aia_lop_instr(v->aia, AIA_CALL, ret_size,
2470         AIA_LONG, v->curr_loc);
2471
2472     v->prev_result = get_tmp_op(v);
2473     aia_instr_set_dest_op(call_in, v->prev_result);
2474     aia_instr_set_src_op(call_in, 0, call_operand);
2475
2476     v->func_call_selfptr = NULL;
2477     v->prev_expr_type = ast_expr_func_call_get_expr_type(n);
2478     v->aia_expr_type = AIA_EXPR_STANDARD;
2479     v->prev_sym_type = func->return_type;
2480     EXPRESSION_END(v);
2481 ASTVF_END
2482
2483 static bool direct_ref_is_concrete(Ast_Node *n)
2484 {
2485     assert(n->accept_visitor ==
2486         AST_NODE_ACCEPT_VISITOR_FUNC(AST_EXPR_DIRECT_REF));
2487     Ast_Node_Binary *bin = AST_CONTAINER_OF(n, Ast_Node_Binary);
2488
2489     if (bin->rhs->accept_visitor ==
2490         AST_NODE_ACCEPT_VISITOR_FUNC(AST_EXPR_FUNC_CALL)) {
2491         Ast_Expr_Func_Call *lhs = AST_CONTAINER_OF(bin->rhs,
2492             Ast_Expr_Func_Call);
2493         if (!lhs->func->is_concrete_func)
2494             return false;
2495     }
2496
2497     return true;
2498 }
2499
2500 static void rec_dot_ref(Ast_Visitor_Aia *v, Ast_Node_Binary *n,
2501     bool is_direct, Aia_Operand *rec_ptr, bool release_lhs_result)
2502 {
2503     Symbol_Type_Rec *rec;
2504
2505     EXPRESSION_SETUP(v);
2506
2507     v->curr_loc = ast_node_get_file_location(AST_NODE_OF(n));

```

```

2508
2509     bool prev_is_direct = v->dot_ref_is_direct;
2510     Symbol_Type_Rec *prev_dot_record = v->dot_ref_record;
2511
2512     v->prev_result = NULL;
2513
2514     if (n->lhs) {
2515         Ast_Node *lhs = n->lhs;
2516
2517         if (is_direct && lhs->accept_visitor ==
2518             AST_NODE_ACCEPT_VISITOR_FUNC(AST_EXPR_CAST)) {
2519             Ast_Node_Binary *tmp = AST_CONTAINER_OF(lhs, Ast_Node_Binary);
2520             lhs = tmp->lhs;
2521         }
2522
2523         lhs->accept_visitor(lhs, AST_VISITOR_OF(v));
2524         if (release_lhs_result && v->prev_result) {
2525             __aia_operand_acquire(v->prev_result);
2526             __aia_operand_release(v->prev_result);
2527         }
2528
2529         assert(v->prev_sym_type->methods->get_type() == SYMBOL_TYPE_REC);
2530         rec = SYMBOL_TYPE_STRUCT_CONTAINER(v->prev_sym_type, Symbol_Type_Rec);
2531     } else {
2532         rec = v->curr_record_selfptr;
2533     }
2534
2535     if (rec_ptr)
2536         v->prev_result = rec_ptr;
2537
2538     v->dot_ref_is_direct = is_direct;
2539     v->dot_ref_record = rec;
2540     // We visit VARIABLE_IDEN node now. It will need v->prev_result
2541     // so we don't use EXPRESSION_ACCEPT_VISITOR() here.
2542     n->rhs->accept_visitor(n->rhs, AST_VISITOR_OF(v));
2543
2544     v->prev_dot_ref_record = rec;
2545     v->dot_ref_is_direct = prev_is_direct;
2546     v->dot_ref_record = prev_dot_record;
2547
2548     EXPRESSION_END(v);
2549 }
2550
2551 ASTVF_BEGIN(AST_EXPR_DOT_REF, Ast_Visitor_Aia, v, Ast_Node_Binary, n)
2552     rec_dot_ref(v, n, false, NULL, false);
2553 ASTVF_END
2554
2555 ASTVF_BEGIN(AST_EXPR_DIRECT_REF, Ast_Visitor_Aia, v, Ast_Node_Binary, n)
2556     rec_dot_ref(v, n, true, v->aia->record_self_ptr, false);
2557 ASTVF_END
2558
2559 ASTVF_BEGIN(AST_SIMPLE_TYPE_INT, Ast_Visitor_Aia, v, Ast_Empty, n)
2560     (void)v;
2561     v->curr_loc = ast_node_get_file_location(AST_NODE_OF(n));
2562 ASTVF_END
2563
2564 ASTVF_BEGIN(AST_SIMPLE_TYPE_BOOL, Ast_Visitor_Aia, v, Ast_Empty, n)
2565     (void)n;
2566     v->curr_loc = ast_node_get_file_location(AST_NODE_OF(n));
2567 ASTVF_END
2568
2569 ASTVF_BEGIN(AST_TYPE_IDEN, Ast_Visitor_Aia, v, Ast_Type_Iden, n)
2570     Symbol *sym;
2571     sym = symbol_table_node_lookup(
2572         ast_node_get_symbol_table_node(AST_NODE_OF(n)),
2573         n->iden,
2574         SYMBOL_PROPERTY_TYPE_DEF);
2575     assert(sym);
2576     v->prev_sym_type = sym->resolved_type;
2577     v->curr_loc = ast_node_get_file_location(AST_NODE_OF(n));
2578 ASTVF_END
2579
2580 ASTVF_BEGIN(AST_VAR_DECL, Ast_Visitor_Aia, v, Ast_Node_Binary, n)
2581     v->curr_loc = ast_node_get_file_location(AST_NODE_OF(n));

```

```

2582
2583     Symbol *sym;
2584
2585     assert(!v->next_variable_iden_is_operand);
2586
2587     Symbol_Property saved_property = v->next_property;
2588     v->next_property = SYMBOL_PROPERTY_VAR;
2589     n->lhs->accept_visitor(n->lhs, AST_VISITOR_OF(v));
2590     v->next_property = saved_property;
2591
2592     Symbol *prev_curr_sym = v->curr_symbol;
2593     sym = symbol_table_node_lookup(
2594         ast_node_get_symbol_table_node(AST_NODE_OF(n)),
2595         v->prev_iden,
2596         SYMBOL_PROPERTY_VAR);
2597     assert(sym);
2598     v->curr_symbol = sym;
2599
2600     Symbol_Table_Node *node = ast_node_get_symbol_table_node(AST_NODE_OF(n));
2601     while (node->type == SYMBOL_TABLE_NODE_INTERMEDIATE)
2602         node = node->parent;
2603
2604     assert(node->type != SYMBOL_TABLE_NODE_IMPORT);
2605
2606     if (node->type == SYMBOL_TABLE_NODE_GLOBAL) {
2607         assert(!v->next_var_decl_is_param);
2608         Aia_Section_Type prev_sec = __aia_get_curr_section_type(v->aia);
2609         Const_String prev_fun = __aia_get_curr_func_name(v->aia);
2610         __aia_switch_section(v->aia, AIA_SECTION_DATA);
2611
2612         uint8_t src_size = symbol_type_struct_to_aia_operand_size(
2613             sym->resolved_type);
2614         Aia_Linkage linkage;
2615         if (v->stmt_list_nest == 1)
2616             linkage = AIA_LINKAGE_GLOBAL;
2617         else
2618             linkage = AIA_LINKAGE_PRIVATE;
2619
2620         __aia_insert_label_instr(v->aia,
2621             sym->unique_name,
2622             0,
2623             src_size ? 4 : 0,
2624             linkage,
2625             AIA_LABEL_TYPE_OBJ,
2626             src_size ? 4 : 1,
2627             v->curr_loc);
2628
2629         Aia_Operand *val = aia_operand_const_int_alloc(v->aia, 0);
2630         __aia_insert_const_val_instr(v->aia, val, src_size, v->curr_loc);
2631
2632         __aia_switch_section(v->aia, prev_sec);
2633         __aia_switch_func(v->aia, prev_fun);
2634     } else if (v->next_var_decl_is_param) {
2635         aia_func_append_param(v->aia, sym->unique_name);
2636         v->next_var_decl_is_param = false;
2637     } else if (node->type == SYMBOL_TABLE_NODE_FUNC) {
2638         __aia_func_append_local(v->aia, sym->unique_name);
2639     } else {
2640         // Ignore record variable declaration.
2641     }
2642
2643     on_symrec_gen_vitrec(sym->resolved_type, v);
2644     n->rhs->accept_visitor(n->rhs, AST_VISITOR_OF(v));
2645
2646     v->curr_symbol = prev_curr_sym;
2647     v->prev_sym_type = sym->resolved_type;
2648 ASTVF_END
2649
2650 ASTVF_BEGIN(AST_TYPE_DEF, Ast_Visitor_Aia, v, Ast_Node_Binary, n)
2651     v->curr_loc = ast_node_get_file_location(AST_NODE_OF(n));
2652
2653     Symbol *sym;
2654
2655     Symbol_Property saved_property = v->next_property;

```

```

2656     v->next_property = SYMBOL_PROPERTY_TYPE_DEF;
2657     n->lhs->accept_visitor(n->lhs, AST_VISITOR_OF(v));
2658     v->next_property = saved_property;
2659
2660     Symbol *prev_curr_sym = v->curr_symbol;
2661     sym = symbol_table_node_lookup(
2662         ast_node_get_symbol_table_node(AST_NODE_OF(n)),
2663         v->prev_iden,
2664         SYMBOL_PROPERTY_TYPE_DEF);
2665     assert(sym);
2666     v->curr_symbol = sym;
2667
2668     on_symrec_gen_vitrec(sym->resolved_type, v);
2669
2670     n->rhs->accept_visitor(n->rhs, AST_VISITOR_OF(v));
2671
2672     v->curr_symbol = prev_curr_sym;
2673     v->prev_sym_type = sym->resolved_type;
2674 ASTVF_END
2675
2676 ASTVF_BEGIN(AST_TYPE_ARY, Ast_Visitor_Aia, v, Ast_Type, n)
2677     v->curr_loc = ast_node_get_file_location(AST_NODE_OF(n));
2678     n->type->accept_visitor(n->type, AST_VISITOR_OF(v));
2679 ASTVF_END
2680
2681 ASTVF_BEGIN(AST_TYPE_REC, Ast_Visitor_Aia, v, Ast_Type_Rec, n)
2682     Vector *vec;
2683     Ast_Node *node;
2684     Symbol_Type tmp_type UNUSED;
2685     Symbol_Type_Struct *tmp_struct;
2686
2687     v->curr_loc = ast_node_get_file_location(AST_NODE_OF(n));
2688
2689     vec = n->extend_list;
2690     VECTOR_FOR_EACH_ENTRY(vec, node)
2691         node->accept_visitor(node, AST_VISITOR_OF(v));
2692
2693     Symbol_Type curr_type = v->curr_symbol->resolved_type->methods->get_type();
2694
2695     Symbol_Type_Rec *prev_rec_selfptr = v->curr_record_selfptr;
2696
2697     switch (curr_type) {
2698     case SYMBOL_TYPE_REC:
2699         v->curr_record_selfptr = SYMBOL_TYPE_STRUCT_CONTAINER(
2700             v->curr_symbol->resolved_type, Symbol_Type_Rec);
2701         break;
2702
2703     case SYMBOL_TYPE_FUNC:
2704         assert(v->func_ret_sym_type);
2705
2706         tmp_type = v->func_ret_sym_type->methods->get_type();
2707         assert(tmp_type == SYMBOL_TYPE_REC || tmp_type == SYMBOL_TYPE_ARY);
2708
2709         tmp_struct = on_symbol_type_ary_get_struct(
2710             v->func_ret_sym_type);
2711         assert(tmp_struct->methods->get_type() == SYMBOL_TYPE_REC);
2712
2713         v->curr_record_selfptr = SYMBOL_TYPE_STRUCT_CONTAINER(tmp_struct,
2714             Symbol_Type_Rec);
2715         break;
2716
2717     case SYMBOL_TYPE_ARY:
2718         tmp_struct = on_symbol_type_ary_get_struct(
2719             v->curr_symbol->resolved_type);
2720         assert(tmp_struct->methods->get_type() == SYMBOL_TYPE_REC);
2721
2722         v->curr_record_selfptr = SYMBOL_TYPE_STRUCT_CONTAINER(tmp_struct,
2723             Symbol_Type_Rec);
2724         break;
2725
2726     default:
2727         assert(false);
2728         break;
2729 }

```

```

2730
2731     vec = n->body;
2732     VECTOR_FOR_EACH_ENTRY(vec, node)
2733         node->accept_visitor(node, AST_VISITOR_OF(v));
2734
2735     v->curr_record_selfptr = prev_rec_selfptr;
2736 ASTVF_END
2737
2738 static void stmt_list_action(Ast_Visitor_Aia *v, Ast_Stmt_List *n)
2739 {
2740     v->curr_loc = ast_node_get_file_location(AST_NODE_OF(n));
2741
2742     Ast_Node *stmt;
2743     Vector *stmts = n->statements;
2744     ++v->stmt_list_nest;
2745     VECTOR_FOR_EACH_ENTRY(stmts, stmt) {
2746         stmt->accept_visitor(stmt, AST_VISITOR_OF(v));
2747         if (v->prev_result && v->prev_result->ref_count == 0) {
2748             __aia_operand_destroy(v->prev_result);
2749             v->prev_result = NULL;
2750         }
2751     }
2752     --v->stmt_list_nest;
2753 }
2754
2755 ASTVF_BEGIN(AST_FIN_STMT_LIST, Ast_Visitor_Aia, v, Ast_Stmt_List, n)
2756     Aia_Section_Type prev_sec_type = __aia_get_curr_section_type(v->aia);
2757     __aia_switch_section(v->aia, AIA_SECTION_FINI);
2758     stmt_list_action(v, n);
2759     __aia_switch_section(v->aia, prev_sec_type);
2760 ASTVF_END
2761
2762 ASTVF_BEGIN(AST_STMT_LIST, Ast_Visitor_Aia, v, Ast_Stmt_List, n)
2763     stmt_list_action(v, n);
2764 ASTVF_END
2765
2766 static bool func_def_insert_trampolines(Ast_Visitor_Aia *v,
2767     Const_String func_name, Vector *parameters)
2768 {
2769     Hash_Map_Slot *tr_slot = hash_map_get(&v->trampolines,
2770         (String)func_name, string_hash_code(func_name));
2771     if (tr_slot) {
2772         Vit_Vmt_Trampoline_Entry *tre = VIT_VMT_TRAMPOLINE_ENTRY_OF(tr_slot);
2773         Vit_Vmt_Trampoline *tr;
2774         Aia_Func *func = __aia_get_curr_func(v->aia);
2775         __aia_switch_func(v->aia, NULL);
2776
2777         VECTOR_FOR_EACH_ENTRY(&tre->trampolines, tr) {
2778             Aia_Func_Trampoline *func_tramp = __aia_func_trampoline_alloc(
2779                 v->aia, tr->tramp_name, parameters, func);
2780
2781             __aia_set_func_trampoline_block(v->aia, func_tramp);
2782
2783             __aia_insert_label_instr(v->aia, tr->tramp_name, 0, 4,
2784                 aia_func_get_linkage(func), AIA_LABEL_TYPE_FUNC, 0,
2785                 aia_get_null_location(v->aia));
2786
2787             Aia_Operand *tmp = get_tmp_op(v);
2788             Aia_Instr *add_in = __aia_2op_instr(v->aia, AIA_ADD,
2789                 AIA_LONG, AIA_LONG, aia_get_null_location(v->aia));
2790             aia_instr_set_dest_op(add_in, tmp);
2791             Aia_Operand *this_op = v->aia->record_self_ptr;
2792             Aia_Operand *iconst = aia_operand_const_int_alloc(v->aia,
2793                 -tr->rec_offset);
2794             aia_instr_set_src_op(add_in, 0, this_op);
2795             aia_instr_set_src_op(add_in, 1, iconst);
2796             this_op = v->aia->record_self_ptr;
2797             mov_instr(v, AIA_MOV, this_op, tmp, AIA_LONG, AIA_LONG);
2798
2799             Aia_Operand *func_lbl = aia_operand_label_alloc(v->aia,
2800                 func_name, 0);
2801             Aia_Instr *jmp_in = __aia_1op_instr(v->aia, __AIA_JMP, -1,
2802                 AIA_LONG, aia_get_null_location(v->aia));
2803             aia_instr_set_src_op(jmp_in, 0, func_lbl);

```

```

2804     }
2805     v->aia->curr_block = NULL;
2806     v->aia->curr_blist = NULL;
2807     return true;
2808 }
2809 return false;
2810 }
2811
2812 static Symbol *rec_get_finalize_func(Symbol_Type_Rec *rec)
2813 {
2814     Symbol_Func_Map *fmap = symbol_table_node_get_func_map(
2815         rec->rec_sym_node,
2816         (String)S("finalize"),
2817         string_hash_code(S("finalize")));
2818
2819     if (!fmap) // Then the record does not have finalize func
2820         return NULL;
2821
2822     /* Record must only have 1 finalize func. */
2823     assert(vector_size(&fmap->overload_idens) == 1);
2824     Symbol *sym = vector_get(&fmap->overload_idens, 0);
2825     return sym;
2826 }
2827
2828 static Symbol *rec_get_default_rec_func(Symbol_Type_Rec *rec)
2829 {
2830     Symbol_Func_Map *fmap = symbol_table_node_get_func_map(
2831         rec->rec_sym_node,
2832         (String)S("record"),
2833         string_hash_code(S("record")));
2834
2835     if (!fmap) // Then the record does not have record func
2836         return NULL;
2837
2838     Symbol_Type_Func *func = NULL;
2839     Symbol *sym = NULL;
2840     VECTOR_FOR_EACH_ENTRY(&fmap->overload_idens, sym) {
2841         assert(sym->resolved_type->methods->get_type() == SYMBOL_TYPE_FUNC);
2842         func = SYMBOL_TYPE_STRUCT_CONTAINER(sym->resolved_type,
2843             Symbol_Type_Func);
2844         if (vector_is_empty(&func->param_identifiers))
2845             break;
2846     }
2847
2848     assert(func);
2849     assert(sym);
2850
2851     return sym;
2852 }
2853
2854 static void default_func_call(Ast_Visitor_Aia *v, Vit_Record *vrec,
2855     Symbol_Type_Rec *rec, Symbol *func_sym, Aia_Operand *self_ptr)
2856 {
2857     if (!func_sym)
2858         return;
2859
2860     Vit_Record_Field *rec_field = vit_record_get_field(vrec, rec->unique_name);
2861     /* if (!rec_field) it's assumed it's the record itself. */
2862     DEBUGT(def,
2863         if (!rec_field) {
2864             Symbol_Type_Struct *rs = SYMBOL_TYPE_STRUCT_OF_CONTAINER(rec);
2865             Symbol_Type_Struct *fs = SYMBOL_TYPE_STRUCT_OF_CONTAINER(vrec->rec);
2866             assert(rs->methods->same_type(rs, fs));
2867         }
2868     );
2869
2870     Aia_Operand *arg = aia_operand_arg_alloc(v->aia, 0);
2871     if (rec_field && rec_field->field_offset) {
2872         Aia_Operand *tmp = get_tmp_op(v);
2873         Aia_Operand *off = aia_operand_const_int_alloc(v->aia,
2874             rec_field->field_offset);
2875         Aia_Instr *add_in = __aia_2op_instr(v->aia, AIA_ADD,
2876             AIA_LONG, AIA_LONG, v->curr_loc);
2877         aia_instr_set_dest_op(add_in, tmp);

```

```

2878     aia_instr_set_src_op(add_in, 0, self_ptr);
2879     aia_instr_set_src_op(add_in, 1, off);
2880     mov_instr(v, AIA_MOV, arg, tmp, AIA_LONG, AIA_LONG);
2881 } else {
2882     mov_instr(v, AIA_MOV, arg, self_ptr, AIA_LONG, AIA_LONG);
2883 }
2884
2885 Aia_Instr *call_in = __aia_lop_instr(v->aia, AIA_CALL, -1, AIA_LONG,
2886     v->curr_loc);
2887 Aia_Operand *call_op = aia_operand_label_alloc(v->aia,
2888     func_sym->unique_name, 0);
2889 aia_instr_set_src_op(call_in, 0, call_op);
2890 }
2891
2892 static void __func_finalize_call(Ast_Visitor_Aia *v, Vit_Record *vrec,
2893     Symbol_Type_Rec *rec, Aia_Operand *self_ptr)
2894 {
2895     Symbol *fin_func_sym = rec_get_finalize_func(rec);
2896     default_func_call(v, vrec, rec, fin_func_sym, self_ptr);
2897 }
2898
2899 static void func_finalize_call(Ast_Visitor_Aia *v, Vit_Record *vrec,
2900     Symbol_Type_Rec *rec)
2901 {
2902     __func_finalize_call(v, vrec, rec, v->aia->record_self_ptr);
2903 }
2904
2905 static void __func_record_call(Ast_Visitor_Aia *v, Vit_Record *vrec,
2906     Symbol_Type_Rec *rec, Aia_Operand *self_ptr)
2907 {
2908     Symbol *rec_func_sym = rec_get_default_rec_func(rec);
2909     default_func_call(v, vrec, rec, rec_func_sym, self_ptr);
2910 }
2911
2912 static void func_record_call(Ast_Visitor_Aia *v, Vit_Record *vrec,
2913     Symbol_Type_Rec *rec)
2914 {
2915     __func_record_call(v, vrec, rec, v->aia->record_self_ptr);
2916 }
2917
2918 static void func_finalize_setup_vit_rec(Ast_Visitor_Aia *v, Vit_Record *vrec)
2919 {
2920     Symbol_Type_Struct *etype;
2921     VECTOR_FOR_EACH_ENTRY(&vrec->extended_sym_types, etype)
2922         func_finalize_call(v, vrec,
2923             SYMBOL_TYPE_STRUCT_CONTAINER(etype, Symbol_Type_Rec));
2924 }
2925
2926 static void func_record_setup_vit_rec(Ast_Visitor_Aia *v, Vit_Record *vrec,
2927     Vector *bases_missing_call)
2928 {
2929     Symbol_Type_Struct *etype;
2930     VECTOR_FOR_EACH_ENTRY(bases_missing_call, etype)
2931         func_record_call(v, vrec,
2932             SYMBOL_TYPE_STRUCT_CONTAINER(etype, Symbol_Type_Rec));
2933 }
2934
2935 static void func_finalize_setup(Ast_Visitor_Aia *v, Symbol_Type_Rec *rec)
2936 {
2937     Hash_Map_Slot *slot = hash_map_get(&v->record_map, rec->unique_name,
2938         string_hash_code(rec->unique_name));
2939     assert(slot);
2940
2941     Vit_Record *vrec = VIT_RECORD_OF(slot);
2942     assert(!vrec->rec->missing_finalize_name);
2943
2944     func_finalize_setup_vit_rec(v, vrec);
2945 }
2946
2947 static void func_record_setup(Ast_Visitor_Aia *v, Symbol_Type_Rec *rec,
2948     Vector *bases_missing_call)
2949 {
2950     if (vector_is_empty(bases_missing_call))
2951         return;

```

```

2952
2953     Hash_Map_Slot *slot = hash_map_get(&v->record_map, rec->unique_name,
2954     string_hash_code(rec->unique_name));
2955     assert(slot);
2956
2957     Vit_Record *vrec = VIT_RECORD_OF(slot);
2958     assert(!vrec->rec->missing_record_func_name);
2959
2960     func_record_setup_vit_rec(v, vrec, bases_missing_call);
2961 }
2962
2963 static void func_aia(Ast_Visitor_Aia *v, Ast_Func_Def *n,
2964     Aia_Visitor_Func_Type func_type)
2965 {
2966     v->curr_loc = ast_node_get_file_location(AST_NODE_OF(n));
2967
2968     Vector *vec;
2969     Ast_Node *node;
2970     String func_iden;
2971     Symbol *sym;
2972     Vector *dpres_vec;
2973     Ast_Expr_Type func_ret_type;
2974
2975     Symbol_Type_Struct *prev_func_ret = v->func_ret_sym_type;
2976
2977     Symbol_Property saved_property = v->next_property;
2978     v->next_property = SYMBOL_PROPERTY_FUNC;
2979     n->iden->accept_visitor(n->iden, AST_VISITOR_OF(v));
2980     v->next_property = saved_property;
2981     func_iden = v->prev_iden;
2982
2983     Symbol_Table_Node *func_stn = ast_node_get_symbol_table_node(
2984         n->statements);
2985
2986     Stn_Display_Preserve *prev_display = v->curr_display;
2987     v->curr_display = stn_display_preserve_get(func_stn);
2988     dpres_vec = &v->curr_display->stn_displays;
2989
2990     Symbol *prev_curr_sym = v->curr_symbol;
2991     sym = symbol_table_node_lookup(
2992         ast_node_get_symbol_table_node(AST_NODE_OF(n)),
2993         func_iden,
2994         SYMBOL_PROPERTY_FUNC);
2995     assert(sym);
2996     v->curr_symbol = sym;
2997     v->func_ret_sym_type = SYMBOL_TYPE_STRUCT_CONTAINER(sym->resolved_type,
2998         Symbol_Type_Func->return_type);
2999
3000     on_symrec_gen_vitrec(v->func_ret_sym_type, v);
3001
3002     v->aia_expr_type = AIA_EXPR_STANDARD;
3003     v->prev_expr_type = AST_EXPR_TYPE_INT;
3004     n->return_type->accept_visitor(n->return_type, AST_VISITOR_OF(v));
3005     func_ret_type = v->prev_expr_type;
3006
3007     Aia_Func *prev_func = __aia_get_curr_func(v->aia);
3008     Const_String prev_func_name = __aia_get_curr_func_name(v->aia);
3009
3010     Aia_Section_Type prev_sec = __aia_get_curr_section_type(v->aia);
3011     __aia_switch_section(v->aia, AIA_SECTION_TEXT);
3012
3013     bool func_needs_self_ptr = ast_node_get_symbol_table_node(
3014         AST_NODE_OF(n))->type == SYMBOL_TABLE_NODE_REC;
3015
3016     __aia_switch_func(v->aia, sym->unique_name);
3017     aia_set_curr_func_location(v->aia,
3018         ast_node_get_file_location(AST_NODE_OF(n)));
3019
3020     Const_String func_name;
3021     Symbol_Table_Node *tmp_node;
3022     VECTOR_FOR_EACH_ENTRY(dpres_vec, tmp_node) {
3023         Int idx;
3024         if (tmp_node != func_stn) {
3025             idx = stn_display_index(func_stn->parent, tmp_node, &func_name);

```



```

3026     aia_func_append_preserve_display(v->aia, idx);
3027 } else {
3028     aia_func_append_preserve_display(v->aia, -1);
3029 }
3030 }
3031
3032 if (func_needs_self_ptr)
3033     aia_func_append_param(v->aia, SELF_STR);
3034
3035 vec = n->parameters;
3036 VECTOR_FOR_EACH_ENTRY(vec, node) {
3037     v->next_var_decl_is_param = true;
3038     node->accept_visitor(node, AST_VISITOR_OF(v));
3039 }
3040
3041 if (!prev_func) {
3042     aia_func_set_linkage(v->aia, AIA_LINKAGE_GLOBAL);
3043 } else {
3044     __aia_set_curr_func_parent(v->aia, prev_func);
3045     ssize_t disp_num = vector_size(&prev_func->preserve_display_indices);
3046     while (--disp_num >= 0) {
3047         String disp_str = generate_display_str(disp_num);
3048         __aia_func_append_display_param(v->aia, disp_str);
3049     }
3050 }
3051
3052 if (func_type == AIAV_FUNC_TYPE_FINALIZE) {
3053     Symbol_Type_Rec *container =
3054         ast_node_get_symbol_table_node(AST_NODE_OF(n))->node_rec;
3055     assert(container);
3056     func_finalize_setup(v, container);
3057 }
3058
3059 Aia_Visitor_Func_Type prev_func_type = v->curr_func_type;
3060 v->curr_func_type = func_type;
3061 n->statements->accept_visitor(n->statements, AST_VISITOR_OF(v));
3062 v->curr_func_type = prev_func_type;
3063
3064 if (func_type == AIAV_FUNC_TYPE_FINALIZE) {
3065     Aia_Instr *ret_in = __aia_lop_instr(v->aia, AIA_RET, -1, AIA_LONG,
3066         v->curr_loc);
3067     Aia_Operand *selfptr = v->aia->record_self_ptr;
3068     aia_instr_set_src_op(ret_in, 0, selfptr);
3069 } else if (func_ret_type == AST_EXPR_TYPE_VOID) {
3070     __aia_0op_instr(v->aia, AIA_RET, AIA_LONG,
3071         v->curr_loc);
3072 }
3073
3074 bool inserted_tramp = func_def_insert_trampolines(v, sym->identifier,
3075     &v->aia->curr_func->parameters);
3076 (void)inserted_tramp;
3077 DEBUGT(def, if (inserted_tramp) assert(func_needs_self_ptr));
3078
3079 __aia_switch_section(v->aia, prev_sec);
3080 __aia_switch_func(v->aia, prev_func_name);
3081 v->curr_symbol = prev_curr_sym;
3082 v->curr_display = prev_display;
3083 v->func_ret_sym_type = prev_func_ret;
3084 }
3085
3086 ASTVF_BEGIN(AST_EXT_FUNC_DECL, Ast_Visitor_Aia, v, Ast_Func_Def, n)
3087     (void)n;
3088     v->curr_loc = ast_node_get_file_location(AST_NODE_OF(n));
3089 ASTVF_END
3090
3091 ASTVF_BEGIN(AST_FIN_FUNC_DEF, Ast_Visitor_Aia, v, Ast_Func_Def, n)
3092     func_aia(v, n, AIAV_FUNC_TYPE_FINALIZE);
3093 ASTVF_END
3094
3095 ASTVF_BEGIN(AST_REC_FUNC_DEF, Ast_Visitor_Aia, v, Ast_Func_Def, n)
3096     func_aia(v, n, AIAV_FUNC_TYPE_RECORD);
3097 ASTVF_END
3098
3099 ASTVF_BEGIN(AST_FUNC_DEF, Ast_Visitor_Aia, v, Ast_Func_Def, n)

```

```

3100     func_aia(v, n, AIAV_FUNC_TYPE_NORMAL);
3101 ASTVF_END
3102
3103 static void __if_stmt_expr(Ast_Visitor_Aia *v, uint8_t cmp_size,
3104     Aia_Operand *then_lbl, Aia_Operand *skip_lbl)
3105 {
3106     cmp0_skip_on_false(v, cmp_size, then_lbl, skip_lbl, __AIA_JE);
3107 }
3108
3109 static void if_stmt_expr(Ast_Visitor_Aia *v, Ast_Node *exp_node,
3110     Aia_Operand *then_lbl, Aia_Operand *skip_lbl)
3111 {
3112     uint8_t cmp_size;
3113     EXPRESSION_SETUP(v);
3114
3115     assert(!v->bool_true_label);
3116     assert(!v->bool_false_label);
3117
3118     v->bool_true_label = then_lbl;
3119     v->bool_false_label = skip_lbl;
3120     EXPRESSION_ACCEPT_VISITOR(exp_node, v);
3121     v->bool_false_label = NULL;
3122     v->bool_true_label = NULL;
3123
3124     if (v->aia_expr_type == AIA_EXPR_STANDARD) {
3125         cmp_size = ast_expr_type_to_aia_operand_size(v->prev_expr_type);
3126         __if_stmt_expr(v, cmp_size, then_lbl, skip_lbl);
3127     }
3128
3129     EXPRESSION_END(v);
3130 }
3131
3132 /* AIA block structure:
3133 *      (p)
3134 *      / \
3135 *      | (i)
3136 *      \ /
3137 *      (l) */
3138 ASTVF_BEGIN(AST_IF_STMT, Ast_Visitor_Aia, v, Ast_Node_Binary, n)
3139     v->curr_loc = ast_node_get_file_location(AST_NODE_OF(n));
3140
3141     Aia_Operand *if_thn = aia_operand_tmp_label_alloc(v->aia, S("IF.then"));
3142     Aia_Operand *ifend = aia_operand_tmp_label_alloc(v->aia, S("IF.end"));
3143
3144     if_stmt_expr(v, n->lhs, if_thn, ifend);
3145
3146     __aia_block(v->aia);
3147     __aia_insert_jump_label_instr(v->aia,
3148         aia_operand_label_get_name(if_thn));
3149     n->rhs->accept_visitor(n->rhs, AST_VISITOR_OF(v));
3150     Aia_Instr *jmp = __aia_lob_instr(v->aia, __AIA_JMP, -1, AIA_LONG,
3151         v->curr_loc);
3152     aia_instr_set_src_op(jmp, 0, ifend);
3153
3154     __aia_block(v->aia);
3155     __aia_insert_jump_label_instr(v->aia,
3156         aia_operand_label_get_name(ifend));
3157 ASTVF_END
3158
3159 /* AIA block structure:
3160 *      (p)
3161 *      / \
3162 *      (i) (e)
3163 *      \ /
3164 *      (l) */
3165 ASTVF_BEGIN(AST_IF_ELSE_STMT, Ast_Visitor_Aia, v, Ast_Node_Ternary, n)
3166     v->curr_loc = ast_node_get_file_location(AST_NODE_OF(n));
3167
3168     Aia_Operand *if_thn = aia_operand_tmp_label_alloc(v->aia, S("IF.then"));
3169     Aia_Operand *ifend = aia_operand_tmp_label_alloc(v->aia, S("IF.end"));
3170     Aia_Operand *else_body = aia_operand_tmp_label_alloc(v->aia, S("IF.else"));
3171
3172     if_stmt_expr(v, n->lhs, if_thn, else_body);
3173

```

```

3174     __aia_block(v->aia);
3175     __aia_insert_jump_label_instr(v->aia,
3176         aia_operand_label_get_name(if_thn));
3177
3178     n->mid->accept_visitor(n->mid, AST_VISITOR_OF(v));
3179     Aia_Instr *jmp = __aia_lop_instr(v->aia, __AIA_JUMP, -1, AIA_LONG,
3180         v->curr_loc);
3181     aia_instr_set_src_op(jmp, 0, ifend);
3182
3183     __aia_block(v->aia);
3184     __aia_insert_jump_label_instr(v->aia,
3185         aia_operand_label_get_name(else_body));
3186
3187     n->rhs->accept_visitor(n->rhs, AST_VISITOR_OF(v));
3188     jmp = __aia_lop_instr(v->aia, __AIA_JUMP, -1, AIA_LONG, v->curr_loc);
3189     aia_instr_set_src_op(jmp, 0, ifend);
3190
3191     __aia_block(v->aia);
3192     __aia_insert_jump_label_instr(v->aia,
3193         aia_operand_label_get_name(ifend));
3194     ASTVF_END
3195
3196     /* AIA block structure:
3197     *      (p)
3198     *      |  __
3199     *      (c) \
3200     *      /  \ |
3201     *      /  \ |
3202     *      (l) (b) */
3203     ASTVF_BEGIN(AST_WHILE_STMT, Ast_Visitor_Aia, v, Ast_Node_Binary, n)
3204         v->curr_loc = ast_node_get_file_location(AST_NODE_OF(n));
3205
3206         Aia_Operand *wend_lbl =
3207             aia_operand_tmp_label_alloc(v->aia, S("WHILE.end"));
3208         Aia_Operand *wcmp_lbl =
3209             aia_operand_tmp_label_alloc(v->aia, S("WHILE.cmp"));
3210         Aia_Operand *wtop_lbl =
3211             aia_operand_tmp_label_alloc(v->aia, S("WHILE.top"));
3212
3213         Aia_Instr *jmp = __aia_lop_instr(v->aia, __AIA_JUMP, -1, AIA_LONG,
3214             v->curr_loc);
3215         aia_instr_set_src_op(jmp, 0, wcmp_lbl);
3216
3217         __aia_block(v->aia);
3218         __aia_insert_jump_label_instr(v->aia,
3219             aia_operand_label_get_name(wtop_lbl));
3220
3221         n->rhs->accept_visitor(n->rhs, AST_VISITOR_OF(v));
3222         jmp = __aia_lop_instr(v->aia, __AIA_JUMP, -1, AIA_LONG, v->curr_loc);
3223         aia_instr_set_src_op(jmp, 0, wcmp_lbl);
3224
3225         __aia_block(v->aia);
3226         __aia_insert_jump_label_instr(v->aia,
3227             aia_operand_label_get_name(wcmp_lbl));
3228
3229         if_stmt_expr(v, n->lhs, wtop_lbl, wend_lbl);
3230
3231         __aia_block(v->aia);
3232         __aia_insert_jump_label_instr(v->aia,
3233             aia_operand_label_get_name(wend_lbl));
3234     ASTVF_END
3235
3236     ASTVF_BEGIN(AST_ALLOC_ARY, Ast_Visitor_Aia, v, Ast_Node_Binary, n)
3237         v->curr_loc = ast_node_get_file_location(AST_NODE_OF(n));
3238
3239         EXPRESSION_SETUP(v);
3240
3241         Symbol_Type_Ary *ary_sym;
3242         Aia_Operand *ary_op;
3243         Aia_Operand *tmp;
3244
3245         EXPRESSION_ACCEPT_VISITOR(n->lhs, v);
3246         assert(v->prev_sym_type->methods->get_type() == SYMBOL_TYPE_ARY);
3247         ary_sym = SYMBOL_TYPE_STRUCT_CONTAINER(v->prev_sym_type, Symbol_Type_Ary);

```

```

3248     ary_op = v->prev_result;
3249
3250     EXPRESSION_ACCEPT_VISITOR(n->rhs, v);
3251     switch (v->prev_expr_type) {
3252     case AST_EXPR_TYPE_CHAR:
3253         tmp = get_tmp_op(v);
3254         mov_instr(v, AIA_MOVS, tmp, v->prev_result, AIA_LONG, AIA_BYTE);
3255         v->prev_result = tmp;
3256         break;
3257
3258     case AST_EXPR_TYPE_BOOL:
3259         tmp = get_tmp_op(v);
3260         mov_instr(v, AIA_MOVZ, tmp, v->prev_result, AIA_LONG, AIA_BYTE);
3261         v->prev_result = tmp;
3262         break;
3263
3264     default:
3265         assert(ast_expr_type_to_aia_operand_size(v->prev_expr_type) ==
3266                AIA_LONG);
3267         break;
3268     }
3269
3270     Const_String func_name;
3271     uint8_t elm_size = symbol_type_struct_to_aia_operand_size(
3272         ary_sym->ary_type);
3273     if (elm_size == AIA_LONG)
3274         func_name = AIA_FUNC_ALLOCATEAL;
3275     else
3276         func_name = AIA_FUNC_ALLOCATEAB;
3277
3278     Aia_Operand *arg_op = aia_operand_arg_alloc(v->aia, 0);
3279     mov_instr(v, AIA_MOV, arg_op, v->prev_result, AIA_LONG, AIA_LONG);
3280
3281     Aia_Instr *call_in = __aia_lop_instr(v->aia, AIA_CALL, AIA_LONG,
3282         AIA_LONG, v->curr_loc);
3283     Aia_Operand *result = get_tmp_op(v);
3284     Aia_Operand *call_op = aia_operand_label_alloc(v->aia, func_name, 0);
3285     aia_instr_set_src_op(call_in, 0, call_op);
3286     aia_instr_set_dest_op(call_in, result);
3287     mov_instr(v, AIA_MOV, ary_op, result, AIA_LONG, AIA_LONG);
3288
3289     EXPRESSION_END(v);
3290 ASTVF_END
3291
3292 // Returns end label
3293 static Aia_Operand *record_alloc(Ast_Visitor_Aia *v, Ast_Node *rec_iden,
3294     bool call_default_ctor, Aia_Operand **alloc_result_out)
3295 {
3296     Symbol_Type_Rec *alloc_rec;
3297     EXPRESSION_SETUP(v);
3298
3299     v->allocating_record = true;
3300     v->allocated_record_offset = 0;
3301     EXPRESSION_ACCEPT_VISITOR(rec_iden, v);
3302     v->allocating_record = false;
3303     alloc_rec = SYMBOL_TYPE_STRUCT_CONTAINER(v->prev_sym_type,
3304         Symbol_Type_Rec);
3305
3306     Hash_Map_Slot *vit_slot = hash_map_get(&v->record_map,
3307         alloc_rec->unique_name, string_hash_code(alloc_rec->unique_name));
3308     assert(vit_slot);
3309     Vit_Record *vit_rec = VIT_RECORD_OF(vit_slot);
3310
3311     Aia_Operand *rec_size = aia_operand_const_int_alloc(v->aia,
3312         vit_rec->byte_size);
3313     Aia_Operand *arg_op = aia_operand_arg_alloc(v->aia, 0);
3314     mov_instr(v, AIA_MOV, arg_op, rec_size, AIA_LONG, AIA_LONG);
3315
3316     Aia_Instr *call_in = __aia_lop_instr(v->aia, AIA_CALL,
3317         AIA_LONG, AIA_LONG, v->curr_loc);
3318
3319     Const_String func_name = AIA_FUNC_ALLOCATE;
3320
3321     Aia_Operand *alloc_result = get_tmp_op(v);

```

```

3322 Aia_Operand *call_op = aia_operand_label_alloc(v->aia, func_name, 0);
3323 aia_instr_set_src_op(call_in, 0, call_op);
3324 aia_instr_set_dest_op(call_in, alloc_result);
3325
3326 Aia_Operand *end_lbl = NULL;
3327 if (alloc_result_out ||
3328     vit_rec->initializer_name ||
3329     v->allocated_record_offset ||
3330     (call_default_ctor && alloc_rec->rec_sym_node->has_record_func)) {
3331
3332     Aia_Operand *saved_prev_result = v->prev_result;
3333     v->prev_result = alloc_result;
3334
3335     Aia_Operand *do_alloc_lbl;
3336     Aia_Operand *null_lbl;
3337     null_lbl = aia_operand_tmp_label_alloc(v->aia, S("ALOC.null"));
3338     do_alloc_lbl = aia_operand_tmp_label_alloc(v->aia, S("ALOC.do"));
3339     end_lbl = aia_operand_tmp_label_alloc(v->aia, S("ALOC.end"));
3340     cmp0_skip_on_false(v, AIA_LONG, null_lbl, do_alloc_lbl, __AIA_JNE);
3341
3342     __aia_block(v->aia);
3343     __aia_insert_jump_label_instr(v->aia,
3344         aia_operand_label_get_name(null_lbl));
3345
3346     v->prev_result = saved_prev_result;
3347     Aia_Operand *iconst0 = aia_operand_const_int_alloc(v->aia, 0);
3348     mov_instr(v, AIA_MOV, v->prev_result, iconst0,
3349         AIA_LONG, AIA_LONG);
3350
3351     Aia_Instr *jmp = __aia_lop_instr(v->aia, __AIA_JMP, -1, AIA_LONG,
3352         v->curr_loc);
3353     aia_instr_set_src_op(jmp, 0, end_lbl);
3354
3355     __aia_block(v->aia);
3356     __aia_insert_jump_label_instr(v->aia,
3357         aia_operand_label_get_name(do_alloc_lbl));
3358 }
3359
3360 if (vit_rec->initializer_name) {
3361     arg_op = aia_operand_arg_alloc(v->aia, 0);
3362     mov_instr(v, AIA_MOV, arg_op, alloc_result, AIA_LONG, AIA_LONG);
3363 }
3364
3365 Aia_Operand *call_result;
3366 if (v->allocated_record_offset) {
3367     Aia_Instr *add_in = __aia_2op_instr(v->aia, AIA_ADD,
3368         AIA_LONG, AIA_LONG, v->curr_loc);
3369     Aia_Operand *offset = aia_operand_const_int_alloc(v->aia,
3370         -v->allocated_record_offset);
3371     call_result = get_tmp_op(v);
3372     aia_instr_set_src_op(add_in, 0, alloc_result);
3373     aia_instr_set_src_op(add_in, 1, offset);
3374     aia_instr_set_dest_op(add_in, call_result);
3375 } else {
3376     call_result = alloc_result;
3377 }
3378 mov_instr(v, AIA_MOV, v->prev_result, call_result,
3379     AIA_LONG, AIA_LONG);
3380
3381 if (vit_rec->initializer_name) {
3382     call_in = __aia_lop_instr(v->aia, AIA_CALL, -1, AIA_LONG,
3383         v->curr_loc);
3384     call_op = aia_operand_label_alloc(v->aia,
3385         vit_rec->initializer_name, 0);
3386     aia_instr_set_src_op(call_in, 0, call_op);
3387 }
3388
3389 if (call_default_ctor && alloc_rec->rec_sym_node->has_record_func)
3390     __func_record_call(v, vit_rec, alloc_rec, alloc_result);
3391
3392 EXPRESSION_END(v);
3393
3394 if (alloc_result_out)
3395     *alloc_result_out = alloc_result;

```

```

3396
3397     return end_lbl;
3398 }
3399
3400 ASTVF_BEGIN(AST_ALLOC_REC, Ast_Visitor_Aia, v, Ast_Node_Unary, n)
3401     v->curr_loc = ast_node_get_file_location(AST_NODE_OF(n));
3402     Aia_Operand *end_lbl = record_alloc(v, n->expr, true, NULL);
3403     if (end_lbl) {
3404         Aia_Instr *jmp = __aia_lop_instr(v->aia, __AIA_JMP, -1, AIA_LONG,
3405             v->curr_loc);
3406         aia_instr_set_src_op(jmp, 0, end_lbl);
3407         __aia_block(v->aia);
3408         __aia_insert_jump_label_instr(v->aia,
3409             aia_operand_label_get_name(end_lbl));
3410     }
3411 ASTVF_END
3412
3413 ASTVF_BEGIN(AST_ALLOC_REC_CALL, Ast_Visitor_Aia, v, Ast_Node_Binary, n)
3414     v->curr_loc = ast_node_get_file_location(AST_NODE_OF(n));
3415
3416     Aia_Operand *alloc_res;
3417     Aia_Operand *end_lbl = record_alloc(v, n->lhs, false, &alloc_res);
3418     assert(end_lbl);
3419     Symbol_Type_Rec *rec = SYMBOL_TYPE_STRUCT_CONTAINER(v->prev_sym_type,
3420         Symbol_Type_Rec);
3421
3422     Symbol_Type_Rec *prev_rec = v->alloc_record_selfptr;
3423     v->alloc_record_selfptr = rec;
3424
3425     rec_dot_ref(v, n, true, alloc_res, true);
3426
3427     Aia_Instr *jmp = __aia_lop_instr(v->aia, __AIA_JMP, -1, AIA_LONG,
3428         v->curr_loc);
3429     aia_instr_set_src_op(jmp, 0, end_lbl);
3430     __aia_block(v->aia);
3431     __aia_insert_jump_label_instr(v->aia,
3432         aia_operand_label_get_name(end_lbl));
3433
3434     v->alloc_record_selfptr = prev_rec;
3435
3436 ASTVF_END
3437
3438 static void finalize_call_prev_result(Ast_Visitor_Aia *v,
3439     Const_String func_name)
3440 {
3441     Aia_Operand *arg_op = aia_operand_arg_alloc(v->aia, 0);
3442     mov_instr(v, AIA_MOV, arg_op, v->prev_result, AIA_LONG, AIA_LONG);
3443
3444     Aia_Instr *call_in = __aia_lop_instr(v->aia, AIA_CALL, -1, AIA_LONG,
3445         v->curr_loc);
3446
3447     Aia_Operand *call_op = aia_operand_label_alloc(v->aia, func_name, 0);
3448     aia_instr_set_src_op(call_in, 0, call_op);
3449 }
3450
3451 static void record_call_finalize(Ast_Visitor_Aia *v, Symbol_Type_Rec *rec,
3452     Const_String func_name)
3453 {
3454     Hash_Map_Slot *slot = hash_map_get(&v->vmt_map,
3455         rec->unique_name, string_hash_code(rec->unique_name));
3456     assert(slot);
3457
3458     Aia_Operand *selfptr = v->prev_result;
3459
3460     Vit_Vmt *vmt = VIT_VMT_OF(slot);
3461     slot = hash_map_get(&vmt->func_off_map, (String)func_name,
3462         string_hash_code(func_name));
3463     assert(slot);
3464     Vit_Func_Off_Entry *e = VIT_FUNC_OFF_ENTRY_OF(slot);
3465
3466     variable_iden_record_vmt_mov(v, selfptr);
3467
3468     Aia_Operand *disp = aia_operand_const_int_alloc(v->aia, e->func_offset);
3469     Aia_Operand *call_op = aia_operand_addr_ref_alloc(v->aia,

```

```

3470     NULL, disp, v->prev_result, NULL, NULL);
3471     Aia_Operand *arg = aia_operand_arg_alloc(v->aia, 0);
3472     mov_instr(v, AIA_MOV, arg, selfptr, AIA_LONG, AIA_LONG);
3473     Aia_Instr *call_in = __aia_lop_instr(v->aia, AIA_CALL,
3474     AIA_LONG, AIA_LONG, v->curr_loc);
3475     aia_instr_set_src_op(call_in, 0, call_op);
3476     v->prev_result = get_tmp_op(v);
3477     aia_instr_set_dest_op(call_in, v->prev_result);
3478 }
3479
3480 /* AIA block structure:
3481 *      (p)
3482 *      / \
3483 *      | (i)
3484 *      \ /
3485 *      (l) */
3486 static void record_delete(Ast_Visitor_Aia *v, Symbol_Type_Rec *rec)
3487 {
3488     Aia_Operand *if_thn = aia_operand_tmp_label_alloc(v->aia, S("DEL.do"));
3489     Aia_Operand *ifend = aia_operand_tmp_label_alloc(v->aia, S("DEL.end"));
3490
3491     /* Don't try to delete record if it's a null ptr.
3492     * Thus we insert an if-statement here. */
3493     cmp0_skip_on_false(v, AIA_LONG, if_thn, ifend, __AIA_JE);
3494
3495     __aia_block(v->aia);
3496     __aia_insert_jump_label_instr(v->aia,
3497     aia_operand_label_get_name(if_thn));
3498
3499     Symbol *fsym = rec_get_finalize_func(rec);
3500     if (fsym) // Then the record has a finalize function
3501     record_call_finalize(v, rec, fsym->identifier);
3502
3503     finalize_call_prev_result(v, AIA_FUNC_DELETE);
3504
3505     Aia_Instr *jmp = __aia_lop_instr(v->aia, __AIA_JMP, -1, AIA_LONG,
3506     v->curr_loc);
3507     aia_instr_set_src_op(jmp, 0, ifend);
3508
3509     __aia_block(v->aia);
3510     __aia_insert_jump_label_instr(v->aia,
3511     aia_operand_label_get_name(ifend));
3512 }
3513
3514 static void array_delete(Ast_Visitor_Aia *v)
3515 {
3516     finalize_call_prev_result(v, AIA_FUNC_DELETE);
3517 }
3518
3519 ASTVF_BEGIN(AST_DELETE, Ast_Visitor_Aia, v, Ast_Node_Unary, n)
3520     v->curr_loc = ast_node_get_file_location(AST_NODE_OF(n));
3521
3522     EXPRESSION_SETUP(v);
3523
3524     EXPRESSION_ACCEPT_VISITOR(n->expr, v);
3525     switch (v->prev_expr_type) {
3526     case AST_EXPR_TYPE_REC:
3527         assert(v->prev_sym_type->methods->get_type() == SYMBOL_TYPE_REC);
3528         record_delete(v, SYMBOL_TYPE_STRUCT_CONTAINER(v->prev_sym_type,
3529         Symbol_Type_Rec));
3530         break;
3531     case AST_EXPR_TYPE_ARY:
3532         assert(v->prev_sym_type->methods->get_type() == SYMBOL_TYPE_ARY);
3533         array_delete(v);
3534         break;
3535     default:
3536         fatal_error(S("Delete of invalid type. Aborting...\n"));
3537     }
3538
3539     EXPRESSION_END(v);
3540 ASTVF_END
3541
3542 ASTVF_BEGIN(AST_WRITE_STMT, Ast_Visitor_Aia, v, Ast_Node_Unary, n)
3543     v->curr_loc = ast_node_get_file_location(AST_NODE_OF(n));

```

```

3544     EXPRESSION_SETUP(v);
3545     EXPRESSION_ACCEPT_VISITOR(n->expr, v);
3546
3547     Const_String func_name;
3548     switch (v->prev_expr_type) {
3549     case AST_EXPR_TYPE_INT:
3550         func_name = AIA_FUNC_WRITEINI;
3551         break;
3552     case AST_EXPR_TYPE_STRING:
3553         func_name = AIA_FUNC_WRITEINS;
3554         break;
3555     case AST_EXPR_TYPE_REC:
3556         func_name = AIA_FUNC_WRITEINR;
3557         break;
3558     case AST_EXPR_TYPE_ARY:
3559         func_name = AIA_FUNC_WRITEINA;
3560         break;
3561     case AST_EXPR_TYPE_NULL:
3562         func_name = AIA_FUNC_WRITEINN;
3563         break;
3564     case AST_EXPR_TYPE_CHAR:
3565         func_name = AIA_FUNC_WRITEINC;
3566         type_cast_char_to_long(v, v->prev_result, AST_EXPR_TYPE_INT);
3567         break;
3568     case AST_EXPR_TYPE_BOOL:
3569         func_name = AIA_FUNC_WRITEINB;
3570         type_cast_bool_to_long(v, v->prev_result, AST_EXPR_TYPE_INT);
3571         break;
3572     default:
3573         fatal_error(S("unexpected " QFY("write") " operand. Aborting...\n"));
3574     }
3575
3576     Aia_Operand *arg_op = aia_operand_arg_alloc(v->aia, 0);
3577     mov_instr(v, AIA_MOV, arg_op, v->prev_result, AIA_LONG, AIA_LONG);
3578
3579     Aia_Instr *call_in = __aia_lop_instr(v->aia, AIA_CALL, -1,
3580         AIA_LONG, v->curr_loc);
3581     Aia_Operand *call_op = aia_operand_label_alloc(v->aia, func_name, 0);
3582     aia_instr_set_src_op(call_in, 0, call_op);
3583
3584     EXPRESSION_END(v);
3585 ASTVF_END
3586
3587 static void return_stmt_unexpected_lhs()
3588 {
3589     fatal_error(S("Unexpected return type. Aborting...\n"));
3590 }
3591
3592 ASTVF_BEGIN(AST_RETURN_STMT, Ast_Visitor_Aia, v, Ast_Node_Unary, n)
3593     v->curr_loc = ast_node_get_file_location(AST_NODE_OF(n));
3594
3595     EXPRESSION_SETUP(v);
3596     EXPRESSION_ACCEPT_VISITOR(n->expr, v);
3597
3598     Aia_Operand *tmp;
3599     Ast_Expr_Type lhs_t = symbol_type_to_expr_type(v->func_ret_sym_type);
3600     Ast_Expr_Type rhs_t = v->prev_expr_type;
3601
3602     Aia_Instr *reti;
3603     switch (lhs_t) {
3604     case AST_EXPR_TYPE_BOOL:
3605         switch (rhs_t) {
3606         case AST_EXPR_TYPE_BOOL:
3607             break;
3608         case AST_EXPR_TYPE_CHAR:

```



```

3618         tmp = get_tmp_op(v);
3619         mov_prev_to_bool(v, AIA_BYTE, tmp);
3620         v->prev_result = tmp;
3621         break;
3622
3623     default:
3624         tmp = get_tmp_op(v);
3625         mov_prev_to_bool(v, AIA_LONG, tmp);
3626         v->prev_result = tmp;
3627         break;
3628
3629     }
3630     break;
3631
3632 case AST_EXPR_TYPE_INT:
3633     switch (rhs_t) {
3634     case AST_EXPR_TYPE_INT:
3635         break;
3636     case AST_EXPR_TYPE_CHAR:
3637         tmp = get_tmp_op(v);
3638         mov_instr(v, AIA_MOVS, tmp, v->prev_result, AIA_LONG, AIA_BYTE);
3639         v->prev_result = tmp;
3640         break;
3641     case AST_EXPR_TYPE_BOOL:
3642         tmp = get_tmp_op(v);
3643         mov_instr(v, AIA_MOVZ, tmp, v->prev_result, AIA_LONG, AIA_BYTE);
3644         v->prev_result = tmp;
3645         break;
3646     default:
3647         return_stmt_unexpected_lhs();
3648     }
3649     break;
3650
3651 case AST_EXPR_TYPE_CHAR:
3652     switch (rhs_t) {
3653     case AST_EXPR_TYPE_INT:
3654         /* Fall. */
3655     case AST_EXPR_TYPE_BOOL:
3656         /* Fall. */
3657     case AST_EXPR_TYPE_CHAR:
3658         break;
3659     default:
3660         return_stmt_unexpected_lhs();
3661     }
3662     break;
3663
3664 case AST_EXPR_TYPE_STRING:
3665     break;
3666
3667 case AST_EXPR_TYPE_ARY:
3668     break;
3669
3670 case AST_EXPR_TYPE_REC:
3671     switch (rhs_t) {
3672     case AST_EXPR_TYPE_NULL:
3673         break;
3674     case AST_EXPR_TYPE_REC:
3675         type_cast_rec_rec(v, v->func_ret_sym_type, v->prev_sym_type,
3676             NULL, true);
3677         break;
3678     default:
3679         return_stmt_unexpected_lhs();
3680     }
3681     break;
3682
3683 case AST_EXPR_TYPE_VOID:
3684     reti = __aia_0op_instr(v->aia, AIA_RET, -1, v->curr_loc);
3685     goto out;
3686
3687 default:
3688     fatal_error(S("unexpected return type. Aborting...\n"));
3689 }
3690
3691 uint8_t val_size = symbol_type_struct_to_aia_operand_size(

```

```

3692         v->func_ret_sym_type);
3693
3694     reti = __aia_lob_instr(v->aia, AIA_RET, -1, val_size,
3695         v->curr_loc);
3696     aia_instr_set_src_op(reti, 0, v->prev_result);
3697
3698 out:
3699     EXPRESSION_END(v);
3700 ASTVF_END
3701
3702 static void assignment_unexpected_lhs()
3703 {
3704     fatal_error(S("Unexpected assignment LHS. Aborting...\n"));
3705 }
3706
3707 ASTVF_BEGIN(AST_ASSIGNMENT, Ast_Visitor_Aia, v, Ast_Node_Binary, n)
3708     v->curr_loc = ast_node_get_file_location(AST_NODE_OF(n));
3709
3710     Aia_Operand *lhs_r, *rhs_r;
3711     Ast_Expr_Type lhs_t, rhs_t;
3712     Symbol_Type_Struct *lhs_s, *rhs_s;
3713
3714     EXPRESSION_SETUP(v);
3715
3716     EXPRESSION_ACCEPT_VISITOR(n->rhs, v);
3717     rhs_r = v->prev_result;
3718     rhs_t = v->prev_expr_type;
3719     rhs_s = v->prev_sym_type;
3720
3721     EXPRESSION_ACCEPT_VISITOR(n->lhs, v);
3722     lhs_r = v->prev_result;
3723     lhs_t = v->prev_expr_type;
3724     lhs_s = v->prev_sym_type;
3725
3726     v->prev_result = rhs_r;
3727
3728     switch (lhs_t) {
3729     case AST_EXPR_TYPE_BOOL:
3730         switch (rhs_t) {
3731         case AST_EXPR_TYPE_BOOL:
3732             mov_instr(v, AIA_MOV, lhs_r, rhs_r, AIA_BYTE, AIA_BYTE);
3733             break;
3734
3735         case AST_EXPR_TYPE_CHAR:
3736             mov_prev_to_bool(v, AIA_BYTE, lhs_r);
3737             break;
3738
3739         default:
3740             mov_prev_to_bool(v, AIA_LONG, lhs_r);
3741             break;
3742         }
3743     }
3744     break;
3745
3746     case AST_EXPR_TYPE_INT:
3747         switch (rhs_t) {
3748         case AST_EXPR_TYPE_INT:
3749             mov_instr(v, AIA_MOV, lhs_r, rhs_r, AIA_LONG, AIA_LONG);
3750             break;
3751
3752         case AST_EXPR_TYPE_CHAR:
3753             if (!aia_operand_is_reg(lhs_r)) {
3754                 rhs_r = movz_tmp_instr(v, rhs_r);
3755                 mov_instr(v, AIA_MOV, lhs_r, rhs_r, AIA_LONG, AIA_LONG);
3756             } else {
3757                 mov_instr(v, AIA_MOVS, lhs_r, rhs_r, AIA_LONG, AIA_BYTE);
3758             }
3759             break;
3760
3761         case AST_EXPR_TYPE_BOOL:
3762             if (!aia_operand_is_reg(lhs_r)) {
3763                 rhs_r = movz_tmp_instr(v, rhs_r);
3764                 mov_instr(v, AIA_MOV, lhs_r, rhs_r, AIA_LONG, AIA_LONG);
3765             } else {
3766                 mov_instr(v, AIA_MOVZ, lhs_r, rhs_r, AIA_LONG, AIA_BYTE);
3767             }
3768         }
3769     }

```

```

3766         break;
3767     default:
3768         assignment_unexpected_lhs();
3769     }
3770     break;
3771
3772     case AST_EXPR_TYPE_CHAR:
3773         switch (rhs_t) {
3774             case AST_EXPR_TYPE_INT:
3775                 mov_instr(v, AIA_MOV, lhs_r, rhs_r, AIA_BYTE, AIA_BYTE);
3776                 break;
3777             case AST_EXPR_TYPE_BOOL:
3778                 /* Fall. */
3779             case AST_EXPR_TYPE_CHAR:
3780                 mov_instr(v, AIA_MOV, lhs_r, rhs_r, AIA_BYTE, AIA_BYTE);
3781                 break;
3782             default:
3783                 assignment_unexpected_lhs();
3784         }
3785         break;
3786
3787     case AST_EXPR_TYPE_STRING:
3788         mov_instr(v, AIA_MOV, lhs_r, rhs_r, AIA_LONG, AIA_LONG);
3789         break;
3790
3791     case AST_EXPR_TYPE_ARY:
3792         mov_instr(v, AIA_MOV, lhs_r, rhs_r, AIA_LONG, AIA_LONG);
3793         break;
3794
3795     case AST_EXPR_TYPE_REC:
3796         switch (rhs_t) {
3797             case AST_EXPR_TYPE_NULL:
3798                 mov_instr(v, AIA_MOV, lhs_r, rhs_r, AIA_LONG, AIA_LONG);
3799                 break;
3800             case AST_EXPR_TYPE_REC:
3801                 type_cast_rec_rec(v, lhs_s, rhs_s, lhs_r, true);
3802                 break;
3803             default:
3804                 assignment_unexpected_lhs();
3805         }
3806         break;
3807
3808     default:
3809         fatal_error(S("unexpected variable assignemnt. Aborting...\n"));
3810 }
3811
3812 EXPRESSION_END(v);
3813 ASTVF_END
3814
3815 static void rec_struct_vec_remove(Vector *recs, Symbol_Type_Rec *remove_rec)
3816 {
3817     Symbol_Type_Struct *rem = SYMBOL_TYPE_STRUCT_OF_CONTAINER(remove_rec);
3818     for (Uns i = 0; i < vector_size(recs); i++) {
3819         Symbol_Type_Struct *tmp = vector_get(recs, i);
3820         if (tmp->methods->same_type(tmp, rem)) {
3821             vector_remove(recs, i);
3822             return;
3823         }
3824     }
3825 }
3826
3827 static void record_func_stmt_startup(Ast_Visitor_Aia *v, Ast_Stmt_List *n)
3828 {
3829     inline bool recs_equal(Symbol_Type_Rec *lhs, Symbol_Type_Rec *rhs)
3830     {
3831         extern bool ___type_def_symbol_types_equal(Symbol_Type_Struct *lhs,
3832             Symbol_Type_Struct *rhs);
3833         Symbol_Type_Struct *lhs_s = SYMBOL_TYPE_STRUCT_OF_CONTAINER(lhs);
3834         Symbol_Type_Struct *rhs_s = SYMBOL_TYPE_STRUCT_OF_CONTAINER(rhs);
3835         return ___type_def_symbol_types_equal(lhs_s, rhs_s);
3836     }
3837
3838     assert(v->curr_record_selfptr);
3839     Symbol_Type_Rec *self = v->curr_record_selfptr;

```

```

3840
3841     VECTOR(bases);
3842
3843     Vector *extended = &v->curr_record_selfptr->extended_types;
3844     Symbol_Type_Struct *ext;
3845     Symbol_Type_Rec *ext_r;
3846     VECTOR_FOR_EACH_ENTRY(extended, ext) {
3847         ext_r = SYMBOL_TYPE_STRUCT_CONTAINER(ext, Symbol_Type_Rec);
3848         if (ext_r->rec_sym_node->has_record_func)
3849             vector_append(&bases, ext);
3850     }
3851
3852     Vector *stmts = n->statements;
3853     assert(n->num_rec_ctor_stmts <= vector_size(stmts));
3854
3855     Ast_Node *rec_call;
3856     for (Uns i = 0; i < n->num_rec_ctor_stmts; i++) {
3857         DEBUGT(def, v->prev_dot_ref_record = NULL);
3858         rec_call = vector_get(stmts, i);
3859         if (direct_ref_is_concrete(rec_call)) {
3860             rec_call->accept_visitor(rec_call, AST_VISITOR_OF(v));
3861             assert(v->prev_dot_ref_record);
3862             assert(recs_equal(self, v->curr_record_selfptr));
3863             if (recs_equal(v->prev_dot_ref_record, self)) {
3864                 assert(n->num_rec_ctor_stmts == 1);
3865                 vector_clear(&bases);
3866                 break;
3867             }
3868             rec_struct_vec_remove(&bases, v->prev_dot_ref_record);
3869         }
3870     }
3871     func_record_setup(v, v->curr_record_selfptr, &bases);
3872     vector_clear(&bases);
3873 }
3874
3875 ASTVF_BEGIN(AST_FUNC_BODY, Ast_Visitor_Aia, v, Ast_Stmt_List, n)
3876     Vector *stmts;
3877     Ast_Node *node;
3878
3879     v->curr_loc = ast_node_get_file_location(AST_NODE_OF(n));
3880
3881     if (v->curr_func_type == AIAV_FUNC_TYPE_RECORD)
3882         record_func_stmt_startup(v, n);
3883
3884     stmts = n->statements;
3885     for (Uns i = 0; i < n->num_rec_ctor_stmts; i++) {
3886         node = vector_get(stmts, i);
3887         node->accept_visitor(node, AST_VISITOR_OF(v));
3888         if (v->prev_result && v->prev_result->ref_count == 0) {
3889             __aia_operand_destroy(v->prev_result);
3890             v->prev_result = NULL;
3891         }
3892     }
3893 ASTVF_END
3894
3895 ASTVF_BEGIN(AST_EXPR_CHAR, Ast_Visitor_Aia, v, Ast_Expr_Char, n)
3896     v->curr_loc = ast_node_get_file_location(AST_NODE_OF(n));
3897     v->prev_result = aia_operand_const_int_alloc(v->aia, n->val);
3898     v->prev_expr_type = AST_EXPR_TYPE_CHAR;
3899     v->aia_expr_type = AIA_EXPR_STANDARD;
3900 ASTVF_END
3901
3902 ASTVF_BEGIN(AST_EXPR_STRING, Ast_Visitor_Aia, v, Ast_Expr_String, n)
3903     v->curr_loc = ast_node_get_file_location(AST_NODE_OF(n));
3904
3905     Aia_Section_Type prev_sec = __aia_get_curr_section_type(v->aia);
3906     Const_String prev_fun = __aia_get_curr_func_name(v->aia);
3907
3908     __aia_switch_section(v->aia, AIA_SECTION_RODATA);
3909
3910     Aia_Operand *lbl = aia_operand_tmp_label_alloc(v->aia, S("STR"));
3911     __aia_insert_label_instr(v->aia,
3912         aia_operand_label_get_name(lbl),
3913         aia_operand_label_get_offset(lbl),

```

```

3914         0,
3915         AIA_LINKAGE_PRIVATE,
3916         AIA_LABEL_TYPE_OBJ,
3917         string_length(n->val) + 1,
3918         v->curr_loc);
3919
3920     Aia_Operand *str = aia_operand_const_string_alloc(v->aia, n->val);
3921     __aia_insert_string_instr(v->aia, str, v->curr_loc);
3922
3923     v->prev_expr_type = AST_EXPR_TYPE_STRING;
3924     v->aia_expr_type = AIA_EXPR_STANDARD;
3925     v->prev_result = aia_operand_label_addr_alloc(v->aia,
3926         aia_operand_label_get_name(lbl), 0);
3927
3928     __aia_operand_destroy(lbl);
3929
3930     __aia_switch_section(v->aia, prev_sec);
3931     __aia_switch_func(v->aia, prev_func);
3932 ASTVF_END
3933
3934 ASTVF_BEGIN(AST_IMPORT_STRING, Ast_Visitor_Aia, v, Ast_Module_String, n)
3935     (void)n;
3936     v->curr_loc = ast_node_get_file_location(AST_NODE_OF(n));
3937 ASTVF_END
3938
3939 ASTVF_BEGIN(AST_PACKAGE_STRING, Ast_Visitor_Aia, v, Ast_Module_String, n)
3940     (void)n;
3941     v->curr_loc = ast_node_get_file_location(AST_NODE_OF(n));
3942 ASTVF_END
3943
3944 ASTVF_BEGIN(AST_SIMPLE_TYPE_CHAR, Ast_Visitor_Aia, v, Ast_Empty, n)
3945     (void)n;
3946     v->curr_loc = ast_node_get_file_location(AST_NODE_OF(n));
3947 ASTVF_END
3948
3949 ASTVF_BEGIN(AST_SIMPLE_TYPE_VOID, Ast_Visitor_Aia, v, Ast_Empty, n)
3950     v->curr_loc = ast_node_get_file_location(AST_NODE_OF(n));
3951     v->prev_expr_type = AST_EXPR_TYPE_VOID;
3952     v->aia_expr_type = AIA_EXPR_STANDARD;
3953 ASTVF_END
3954
3955 ASTVF_BEGIN(AST_SIMPLE_TYPE_STRING, Ast_Visitor_Aia, v, Ast_Empty, n)
3956     (void)n;
3957     v->curr_loc = ast_node_get_file_location(AST_NODE_OF(n));
3958 ASTVF_END
3959
3960 ASTVF_BEGIN(AST_REC_SELF_PTR, Ast_Visitor_Aia, v, Ast_Empty, n)
3961     v->curr_loc = ast_node_get_file_location(AST_NODE_OF(n));
3962
3963     Symbol *sym = symbol_table_get_from_location(
3964         ast_node_get_symbol_table_node(AST_NODE_OF(n)),
3965         ast_node_get_file_location(AST_NODE_OF(n)));
3966
3967     v->prev_sym_type = sym->resolved_type;
3968     v->prev_expr_type = AST_EXPR_TYPE_REC;
3969     v->aia_expr_type = AIA_EXPR_STANDARD;
3970
3971     v->prev_iden = NULL;
3972     v->prev_variable_result = NULL;
3973     v->prev_result = get_curr_display_selfptr(v);
3974 ASTVF_END
3975
3976 static void ast_visitor_insert_initializer(Ast_Visitor_Aia *v,
3977     Vit_Vmt *vmt)
3978 {
3979     Vit_Record_Initializer *ini = &vmt->initializer;
3980     __aia_switch_section(v->aia, AIA_SECTION_TEXT);
3981     __aia_switch_func(v->aia, ini->initializer_name);
3982     aia_func_set_linkage(v->aia, AIA_LINKAGE_GLOBAL);
3983
3984     aia_func_append_param(v->aia, SELF_STR);
3985     Aia_Operand *self = v->aia->record_self_ptr;
3986
3987     Uns size = vector_size(&ini->vmt_offsets);

```

```

3988     assert(vector_size(&ini->rec_offsets) == size);
3989
3990     for (Uns i = 0; i < size; i++) {
3991         int32_t off = PTR_TO_INT(vector_get(&ini->vmt_offsets, i));
3992         Aia_Operand *vmt_op = aia_operand_label_addr_alloc(v->aia,
3993             vmt->record_name, off);
3994         off = PTR_TO_INT(vector_get(&ini->rec_offsets, i));
3995         Aia_Operand *disp = aia_operand_const_int_alloc(v->aia, off);
3996         Aia_Operand *addr_ref = aia_operand_addr_ref_alloc(v->aia,
3997             NULL, disp, self, NULL, NULL);
3998
3999         mov_instr(v, AIA_MOV, addr_ref, vmt_op, AIA_LONG, AIA_LONG);
4000     }
4001     __aia_0op_instr(v->aia, AIA_RET, -1, v->curr_loc);
4002 }
4003
4004 static void ast_visitor_insert_vmt(Ast_Visitor_Aia *v, Vit_Vmt *vmt)
4005 {
4006     __aia_switch_section(v->aia, AIA_SECTION_RODATA);
4007
4008     __aia_insert_label_instr(v->aia,
4009         vmt->record_name,
4010         0,
4011         4,
4012         AIA_LINKAGE_GLOBAL,
4013         AIA_LABEL_TYPE_OBJ,
4014         vmt->current_func_offset,
4015         v->curr_loc);
4016
4017     Aia_Operand *lbl;
4018     String fname;
4019     VECTOR_FOR_EACH_ENTRY(&vmt->func_names, fname) {
4020         lbl = aia_operand_label_alloc(v->aia, fname, 0);
4021         __aia_insert_const_val_instr(v->aia, lbl, AIA_LONG, v->curr_loc);
4022     }
4023 }
4024
4025 static void ast_visitor_default_finalize(Ast_Visitor_Aia *v, Vit_Record *r)
4026 {
4027     __aia_switch_section(v->aia, AIA_SECTION_TEXT);
4028     __aia_switch_func(v->aia, r->rec->missing_finalize_name);
4029     aia_set_curr_func_location(v->aia, v->curr_loc);
4030     aia_func_set_linkage(v->aia, AIA_LINKAGE_GLOBAL);
4031
4032     aia_func_append_param(v->aia, SELF_STR);
4033
4034     func_finalize_setup_vit_rec(v, r);
4035
4036     Aia_Operand *self = v->aia->record_self_ptr;
4037     Aia_Instr *ret_in = __aia_1op_instr(v->aia, AIA_RET, -1, AIA_LONG,
4038         v->curr_loc);
4039     aia_instr_set_src_op(ret_in, 0, self);
4040
4041     func_def_insert_trampoline(v, r->rec->missing_finalize_name,
4042         &v->aia->curr_func->parameters);
4043 }
4044
4045 static void ast_visitor_default_rec_func(Ast_Visitor_Aia *v, Vit_Record *r)
4046 {
4047     __aia_switch_section(v->aia, AIA_SECTION_TEXT);
4048     __aia_switch_func(v->aia, r->rec->missing_record_func_name);
4049     aia_set_curr_func_location(v->aia, v->curr_loc);
4050     aia_func_set_linkage(v->aia, AIA_LINKAGE_GLOBAL);
4051
4052     aia_func_append_param(v->aia, SELF_STR);
4053
4054     func_record_setup_vit_rec(v, r, &r->rec->extended_types);
4055
4056     Aia_Operand *self = v->aia->record_self_ptr;
4057     Aia_Instr *ret_in = __aia_1op_instr(v->aia, AIA_RET, -1, AIA_LONG,
4058         v->curr_loc);
4059     aia_instr_set_src_op(ret_in, 0, self);
4060
4061     func_def_insert_trampoline(v, r->rec->missing_record_func_name,

```

```

4062         &v->aia->curr_func->parameters);
4063     }
4064
4065     static void ast_visitor_insert_def_functions(Ast_Visitor_Aia *v)
4066     {
4067         File_Location *prev_loc = v->curr_loc;
4068         v->curr_loc = aia_get_null_location(v->aia);
4069         Hash_Map_Slot *slot;
4070         Hash_Map *m = &v->record_map;
4071         HASH_MAP_FOR_EACH(m, slot) {
4072             Vit_Record *r = VIT_RECORD_OF(slot);
4073             if (r->rec->missing_finalize_name)
4074                 ast_visitor_default_finalize(v, r);
4075             if (r->rec->missing_record_func_name)
4076                 ast_visitor_default_rec_func(v, r);
4077         }
4078         v->curr_loc = prev_loc;
4079     }
4080
4081     static void ast_visitor_insert_vmts(Ast_Visitor_Aia *v)
4082     {
4083         File_Location *prev_loc = v->curr_loc;
4084         v->curr_loc = aia_get_null_location(v->aia);
4085         Hash_Map_Slot *vmt_slot;
4086         Hash_Map *m = &v->vmt_map;
4087         HASH_MAP_FOR_EACH(m, vmt_slot) {
4088             Vit_Vmt *vmt = VIT_VMT_OF(vmt_slot);
4089             if (!vmt->is_imported) {
4090                 ast_visitor_insert_vmt(v, vmt);
4091                 ast_visitor_insert_initializer(v, vmt);
4092             }
4093         }
4094         v->curr_loc = prev_loc;
4095     }
4096
4097     static void ast_visitor_dump_vit_record(FILE *stream, Vit_Record *r)
4098     {
4099         file_print_message(stream, S("# record %S\n# align: %U\n"),
4100             r->identifier,
4101             r->alignment);
4102         if (r->initializer_name)
4103             file_print_message(stream, S("# initializer: %S\n"),
4104                 r->initializer_name);
4105
4106         Vit_Record_Field *field;
4107         VECTOR_FOR_EACH_ENTRY(&r->field_vector, field) {
4108             file_print_message(stream, S("#\t%S offset(%U)\n"),
4109                 field->field_name,
4110                 field->field_offset);
4111         }
4112
4113         file_print_message(stream, S("# end record size(%U)\n\n"),
4114             r->byte_size, r->identifier);
4115     }
4116
4117     static void ast_visitor_dump_aia(Ast_Visitor_Aia *v,
4118         Const_String fname_prefix)
4119     {
4120         String fname = string_from_format(S("%S.vitaly.init-ic"), fname_prefix);
4121         FILE *stream = file_open(fname, S("w"));
4122         if (!stream)
4123             fatal_error(S("unable to create file %S for intermediate "
4124                 "code dump [%m]\n"), fname);
4125         string_destroy(fname);
4126
4127         Hash_Map_Slot *rec_slot;
4128         HASH_MAP_FOR_EACH(&v->record_map, rec_slot)
4129             ast_visitor_dump_vit_record(stream, VIT_RECORD_OF(rec_slot));
4130
4131         aia_dump(v->aia, stream);
4132
4133         file_close(stream);
4134     }
4135

```

```

4136 static String vit_name_to_c_name(Const_String rec_name)
4137 {
4138     String name = string_cpy_replace_all(rec_name, '.', '_');
4139     string_replace_all(name, '$', '_');
4140     return name;
4141 }
4142
4143 static void ast_visitor_dump_h_decl_rec(FILE *stream, Vit_Record *r)
4144 {
4145     String tmp = vit_name_to_c_name(r->identifier);
4146     file_print_message(stream, S("struct %S;\n"), tmp);
4147     string_destroy(tmp);
4148 }
4149
4150 static void ast_visitor_dump_h_def_vmt(FILE *stream, Vit_Vmt *vmt)
4151 {
4152     String name = vit_name_to_c_name(vmt->record_name);
4153     string_append(name, S("_vmt"));
4154
4155     file_print_message(stream, S("\nstruct %S {\n"), name);
4156     string_destroy(name);
4157
4158     Const_String func;
4159     VECTOR_FOR_EACH_ENTRY(&vmt->func_names, func) {
4160         String tmp = vit_name_to_c_name(func);
4161         file_print_message(stream,
4162             S("\tvoid *(*%S)(void *record, ...);\n"), tmp);
4163         string_destroy(tmp);
4164     }
4165
4166     file_print_message(stream, S("};\n"));
4167 }
4168
4169 static void ast_visitor_dump_h_def_rec(FILE *stream, Vit_Record *r,
4170     Ast_Visitor_Aia *v)
4171 {
4172     if (r->is_c_header_printed)
4173         return;
4174     r->is_c_header_printed = true;
4175
4176     Vit_Record_Field *field;
4177     VECTOR_FOR_EACH_ENTRY(&r->field_vector, field) {
4178         if (field->sym_type == SYMBOL_TYPE_IDEN) {
4179             Hash_Map_Slot *vit_slot = hash_map_get(&v->record_map,
4180                 field->field_name, string_hash_code(field->field_name));
4181             assert(vit_slot);
4182             ast_visitor_dump_h_def_rec(stream, VIT_RECORD_OF(vit_slot), v);
4183         }
4184     }
4185
4186     String name = vit_name_to_c_name(r->identifier);
4187     file_print_message(stream, S("\nstruct %S {\n"), name);
4188     string_destroy(name);
4189
4190     String tn;
4191     String fn;
4192
4193     VECTOR_FOR_EACH_ENTRY(&r->field_vector, field) {
4194         switch (field->sym_type) {
4195             case SYMBOL_TYPE_IDEN:
4196                 tn = vit_name_to_c_name(field->field_type_name);
4197                 fn = vit_name_to_c_name(field->field_name);
4198                 file_print_message(stream, S("\tstruct %S %S;\n"), tn, fn);
4199                 string_destroy(tn);
4200                 string_destroy(fn);
4201                 break;
4202             case SYMBOL_TYPE_INT:
4203                 file_print_message(stream, S("\tint %S;\n"), field->field_name);
4204                 break;
4205             case SYMBOL_TYPE_BOOL:
4206                 file_print_message(stream, S("\tbool %S;\n"), field->field_name);
4207                 break;
4208             case SYMBOL_TYPE_CHAR:
4209                 file_print_message(stream, S("\tchar %S;\n"), field->field_name);

```



```

4210         break;
4211     case SYMBOL_TYPE_STRING:
4212         file_print_message(stream, S("\tconst char *%S;\n"),
4213             field->field_name);
4214         break;
4215     case SYMBOL_TYPE_ARY:
4216         file_print_message(stream, S("\tvoid *%S;\n"),
4217             field->field_name);
4218         break;
4219     case SYMBOL_TYPE_REC:
4220         fn = vit_name_to_c_name(field->field_name);
4221         file_print_message(stream, S("\tvoid *%S;\n"), fn);
4222         string_destroy(fn);
4223         break;
4224     default:
4225         fatal_error(S("unexpected record filed type encountered "
4226             "while printing C header. Aborting...\n"));
4227     }
4228 }
4229
4230 file_print_message(stream, S("};\n"));
4231 }
4232
4233 static void ast_visitor_dump_h(Ast_Visitor_Aia *v, Const_String src_fname)
4234 {
4235     String fname = string_from_format(S("%S.vitaly.h"), src_fname);
4236     FILE *stream = file_open(fname, S("w"));
4237     if (!stream)
4238         fatal_error(S("unable to create C header file %S [%m]\n"), fname);
4239
4240     String def = string_cpy_replace_all(fname, '.', '_');
4241     string_toupper(def);
4242     string_destroy(fname);
4243
4244     file_print_message(stream, S("#ifndef %1$S\n#define %1$S\n"
4245         "#include <stdbool.h>\n\n"), def);
4246
4247     Hash_Map_Slot *slot;
4248     HASH_MAP_FOR_EACH(&v->record_map, slot)
4249         ast_visitor_dump_h_decl_rec(stream, VIT_RECORD_OF(slot));
4250
4251     HASH_MAP_FOR_EACH(&v->vmt_map, slot)
4252         ast_visitor_dump_h_def_vmt(stream, VIT_VMT_OF(slot));
4253
4254     HASH_MAP_FOR_EACH(&v->record_map, slot)
4255         ast_visitor_dump_h_def_rec(stream, VIT_RECORD_OF(slot), v);
4256
4257     file_print_message(stream, S("\n#endif // %S\n"), def);
4258     string_destroy(def);
4259
4260     file_close(stream);
4261 }
4262
4263 static Ast_Visitor_Aia aia_visitor = {
4264     .AST_VISITOR_FIELD = AST_VISITOR_STATIC_INIT(),
4265     .next_property = SYMBOL_PROPERTY_VAR,
4266     .null_location = FILE_LOCATION_STATIC_INIT(NULL, 0, 0)
4267 };
4268
4269 Aia *ast_visitor_aia_create(Ast *ast, Const_String src_fname)
4270 {
4271     if (ast_is_valid(ast) && !was_error_reported()) {
4272         ast_visitor_dependency_accept(ast);
4273
4274         file_location_set_file_name(&aia_visitor.null_location,
4275             ast_get_file_name(ast));
4276         aia_visitor.sym_table = ast_get_symbol_table(ast);
4277         aia_visitor.prev_result = NULL;
4278
4279         aia_visitor.record_map = HASH_MAP_INIT_SIZE(HASH_MAP_SIZE_23,
4280             (Hash_Map_Comparator) vit_record_comparator);
4281         aia_visitor.vmt_map = HASH_MAP_INIT_SIZE(HASH_MAP_SIZE_23,
4282             (Hash_Map_Comparator) vit_vmt_comparator);
4283         aia_visitor.trampolines = HASH_MAP_INIT_SIZE(HASH_MAP_SIZE_23,

```

```

4284         (Hash_Map_Comparator) vit_vmt_trampoline_entry_comparator);
4285
4286     aia_visitor.aia = aia_alloc(ast_get_file_name(ast));
4287     __aia_switch_section(aia_visitor.aia, AIA_SECTION_INIT);
4288
4289     ast->root->accept_visitor(ast->root, AST_VISITOR_OF(&aia_visitor));
4290
4291     assert(aia_visitor.next_property == SYMBOL_PROPERTY_VAR);
4292     assert(!aia_visitor.stmt_list_nest);
4293
4294     ast_visitor_insert_def_functions(&aia_visitor);
4295     ast_visitor_insert_vmts(&aia_visitor);
4296     __aia_finish(aia_visitor.aia);
4297
4298     if (cmdopts.dump_init_ic)
4299         ast_visitor_dump_aia(&aia_visitor, src_fname);
4300     if (cmdopts.dump_c_header)
4301         ast_visitor_dump_h(&aia_visitor, src_fname);
4302
4303     hash_map_for_each_destroy(&aia_visitor.record_map,
4304                             vit_record_hash_destroy);
4305     hash_map_for_each_destroy(&aia_visitor.vmt_map,
4306                             vit_vmt_hash_destroy);
4307     hash_map_for_each_destroy(&aia_visitor trampolines,
4308                             vit_vmt_trampoline_entry_hash_destroy);
4309
4310     stn_display_preserve_destroy();
4311
4312     return aia_visitor.aia;
4313 }
4314 return NULL;
4315 }

```

:

A.6.2 src/ast/ast_visitor_aia.h

```

1  #ifndef AST_VISITOR_AIA_H
2  #define AST_VISITOR_AIA_H
3
4  #include "ast_visitor.h"
5  #include <aia/aia.h>
6
7  Aia *ast_visitor_aia_create(Ast *ast, Const_String src_fname);
8
9  #endif // AST_VISITOR_AIA_H

```

:

A.6.3 src/ast/ast_visitor_delete.c

```

1  #include "ast_visitor_delete.h"
2
3  AST_VISITOR_STRUCT_BEGIN(Ast_Visitor_Delete)
4  AST_VISITOR_STRUCT_END(Ast_Visitor_Delete)
5
6  static void stmt_list_action(Ast_Visitor_Delete *v, Ast_Stmt_List *n) {
7      Ast_Node *stmt;
8      Vector *vec = n->statements;
9      VECTOR_FOR_EACH_ENTRY(vec, stmt)
10         stmt->accept_visitor(stmt, AST_VISITOR_OF(v));
11     vector_destroy(vec, NULL);
12     free_mem(n);
13 }
14
15 static void ternary_action(Ast_Visitor_Delete *v, Ast_Node_Ternary *n)
16 {

```

```

17     n->lhs->accept_visitor(n->lhs, AST_VISITOR_OF(v));
18     n->mid->accept_visitor(n->mid, AST_VISITOR_OF(v));
19     n->rhs->accept_visitor(n->rhs, AST_VISITOR_OF(v));
20     free_mem(n);
21 }
22
23 static void binary_action(Ast_Visitor_Delete *v, Ast_Node_Binary *n)
24 {
25     n->lhs->accept_visitor(n->lhs, AST_VISITOR_OF(v));
26     n->rhs->accept_visitor(n->rhs, AST_VISITOR_OF(v));
27     free_mem(n);
28 }
29
30 static void unary_action(Ast_Visitor_Delete *v, Ast_Node_Unary *n)
31 {
32     n->expr->accept_visitor(n->expr, AST_VISITOR_OF(v));
33     free_mem(n);
34 }
35
36 static void type_action(Ast_Visitor_Delete *v, Ast_Type *n)
37 {
38     n->type->accept_visitor(n->type, AST_VISITOR_OF(v));
39     free_mem(n);
40 }
41
42 ASTVF_BEGIN(AST_EXPR_LOR, Ast_Visitor_Delete, v, Ast_Node_Binary, n)
43     binary_action(v, n);
44 ASTVF_END
45
46 ASTVF_BEGIN(AST_EXPR_LAND, Ast_Visitor_Delete, v, Ast_Node_Binary, n)
47     binary_action(v, n);
48 ASTVF_END
49
50 ASTVF_BEGIN(AST_EXPR_EQ, Ast_Visitor_Delete, v, Ast_Node_Binary, n)
51     binary_action(v, n);
52 ASTVF_END
53
54 ASTVF_BEGIN(AST_EXPR_NEQ, Ast_Visitor_Delete, v, Ast_Node_Binary, n)
55     binary_action(v, n);
56 ASTVF_END
57
58 ASTVF_BEGIN(AST_EXPR_GT, Ast_Visitor_Delete, v, Ast_Node_Binary, n)
59     binary_action(v, n);
60 ASTVF_END
61
62 ASTVF_BEGIN(AST_EXPR_LT, Ast_Visitor_Delete, v, Ast_Node_Binary, n)
63     binary_action(v, n);
64 ASTVF_END
65
66 ASTVF_BEGIN(AST_EXPR_GTEQ, Ast_Visitor_Delete, v, Ast_Node_Binary, n)
67     binary_action(v, n);
68 ASTVF_END
69
70 ASTVF_BEGIN(AST_EXPR_LTEQ, Ast_Visitor_Delete, v, Ast_Node_Binary, n)
71     binary_action(v, n);
72 ASTVF_END
73
74 ASTVF_BEGIN(AST_EXPR_PLUS, Ast_Visitor_Delete, v, Ast_Node_Binary, n)
75     binary_action(v, n);
76 ASTVF_END
77
78 ASTVF_BEGIN(AST_EXPR_MINUS, Ast_Visitor_Delete, v, Ast_Node_Binary, n)
79     binary_action(v, n);
80 ASTVF_END
81
82 ASTVF_BEGIN(AST_EXPR_MUL, Ast_Visitor_Delete, v, Ast_Node_Binary, n)
83     binary_action(v, n);
84 ASTVF_END
85
86 ASTVF_BEGIN(AST_EXPR_DIV, Ast_Visitor_Delete, v, Ast_Node_Binary, n)
87     binary_action(v, n);
88 ASTVF_END
89
90 ASTVF_BEGIN(AST_EXPR_CAST, Ast_Visitor_Delete, v, Ast_Node_Binary, n)

```

```

91     binary_action(v, n);
92 ASTVF_END
93
94 ASTVF_BEGIN(AST_EXPR_LNOT, Ast_Visitor_Delete, v, Ast_Node_Unary, n)
95     unary_action(v, n);
96 ASTVF_END
97
98 ASTVF_BEGIN(AST_EXPR_ABS, Ast_Visitor_Delete, v, Ast_Node_Unary, n)
99     unary_action(v, n);
100 ASTVF_END
101
102 ASTVF_BEGIN(AST_EXPR_INT, Ast_Visitor_Delete, v, Ast_Expr_Int, n)
103     (void)v;
104     free_mem(n);
105 ASTVF_END
106
107 ASTVF_BEGIN(AST_EXPR_BOOL, Ast_Visitor_Delete, v, Ast_Expr_Bool, n)
108     (void)v;
109     free_mem(n);
110 ASTVF_END
111
112 ASTVF_BEGIN(AST_EXPR_NULL, Ast_Visitor_Delete, v, Ast_Expr_Null, n)
113     (void)v;
114     free_mem(n);
115 ASTVF_END
116
117 ASTVF_BEGIN(AST_VARIABLE_IDEN, Ast_Visitor_Delete, v, Ast_Variable_Iden, n)
118     (void)v;
119     string_destroy(n->iden);
120     free_mem(n);
121 ASTVF_END
122
123 ASTVF_BEGIN(AST_EXPR_ARY_REF, Ast_Visitor_Delete, v, Ast_Node_Binary, n)
124     binary_action(v, n);
125 ASTVF_END
126
127 ASTVF_BEGIN(AST_EXPR_FUNC_CALL, Ast_Visitor_Delete, v, Ast_Expr_Func_Call, n)
128     n->identifier->accept_visitor(n->identifier, AST_VISITOR_OF(v));
129     Vector *vargs = n->arguments;
130     Ast_Node *arg;
131     VECTOR_FOR_EACH_ENTRY(vargs, arg)
132         arg->accept_visitor(arg, AST_VISITOR_OF(v));
133     vector_destroy(vargs, NULL);
134     free_mem(n);
135 ASTVF_END
136
137 ASTVF_BEGIN(AST_EXPR_DOT_REF, Ast_Visitor_Delete, v, Ast_Node_Binary, n)
138     binary_action(v, n);
139 ASTVF_END
140
141 ASTVF_BEGIN(AST_EXPR_DIRECT_REF, Ast_Visitor_Delete, v, Ast_Node_Binary, n)
142     if (n->lhs) {
143         binary_action(v, n);
144     } else {
145         n->rhs->accept_visitor(n->rhs, AST_VISITOR_OF(v));
146         free_mem(n);
147     }
148 ASTVF_END
149
150 ASTVF_BEGIN(AST_SIMPLE_TYPE_INT, Ast_Visitor_Delete, v, Ast_Empty, n)
151     (void)v;
152     free_mem(n);
153 ASTVF_END
154
155 ASTVF_BEGIN(AST_SIMPLE_TYPE_BOOL, Ast_Visitor_Delete, v, Ast_Empty, n)
156     (void)v;
157     free_mem(n);
158 ASTVF_END
159
160 ASTVF_BEGIN(AST_TYPE_IDEN, Ast_Visitor_Delete, v, Ast_Type_Iden, n)
161     (void)v;
162     string_destroy(n->iden);
163     free_mem(n);
164 ASTVF_END

```

```

165
166 ASTVF_BEGIN(AST_VAR_DECL, Ast_Visitor_Delete, v, Ast_Node_Binary, n)
167     binary_action(v, n);
168 ASTVF_END
169
170 ASTVF_BEGIN(AST_TYPE_DEF, Ast_Visitor_Delete, v, Ast_Node_Binary, n)
171     binary_action(v, n);
172 ASTVF_END
173
174 ASTVF_BEGIN(AST_TYPE_ARY, Ast_Visitor_Delete, v, Ast_Type, n)
175     type_action(v, n);
176 ASTVF_END
177
178 ASTVF_BEGIN(AST_TYPE_REC, Ast_Visitor_Delete, v, Ast_Type_Rec, n)
179     Ast_Node *arg;
180     Vector *vargs = n->extend_list;
181     VECTOR_FOR_EACH_ENTRY(vargs, arg)
182         arg->accept_visitor(arg, AST_VISITOR_OF(v));
183     vector_destroy(vargs, NULL);
184
185     vargs = n->body;
186     VECTOR_FOR_EACH_ENTRY(vargs, arg)
187         arg->accept_visitor(arg, AST_VISITOR_OF(v));
188     vector_destroy(vargs, NULL);
189     free_mem(n);
190 ASTVF_END
191
192 ASTVF_BEGIN(AST_STMT_LIST, Ast_Visitor_Delete, v, Ast_Stmt_List, n)
193     stmt_list_action(v, n);
194 ASTVF_END
195
196 ASTVF_BEGIN(AST_FIN_STMT_LIST, Ast_Visitor_Delete, v, Ast_Stmt_List, n)
197     stmt_list_action(v, n);
198 ASTVF_END
199
200 static void delete_func(Ast_Visitor_Delete *v, Ast_Func_Def *n)
201 {
202     Ast_Node *p;
203     Vector *vec = n->parameters;
204     n->iden->accept_visitor(n->iden, AST_VISITOR_OF(v));
205     n->return_type->accept_visitor(n->return_type, AST_VISITOR_OF(v));
206     VECTOR_FOR_EACH_ENTRY(vec, p)
207         p->accept_visitor(p, AST_VISITOR_OF(v));
208     vector_destroy(vec, NULL);
209     n->statements->accept_visitor(n->statements, AST_VISITOR_OF(v));
210     free_mem(n);
211 }
212
213 ASTVF_BEGIN(AST_EXT_FUNC_DECL, Ast_Visitor_Delete, v, Ast_Func_Def, n)
214     Ast_Node *p;
215     n->iden->accept_visitor(n->iden, AST_VISITOR_OF(v));
216     n->return_type->accept_visitor(n->return_type, AST_VISITOR_OF(v));
217     n->extern_type->accept_visitor(n->extern_type, AST_VISITOR_OF(v));
218     Vector *vec = n->parameters;
219     VECTOR_FOR_EACH_ENTRY(vec, p)
220         p->accept_visitor(p, AST_VISITOR_OF(v));
221     vector_destroy(vec, NULL);
222     free_mem(n);
223 ASTVF_END
224
225 ASTVF_BEGIN(AST_FIN_FUNC_DEF, Ast_Visitor_Delete, v, Ast_Func_Def, n)
226     delete_func(v, n);
227 ASTVF_END
228
229 ASTVF_BEGIN(AST_REC_FUNC_DEF, Ast_Visitor_Delete, v, Ast_Func_Def, n)
230     delete_func(v, n);
231 ASTVF_END
232
233 ASTVF_BEGIN(AST_FUNC_DEF, Ast_Visitor_Delete, v, Ast_Func_Def, n)
234     delete_func(v, n);
235 ASTVF_END
236
237 ASTVF_BEGIN(AST_IF_STMT, Ast_Visitor_Delete, v, Ast_Node_Binary, n)
238     binary_action(v, n);

```

```

239 ASTVF_END
240
241 ASTVF_BEGIN(AST_IF_ELSE_STMT, Ast_Visitor_Delete, v, Ast_Node_Ternary, n)
242     ternary_action(v, n);
243 ASTVF_END
244
245 ASTVF_BEGIN(AST_ALLOC_ARY, Ast_Visitor_Delete, v, Ast_Node_Binary, n)
246     binary_action(v, n);
247 ASTVF_END
248
249 ASTVF_BEGIN(AST_ALLOC_REC, Ast_Visitor_Delete, v, Ast_Node_Unary, n)
250     unary_action(v, n);
251 ASTVF_END
252
253 ASTVF_BEGIN(AST_ALLOC_REC_CALL, Ast_Visitor_Delete, v, Ast_Node_Binary, n)
254     binary_action(v, n);
255 ASTVF_END
256
257 ASTVF_BEGIN(AST_DELETE, Ast_Visitor_Delete, v, Ast_Node_Unary, n)
258     unary_action(v, n);
259 ASTVF_END
260
261 ASTVF_BEGIN(AST_WHILE_STMT, Ast_Visitor_Delete, v, Ast_Node_Binary, n)
262     binary_action(v, n);
263 ASTVF_END
264
265 ASTVF_BEGIN(AST_RETURN_STMT, Ast_Visitor_Delete, v, Ast_Node_Unary, n)
266     unary_action(v, n);
267 ASTVF_END
268
269 ASTVF_BEGIN(AST_WRITE_STMT, Ast_Visitor_Delete, v, Ast_Node_Unary, n)
270     unary_action(v, n);
271 ASTVF_END
272
273 ASTVF_BEGIN(AST_ASSIGNMENT, Ast_Visitor_Delete, v, Ast_Node_Binary, n)
274     binary_action(v, n);
275 ASTVF_END
276
277 ASTVF_BEGIN(AST_FUNC_BODY, Ast_Visitor_Delete, v, Ast_Stmt_List, n)
278     stmt_list_action(v, n);
279 ASTVF_END
280
281 ASTVF_BEGIN(AST_EXPR_CHAR, Ast_Visitor_Delete, v, Ast_Expr_Char, n)
282     (void)v;
283     free_mem(n);
284 ASTVF_END
285
286 ASTVF_BEGIN(AST_EXPR_STRING, Ast_Visitor_Delete, v, Ast_Expr_String, n)
287     (void)v;
288     string_destroy(n->val);
289     free_mem(n);
290 ASTVF_END
291
292 ASTVF_BEGIN(AST_IMPORT_STRING, Ast_Visitor_Delete, v, Ast_Module_String, n)
293     (void)v;
294     string_destroy(n->module);
295     free_mem(n);
296 ASTVF_END
297
298 ASTVF_BEGIN(AST_PACKAGE_STRING, Ast_Visitor_Delete, v, Ast_Module_String, n)
299     (void)v;
300     string_destroy(n->module);
301     free_mem(n);
302 ASTVF_END
303
304 ASTVF_BEGIN(AST_SIMPLE_TYPE_CHAR, Ast_Visitor_Delete, v, Ast_Empty, n)
305     (void)v;
306     free_mem(n);
307 ASTVF_END
308
309 ASTVF_BEGIN(AST_SIMPLE_TYPE_STRING, Ast_Visitor_Delete, v, Ast_Empty, n)
310     (void)v;
311     free_mem(n);
312 ASTVF_END

```

```

313
314 ASTVF_BEGIN(AST_REC_SELF_PTR, Ast_Visitor_Delete, v, Ast_Empty, n)
315     (void)v;
316     free_mem(n);
317 ASTVF_END
318
319 ASTVF_BEGIN(AST_SIMPLE_TYPE_VOID, Ast_Visitor_Delete, v, Ast_Empty, n)
320     (void)v;
321     free_mem(n);
322 ASTVF_END
323
324 static Ast_Visitor_Delete delete_visitor = {
325     .AST_VISITOR_FIELD = AST_VISITOR_STATIC_INIT()
326 };
327
328 void ast_visitor_delete_accept_visitor(Ast_Node *root)
329 {
330     root->accept_visitor(root, AST_VISITOR_OF(&delete_visitor));
331 }

```

:

A.6.4 src/ast/ast_visitor_delete.h

```

1 #ifndef AST_VISITOR_DELETE_H
2 #define AST_VISITOR_DELETE_H
3
4 #include "ast_visitor.h"
5
6 void ast_visitor_delete_accept_visitor(Ast_Node *root);
7
8 #endif // AST_VISITOR_DELETE_H

```

:

A.6.5 src/ast/ast_visitor_dependency.c

```

1 #include "ast_visitor_dependency.h"
2 #include "symbol_table.h"
3
4 #undef DEBUG_TYPE
5 #define DEBUG_TYPE ast-dependency
6
7 typedef struct Stn_Dependency {
8     Symbol_Table_Node *sym_node;
9     Rb_Tree_Node rbnode;
10     bool is_func;
11     bool is_var;
12 } Stn_Dependency;
13
14 #define STN_DEPENDENCY_OF(node) RB_TREE_ENTRY(node, Stn_Dependency, rbnode)
15
16 static bool stn_display_preserve_comparator(Symbol_Table_Node *search_node,
17     Hash_Map_Slot *map_slot)
18 {
19     Stn_Display_Preserve *p = STN_DISPLAY_PRESERVE_OF(map_slot);
20     return search_node == p->sym_node;
21 }
22
23 static void ___stn_display_preserve_destroy(Hash_Map_Slot *s)
24 {
25     Stn_Display_Preserve *p = STN_DISPLAY_PRESERVE_OF(s);
26     vector_clear(&p->stn_displays);
27     vector_clear(&p->func_dependencies);
28     free_mem(p);
29 }
30

```

```

31 static Hash_Map stn_display_map;
32
33 void stn_display_preserve_destroy()
34 {
35     hash_map_for_each_destroy(&stn_display_map,
36         __stn_display_preserve_destroy);
37 }
38
39 void stn_display_add_dependency(Stn_Display_Preserve *p,
40     Stn_Dependency *dep)
41 {
42     assert(dep->sym_node->scope_id <= p->sym_node->scope_id);
43     if (dep->is_var)
44         vector_append(&p->stn_displays, dep->sym_node);
45     if (dep->is_func)
46         vector_append(&p->func_dependencies, dep->sym_node);
47 }
48
49 Stn_Display_Preserve *stn_display_preserve_get(Symbol_Table_Node *n)
50 {
51     Hash_Map_Slot *s = hash_map_get(&stn_display_map, n, n->scope_id);
52     if (s)
53         return STN_DISPLAY_PRESERVE_OF(s);
54     return NULL;
55 }
56
57 static inline Stn_Dependency *stn_dependency_alloc(Symbol_Table_Node *n,
58     bool is_func)
59 {
60     Stn_Dependency *d = ALLOC_NEW(Stn_Dependency);
61     d->sym_node = n;
62     if (is_func) {
63         d->is_func = true;
64         d->is_var = false;
65     } else {
66         d->is_func = false;
67         d->is_var = true;
68     }
69     return d;
70 }
71
72 static inline void stn_dependency_destroy(Stn_Dependency *d)
73 {
74     free_mem(d);
75 }
76
77 static Int stn_dependency_search_comparator(Symbol_Table_Node *search_node,
78     Rb_Tree_Node *tree_node)
79 {
80     Uns tree_id = STN_DEPENDENCY_OF(tree_node)->sym_node->scope_id;
81     if (search_node->scope_id < tree_id)
82         return -1;
83     else if (search_node->scope_id > tree_id)
84         return 1;
85     return 0;
86 }
87
88 static bool stn_dependency_comparator(Rb_Tree_Node *search_node,
89     Rb_Tree_Node *tree_node)
90 {
91     return STN_DEPENDENCY_OF(search_node)->sym_node->scope_id <=
92         STN_DEPENDENCY_OF(tree_node)->sym_node->scope_id;
93 }
94
95 AST_VISITOR_STRUCT_BEGIN(Ast_Visitor_Dependency)
96     Symbol_Table_Node *curr_func_node;
97     Rb_Tree *dependencies;
98     Vector *curr_dependencies;
99     Vector stn_display_vector;
100     String prev_iden;
101     Symbol_Property next_property;
102 AST_VISITOR_STRUCT_END(Ast_Visitor_Dependency)
103
104 static inline Stn_Display_Preserve *stn_display_preserve_alloc(

```



```

105     Symbol_Table_Node *n, Ast_Visitor_Dependency *v,
106     Const_String func_name)
107 {
108     Stn_Display_Preserve *p = ALLOC_NEW(Stn_Display_Preserve);
109     assert(func_name);
110     p->func_name = func_name;
111     p->sym_node = n;
112     p->stn_displays = VECTOR_INIT_SIZE(4);
113     p->func_dependencies = VECTOR_INIT_SIZE(4);
114     vector_append(&v->stn_display_vector, p);
115     p->next_vector_idx = vector_size(&v->stn_display_vector);
116     return p;
117 }
118
119 static inline void dep_visitor_add_dependency(Ast_Visitor_Dependency *v,
120     Symbol_Table_Node *dependency, bool is_func)
121 {
122     while (dependency->type == SYMBOL_TABLE_NODE_INTERMEDIATE)
123         dependency = dependency->parent;
124
125     if (!is_func && dependency->type == SYMBOL_TABLE_NODE_REC &&
126         v->curr_func_node) {
127         dependency = v->curr_func_node;
128         while (dependency->parent->type != SYMBOL_TABLE_NODE_REC)
129             dependency = dependency->parent;
130     }
131     if (dependency->type != SYMBOL_TABLE_NODE_FUNC ||
132         dependency == v->curr_func_node)
133         return;
134
135     Stn_Dependency *tmp;
136     VECTOR_FOR_EACH_ENTRY(v->curr_dependencies, tmp) {
137         if (tmp->sym_node == dependency) {
138             if (is_func)
139                 tmp->is_func = true;
140             else
141                 tmp->is_var = true;
142             return;
143         }
144     }
145
146     DLOG("found dependency %U is func? %d\n", dependency->scope_id, is_func);
147
148     Stn_Dependency *d = stn_dependency_alloc(dependency, is_func);
149     vector_append(v->curr_dependencies, d);
150 }
151
152 static inline void ternary_action(Ast_Visitor_Dependency *v,
153     Ast_Node_Ternary *n)
154 {
155     n->lhs->accept_visitor(n->lhs, AST_VISITOR_OF(v));
156     n->mid->accept_visitor(n->mid, AST_VISITOR_OF(v));
157     n->rhs->accept_visitor(n->rhs, AST_VISITOR_OF(v));
158 }
159
160 static inline void binary_action(Ast_Visitor_Dependency *v,
161     Ast_Node_Binary *n)
162 {
163     n->lhs->accept_visitor(n->lhs, AST_VISITOR_OF(v));
164     n->rhs->accept_visitor(n->rhs, AST_VISITOR_OF(v));
165 }
166
167 static inline void unary_action(Ast_Visitor_Dependency *v,
168     Ast_Node_Unary *n)
169 {
170     n->expr->accept_visitor(n->expr, AST_VISITOR_OF(v));
171 }
172
173 ASTVF_BEGIN(AST_EXPR_LOR, Ast_Visitor_Dependency, v, Ast_Node_Binary, n)
174     binary_action(v, n);
175 ASTVF_END
176
177 ASTVF_BEGIN(AST_EXPR_LAND, Ast_Visitor_Dependency, v, Ast_Node_Binary, n)
178     binary_action(v, n);

```

```

179 ASTVF_END
180
181 ASTVF_BEGIN(AST_EXPR_EQ, Ast_Visitor_Dependency, v, Ast_Node_Binary, n)
182     binary_action(v, n);
183 ASTVF_END
184
185 ASTVF_BEGIN(AST_EXPR_NEQ, Ast_Visitor_Dependency, v, Ast_Node_Binary, n)
186     binary_action(v, n);
187 ASTVF_END
188
189 ASTVF_BEGIN(AST_EXPR_GT, Ast_Visitor_Dependency, v, Ast_Node_Binary, n)
190     binary_action(v, n);
191 ASTVF_END
192
193 ASTVF_BEGIN(AST_EXPR_LT, Ast_Visitor_Dependency, v, Ast_Node_Binary, n)
194     binary_action(v, n);
195 ASTVF_END
196
197 ASTVF_BEGIN(AST_EXPR_GTEQ, Ast_Visitor_Dependency, v, Ast_Node_Binary, n)
198     binary_action(v, n);
199 ASTVF_END
200
201 ASTVF_BEGIN(AST_EXPR_LTEQ, Ast_Visitor_Dependency, v, Ast_Node_Binary, n)
202     binary_action(v, n);
203 ASTVF_END
204
205 ASTVF_BEGIN(AST_EXPR_PLUS, Ast_Visitor_Dependency, v, Ast_Node_Binary, n)
206     binary_action(v, n);
207 ASTVF_END
208
209 ASTVF_BEGIN(AST_EXPR_MINUS, Ast_Visitor_Dependency, v, Ast_Node_Binary, n)
210     binary_action(v, n);
211 ASTVF_END
212
213 ASTVF_BEGIN(AST_EXPR_MUL, Ast_Visitor_Dependency, v, Ast_Node_Binary, n)
214     binary_action(v, n);
215 ASTVF_END
216
217 ASTVF_BEGIN(AST_EXPR_DIV, Ast_Visitor_Dependency, v, Ast_Node_Binary, n)
218     binary_action(v, n);
219 ASTVF_END
220
221 ASTVF_BEGIN(AST_EXPR_CAST, Ast_Visitor_Dependency, v, Ast_Node_Binary, n)
222     Symbol_Property saved_property = v->next_property;
223     v->next_property = SYMBOL_PROPERTY_TYPE_DEF;
224     n->lhs->accept_visitor(n->lhs, AST_VISITOR_OF(v));
225     v->next_property = saved_property;
226     n->rhs->accept_visitor(n->rhs, AST_VISITOR_OF(v));
227 ASTVF_END
228
229 ASTVF_BEGIN(AST_EXPR_LNOT, Ast_Visitor_Dependency, v, Ast_Node_Unary, n)
230     unary_action(v, n);
231 ASTVF_END
232
233 ASTVF_BEGIN(AST_EXPR_ABS, Ast_Visitor_Dependency, v, Ast_Node_Unary, n)
234     unary_action(v, n);
235 ASTVF_END
236
237 ASTVF_BEGIN(AST_EXPR_INT, Ast_Visitor_Dependency, v, Ast_Expr_Int, n)
238     (void)v; (void)n;
239 ASTVF_END
240
241 ASTVF_BEGIN(AST_EXPR_BOOL, Ast_Visitor_Dependency, v, Ast_Expr_Bool, n)
242     (void)v; (void)n;
243 ASTVF_END
244
245 ASTVF_BEGIN(AST_EXPR_NULL, Ast_Visitor_Dependency, v, Ast_Expr_Null, n)
246     (void)v; (void)n;
247 ASTVF_END
248
249 ASTVF_BEGIN(AST_VARIABLE_IDEN, Ast_Visitor_Dependency, v, Ast_Variable_Iden, n)
250
251     Symbol *sym;
252     Symbol_Table_Node *sym_node = ast_node_get_symbol_table_node(

```

```

253     AST_NODE_OF(n));
254
255     switch (v->next_property) {
256     case SYMBOL_PROPERTY_VAR:
257         sym = symbol_table_node_lookup(sym_node, n->iden,
258             SYMBOL_PROPERTY_VAR);
259         assert(sym);
260         dep_visitor_add_dependency(v, sym->sym_node, false);
261         break;
262     case SYMBOL_PROPERTY_FUNC:
263         sym = symbol_table_node_lookup(sym_node, n->iden,
264             SYMBOL_PROPERTY_FUNC);
265         assert(sym);
266         dep_visitor_add_dependency(v, sym->sym_node, true);
267         break;
268
269     case SYMBOL_PROPERTY_TYPE_DEF:
270         break;
271     }
272     v->prev_iden = n->iden;
273 ASTVF_END
274
275 ASTVF_BEGIN(AST_EXPR_ARY_REF, Ast_Visitor_Dependency, v, Ast_Node_Binary, n)
276     binary_action(v, n);
277 ASTVF_END
278
279 ASTVF_BEGIN(AST_EXPR_FUNC_CALL, Ast_Visitor_Dependency, v,
280     Ast_Expr_Func_Call, n)
281     Symbol_Property saved_property = v->next_property;
282     v->next_property = SYMBOL_PROPERTY_FUNC;
283     n->identifier->accept_visitor(n->identifier, AST_VISITOR_OF(v));
284     v->next_property = saved_property;
285
286     Vector *args = n->arguments;
287     Ast_Node *a;
288     VECTOR_FOR_EACH_ENTRY(args, a)
289         a->accept_visitor(a, AST_VISITOR_OF(v));
290 ASTVF_END
291
292 ASTVF_BEGIN(AST_EXPR_DOT_REF, Ast_Visitor_Dependency, v, Ast_Node_Binary, n)
293     n->lhs->accept_visitor(n->lhs, AST_VISITOR_OF(v));
294 ASTVF_END
295
296 ASTVF_BEGIN(AST_EXPR_DIRECT_REF, Ast_Visitor_Dependency, v, Ast_Node_Binary, n)
297     if (n->lhs)
298         n->lhs->accept_visitor(n->lhs, AST_VISITOR_OF(v));
299 ASTVF_END
300
301 ASTVF_BEGIN(AST_SIMPLE_TYPE_INT, Ast_Visitor_Dependency, v, Ast_Empty, n)
302     (void)n; (void)v;
303 ASTVF_END
304
305 ASTVF_BEGIN(AST_SIMPLE_TYPE_BOOL, Ast_Visitor_Dependency, v, Ast_Empty, n)
306     (void)v; (void)n;
307 ASTVF_END
308
309 ASTVF_BEGIN(AST_TYPE_IDEN, Ast_Visitor_Dependency, v, Ast_Type_Iden, n)
310     (void)v; (void)n;
311 ASTVF_END
312
313 ASTVF_BEGIN(AST_VAR_DECL, Ast_Visitor_Dependency, v, Ast_Node_Binary, n)
314     #if 0
315         Symbol_Property saved_property = v->next_property;
316         v->next_property = SYMBOL_PROPERTY_VAR;
317         n->lhs->accept_visitor(n->lhs, AST_VISITOR_OF(v));
318         v->next_property = saved_property;
319     #endif
320     n->rhs->accept_visitor(n->rhs, AST_VISITOR_OF(v));
321 ASTVF_END
322
323 ASTVF_BEGIN(AST_TYPE_DEF, Ast_Visitor_Dependency, v, Ast_Node_Binary, n)
324     #if 0
325         Symbol_Property saved_property = v->next_property;
326         v->next_property = SYMBOL_PROPERTY_TYPE_DEF;

```

```

327     n->lhs->accept_visitor(n->lhs, AST_VISITOR_OF(v));
328     v->next_property = saved_property;
329 #endif
330     n->rhs->accept_visitor(n->rhs, AST_VISITOR_OF(v));
331 ASTVF_END
332
333 ASTVF_BEGIN(AST_TYPE_ARY, Ast_Visitor_Dependency, v, Ast_Type, n)
334     (void)v;
335     n->type->accept_visitor(n->type, AST_VISITOR_OF(v));
336 ASTVF_END
337
338 ASTVF_BEGIN(AST_TYPE_REC, Ast_Visitor_Dependency, v, Ast_Type_Rec, n)
339     Vector *vec;
340     Ast_Node *node;
341     vec = n->extend_list;
342     VECTOR_FOR_EACH_ENTRY(vec, node)
343         node->accept_visitor(node, AST_VISITOR_OF(v));
344     vec = n->body;
345     VECTOR_FOR_EACH_ENTRY(vec, node)
346         node->accept_visitor(node, AST_VISITOR_OF(v));
347 ASTVF_END
348
349 static void stmt_list_action(Ast_Visitor_Dependency *v, Ast_Stmt_List *n)
350 {
351     Ast_Node *s;
352     Vector *stmts = n->statements;
353     VECTOR_FOR_EACH_ENTRY(stmts, s)
354         s->accept_visitor(s, AST_VISITOR_OF(v));
355 }
356
357 ASTVF_BEGIN(AST_STMT_LIST, Ast_Visitor_Dependency, v, Ast_Stmt_List, n)
358     stmt_list_action(v, n);
359 ASTVF_END
360
361 ASTVF_BEGIN(AST_FIN_STMT_LIST, Ast_Visitor_Dependency, v, Ast_Stmt_List, n)
362     stmt_list_action(v, n);
363 ASTVF_END
364
365 static void func_dependency(Ast_Visitor_Dependency *v, Ast_Func_Def *n)
366 {
367     n->return_type->accept_visitor(n->return_type, AST_VISITOR_OF(v));
368
369     Symbol_Property saved_property = v->next_property;
370     v->next_property = SYMBOL_PROPERTY_FUNC;
371     n->iden->accept_visitor(n->iden, AST_VISITOR_OF(v));
372     v->next_property = saved_property;
373     String func_iden;
374     func_iden = v->prev_iden;
375
376     Symbol_Table_Node *func_stn = ast_node_get_symbol_table_node(
377         n->statements);
378
379     Symbol *sym = symbol_table_node_lookup(
380         ast_node_get_symbol_table_node(AST_NODE_OF(n)),
381         func_iden, SYMBOL_PROPERTY_FUNC);
382     assert(sym);
383
384     Stn_Display_Preserve *p = stn_display_preserve_alloc(func_stn, v,
385         sym->unique_name);
386
387     hash_map_insert(&stn_display_map, &p->hash_slot, func_stn->scope_id);
388
389     Vector *prev_curr_dep = v->curr_dependencies;
390     v->curr_dependencies = vector_alloc_size(4);
391
392     Vector *vec;
393     Ast_Node *node;
394     vec = n->parameters;
395     VECTOR_FOR_EACH_ENTRY(vec, node)
396         node->accept_visitor(node, AST_VISITOR_OF(v));
397     n->statements->accept_visitor(n->statements, AST_VISITOR_OF(v));
398
399     DLOG("func %S has scope: %U\n", func_iden, func_stn->scope_id);
400

```

```

401     Stn_Dependency *tmp;
402     VECTOR_FOR_EACH_ENTRY(v->curr_dependencies, tmp) {
403         if (tmp->sym_node == func_stn)
404             continue;
405         if (tmp->is_func) {
406             Rb_Tree_Node *rbn =
407                 rb_tree_search(v->dependencies, (Rb_Tree_Search_Comparator)
408                     stn_dependency_search_comparator, tmp->sym_node);
409             if (!rbn) {
410                 rb_tree_insert(v->dependencies, &tmp->rnode);
411             } else {
412                 Stn_Dependency *curr = STN_DEPENDENCY_OF(rbn);
413                 curr->is_func = true;
414             }
415         }
416     }
417
418     void stn_dependency_callback(Rb_Tree_Node *n)
419     {
420         Stn_Dependency *d = STN_DEPENDENCY_OF(n);
421         DLOG("add dependency with %U is func? %d\n", d->sym_node->scope_id,
422             (int)d->is_func);
423         stn_display_add_dependency(p, d);
424     }
425     rb_tree_for_each(v->dependencies, stn_dependency_callback);
426
427     Rb_Tree_Node *rbn = rb_tree_search_remove(v->dependencies,
428         (Rb_Tree_Search_Comparator)stn_dependency_search_comparator,
429         func_stn);
430
431     VECTOR_FOR_EACH_ENTRY(v->curr_dependencies, tmp) {
432         if (tmp->sym_node == func_stn) {
433             if (rbn && tmp != STN_DEPENDENCY_OF(rbn))
434                 stn_dependency_destroy(tmp);
435             continue;
436         }
437         Rb_Tree_Node *tn = rb_tree_search(v->dependencies,
438             (Rb_Tree_Search_Comparator)stn_dependency_search_comparator,
439             tmp->sym_node);
440         if (tn) {
441             Stn_Dependency *dup = STN_DEPENDENCY_OF(tn);
442             if (!dup->is_func)
443                 dup->is_func = tmp->is_func;
444             if (!dup->is_var)
445                 dup->is_var = tmp->is_var;
446
447             if (tmp != STN_DEPENDENCY_OF(tn))
448                 stn_dependency_destroy(tmp);
449         } else {
450             rb_tree_insert(v->dependencies, &tmp->rnode);
451         }
452     }
453
454     if (rbn) {
455         Stn_Dependency *d = STN_DEPENDENCY_OF(rbn);
456         stn_dependency_destroy(d);
457     }
458
459     vector_destroy(v->curr_dependencies, NULL);
460     v->curr_dependencies = prev_curr_dep;
461 }
462
463 ASTVF_BEGIN(AST_EXT_FUNC_DECL, Ast_Visitor_Dependency, v, Ast_Func_Def, n)
464     //n->extern_type->accept_visitor(n->extern_type, AST_VISITOR_OF(v));
465     //n->iden->accept_visitor(n->iden, AST_VISITOR_OF(v));
466     n->return_type->accept_visitor(n->return_type, AST_VISITOR_OF(v));
467     Vector *vec = n->parameters;
468     Ast_Node *p;
469     VECTOR_FOR_EACH_ENTRY(vec, p)
470         p->accept_visitor(p, AST_VISITOR_OF(v));
471 ASTVF_END
472
473 ASTVF_BEGIN(AST_FIN_FUNC_DEF, Ast_Visitor_Dependency, v, Ast_Func_Def, n)
474     func_dependency(v, n);

```

```

475 ASTVF_END
476
477 ASTVF_BEGIN(AST_REC_FUNC_DEF, Ast_Visitor_Dependency, v, Ast_Func_Def, n)
478     func_dependency(v, n);
479 ASTVF_END
480
481 ASTVF_BEGIN(AST_FUNC_DEF, Ast_Visitor_Dependency, v, Ast_Func_Def, n)
482     func_dependency(v, n);
483 ASTVF_END
484
485 ASTVF_BEGIN(AST_IF_STMT, Ast_Visitor_Dependency, v, Ast_Node_Binary, n)
486     binary_action(v, n);
487 ASTVF_END
488
489 ASTVF_BEGIN(AST_IF_ELSE_STMT, Ast_Visitor_Dependency, v, Ast_Node_Ternary, n)
490     ternary_action(v, n);
491 ASTVF_END
492
493 ASTVF_BEGIN(AST_ALLOC_ARY, Ast_Visitor_Dependency, v, Ast_Node_Binary, n)
494     binary_action(v, n);
495 ASTVF_END
496
497 ASTVF_BEGIN(AST_ALLOC_REC, Ast_Visitor_Dependency, v, Ast_Node_Unary, n)
498     unary_action(v, n);
499 ASTVF_END
500
501 ASTVF_BEGIN(AST_ALLOC_REC_CALL, Ast_Visitor_Dependency, v, Ast_Node_Binary, n)
502     //binary_action(v, n);
503     n->lhs->accept_visitor(n->lhs, AST_VISITOR_OF(v));
504 ASTVF_END
505
506 ASTVF_BEGIN(AST_DELETE, Ast_Visitor_Dependency, v, Ast_Node_Unary, n)
507     unary_action(v, n);
508 ASTVF_END
509
510 ASTVF_BEGIN(AST_WHILE_STMT, Ast_Visitor_Dependency, v, Ast_Node_Binary, n)
511     binary_action(v, n);
512 ASTVF_END
513
514 ASTVF_BEGIN(AST_RETURN_STMT, Ast_Visitor_Dependency, v, Ast_Node_Unary, n)
515     unary_action(v, n);
516 ASTVF_END
517
518 ASTVF_BEGIN(AST_WRITE_STMT, Ast_Visitor_Dependency, v, Ast_Node_Unary, n)
519     unary_action(v, n);
520 ASTVF_END
521
522 ASTVF_BEGIN(AST_ASSIGNMENT, Ast_Visitor_Dependency, v, Ast_Node_Binary, n)
523     binary_action(v, n);
524 ASTVF_END
525
526 ASTVF_BEGIN(AST_FUNC_BODY, Ast_Visitor_Dependency, v, Ast_Stmt_List, n)
527     Vector *stmts;
528     Ast_Node *s;
529
530     Symbol_Table_Node *prev_n = v->curr_func_node;
531     v->curr_func_node = ast_node_get_symbol_table_node(AST_NODE_OF(n));
532     stmts = n->statements;
533
534     Rb_Tree *dep = v->dependencies;
535
536     VECTOR(depv);
537     void stn_dependency_append(Rb_Tree_Node *n)
538     {
539         vector_append(&depv, STN_DEPENDENCY_OF(n));
540     }
541
542     VECTOR_FOR_EACH_ENTRY(stmts, s) {
543         v->dependencies = rb_tree_alloc(stn_dependency_comparator);
544         s->accept_visitor(s, AST_VISITOR_OF(v));
545         rb_tree_for_each(v->dependencies, stn_dependency_append);
546         rb_tree_destroy(v->dependencies, NULL);
547     }
548

```

```

549     Stn_Dependency *d;
550     VECTOR_FOR_EACH_ENTRY(&depv, d) {
551         if (!rb_tree_contains(dep,
552             (Rb_Tree_Search_Comparator) stn_dependency_search_comparator,
553             d->sym_node))
554             rb_tree_insert(dep, &d->rbnode);
555         else
556             stn_dependency_destroy(d);
557     }
558
559     vector_clear(&depv);
560
561     v->dependencies = dep;
562     v->curr_func_node = prev_n;
563 ASTVF_END
564
565 ASTVF_BEGIN(AST_EXPR_CHAR, Ast_Visitor_Dependency, v, Ast_Expr_Char, n)
566     (void)v; (void)n;
567 ASTVF_END
568
569 ASTVF_BEGIN(AST_EXPR_STRING, Ast_Visitor_Dependency, v, Ast_Expr_String, n)
570     (void)v; (void)n;
571 ASTVF_END
572
573 ASTVF_BEGIN(AST_IMPORT_STRING, Ast_Visitor_Dependency, v, Ast_Module_String, n)
574     (void)v; (void)n;
575 ASTVF_END
576
577 ASTVF_BEGIN(AST_PACKAGE_STRING, Ast_Visitor_Dependency, v, Ast_Module_String, n)
578     (void)v; (void)n;
579 ASTVF_END
580
581 ASTVF_BEGIN(AST_SIMPLE_TYPE_CHAR, Ast_Visitor_Dependency, v, Ast_Empty, n)
582     (void)v; (void)n;
583 ASTVF_END
584
585 ASTVF_BEGIN(AST_SIMPLE_TYPE_STRING, Ast_Visitor_Dependency, v, Ast_Empty, n)
586     (void)v; (void)n;
587 ASTVF_END
588
589 ASTVF_BEGIN(AST_SIMPLE_TYPE_VOID, Ast_Visitor_Dependency, v, Ast_Empty, n)
590     (void)v; (void)n;
591 ASTVF_END
592
593 ASTVF_BEGIN(AST_REC_SELF_PTR, Ast_Visitor_Dependency, v, Ast_Empty, n)
594     (void)v; (void)n;
595 ASTVF_END
596
597 static Ast_Visitor_Dependency dep_visitor = {
598     .AST_VISITOR_FIELD = AST_VISITOR_STATIC_INIT(),
599     .next_property = SYMBOL_PROPERTY_VAR,
600     .stn_display_vector = VECTOR_STATIC_INIT()
601 };
602
603 static int symbol_table_node_comp(const Symbol_Table_Node *search_node,
604     const Symbol_Table_Node *vec_node)
605 {
606     if (search_node->scope_id < vec_node->scope_id)
607         return 1;
608     if (search_node->scope_id > vec_node->scope_id)
609         return -1;
610     return 0;
611 }
612
613 void ast_visitor_dependency_accept(Ast *ast)
614 {
615     if (!ast_is_valid(ast))
616         return;
617     dep_visitor.dependencies = rb_tree_alloc(stn_dependency_comparator);
618     stn_display_map = HASH_MAP_INIT_SIZE(HASH_MAP_SIZE_23,
619         (Hash_Map_Comparator) stn_display_preserve_comparator);
620     ast->root->accept_visitor(ast->root, AST_VISITOR_OF(&dep_visitor));
621     assert(dep_visitor.next_property == SYMBOL_PROPERTY_VAR);
622

```

```

623     Stn_Display_Preserve *p;
624     VECTOR_FOR_EACH_ENTRY(&dep_visitor.stn_display_vector, p) {
625         DLOG("find func dependency for %S\n", p->func_name);
626         Symbol_Table_Node *node;
627         VECTOR_FOR_EACH_ENTRY(&p->func_dependencies, node) {
628             DLOG("func dependency for node %U\n", node->scope_id);
629             Stn_Display_Preserve *tmp = stn_display_preserve_get(node);
630             Symbol_Table_Node *tmp_node;
631             VECTOR_FOR_EACH_ENTRY(&tmp->stn_displays, tmp_node) {
632                 if (vector_contains_ptr(&p->stn_displays, tmp_node))
633                     continue;
634                 vector_append(&p->stn_displays, tmp_node);
635             }
636         }
637         vector_sort(&p->stn_displays,
638                     (Vector_Comparator) symbol_table_node_comp);
639     }
640
641     DEBUG(
642         DLOG("\n");
643         Stn_Display_Preserve *p;
644         VECTOR_FOR_EACH_ENTRY(&dep_visitor.stn_display_vector, p) {
645             DLOG("Scope %U preserves:\n", p->sym_node->scope_id);
646             Symbol_Table_Node *node;
647             VECTOR_FOR_EACH_ENTRY(&p->stn_displays, node) {
648                 DLOG("\tdisplay for: %U\n", node->scope_id);
649             }
650         }
651         DLOG("\n");
652     );
653     DEBUGT(def,
654         Stn_Display_Preserve *p;
655         VECTOR_FOR_EACH_ENTRY(&dep_visitor.stn_display_vector, p) {
656             Uns i = UNSIGNED_MAX;
657             Symbol_Table_Node *node;
658             VECTOR_FOR_EACH_ENTRY(&p->stn_displays, node) {
659                 if (i > node->scope_id)
660                     i = node->scope_id;
661                 else
662                     fatal_error(S("AST visitor dependency detected "
663                                   "unexpected scope dependency\n"));
664             }
665         }
666     );
667     rb_tree_destroy(dep_visitor.dependencies, NULL);
668     vector_clear(&dep_visitor.stn_display_vector);
669 }

```

:

A.6.6 src/ast/ast_visitor_dependency.h

```

1  #ifndef AST_VISITOR_DEPENDENCY_H
2  #define AST_VISITOR_DEPENDENCY_H
3
4  #include "ast_visitor.h"
5  #include <hash_map.h>
6  #include <rb_tree.h>
7
8  typedef struct Stn_Display_Preserve {
9      Const_String func_name;
10     Symbol_Table_Node *sym_node;
11     /* Symbol table node (function) displays to preserve
12      * when function with symbol table node sym_node is called. */
13     Vector stn_displays;
14     Vector func_dependencies;
15     Uns next_vector_idx;
16     Hash_Map_Slot hash_slot;
17 } Stn_Display_Preserve;
18
19 #define STN_DISPLAY_PRESERVE_OF(slot) \

```



```

20     HASH_MAP_ENTRY(slot, Stn_Display_Preserve, hash_slot)
21
22     /* Get stn dependencies for function with symbol table node n.
23      * Should always return != NULL if n is a symbol table node of a function. */
24     Stn_Display_Preserve *stn_display_preserve_get(Symbol_Table_Node *n);
25
26     void stn_display_preserve_destroy();
27
28     void ast_visitor_dependency_accept(Ast *ast);
29
30 #endif // AST_VISITOR_DEPENDENCY_H

```

A.7 Intermediate Representation

:

A.7.1 src/aia/aia.c

```

1  #include "aia.h"
2  #include <main.h>
3
4  Aia_Section *__aia_get_section(Aia *aia,
5                               Aia_Section_Type sec)
6  {
7      Aia_Section *s;
8
9      if (aia->sections[sec])
10         return aia->sections[sec];
11
12     s = ALLOC_NEW(Aia_Section);
13     aia->sections[sec] = s;
14     __aia_section_init(s, sec, aia);
15     __aia_set_section(aia, sec, s);
16     String lbl_name = aia_tmp_name_gen(aia, S("SECT"));
17     __aia_insert_jump_label_instr(aia, lbl_name);
18     string_destroy(lbl_name);
19
20     return NULL;
21 }
22
23 Aia_Func_Trampoline *__aia_func_trampoline_alloc(Aia *aia,
24                                                  Const_String tramp_name, Vector *params, Aia_Func *func)
25 {
26     Aia_Func_Trampoline *t = ALLOC_NEW(Aia_Func_Trampoline);
27     t->trampoline_name = string_duplicate(tramp_name);
28     t->func_name = func->func_name;
29     t->func_params = params;
30     t->block = __aia_block_alloc_entry(
31         aia->sections[aia->curr_sec], NULL);
32     t->blist = DOUBLE_LIST_INIT(t->blist);
33     double_list_append(&t->blist, &t->block->blist_node);
34     vector_append(&func->trampolines, t);
35     return t;
36 }
37
38 void __aia_func_destroy(Aia_Func *f)
39 {
40     string_destroy(f->func_name);
41     vector_for_each_destroy(&f->parameters,
42         (Vector_Destructor)string_destroy);
43     vector_for_each_destroy(&f->locals,
44         (Vector_Destructor)string_destroy);
45     vector_for_each_destroy(&f->trampolines,
46         (Vector_Destructor)__aia_func_trampoline_destroy);
47
48     vector_clear(&f->preserve_display_indices);
49
50     //__aia_block_for_each_sucessor_destroy(f->entry_block);

```

```

51     double_list_for_each_destroy(&f->blist, __aia_block_lone_db_destroy);
52     free_mem(f);
53 }
54
55 void __aia_func_hash_destroy(Hash_Map_Slot *s)
56 {
57     __aia_func_destroy(AIA_FUNC_OF(s));
58 }
59
60 Aia_Func *__aia_func_alloc(Aia *aia, Const_String func_name)
61 {
62     Aia_Func *f = ALLOC_NEW(Aia_Func);
63     f->parameters = VECTOR_INIT_SIZE(4);
64     f->locals = VECTOR_INIT_SIZE(4);
65     f->preserve_display_indices = VECTOR_INIT_SIZE(2);
66     f->func_name = string_duplicate(func_name);
67     f->trampolines = VECTOR_INIT_SIZE(2);
68     Aia_Block *entry = __aia_block_alloc_entry(
69         aia->sections[aia->curr_sec], f);
70     f->entry_block = f->exit_block = entry;
71     f->parent_func = NULL;
72     f->num_display_params = 0;
73     f->blist = DOUBLE_LIST_INIT(f->blist);
74     double_list_append(&f->blist, &f->entry_block->blist_node);
75     return f;
76 }
77
78 Aia_Func *__aia_get_func(Aia *aia, Const_String func_name)
79 {
80     Aia_Func *f;
81     Hash_Map_Slot *s;
82     Uns hash = string_hash_code(func_name);
83
84     s = hash_map_get(&aia->sections[aia->curr_sec]->functions,
85         (String)func_name, hash);
86     if (s)
87         return AIA_FUNC_OF(s);
88
89     f = __aia_func_alloc(aia, func_name);
90     hash_map_insert(&aia->sections[aia->curr_sec]->functions,
91         &f->hash_slot, hash);
92     __aia_set_func(aia, f);
93
94     /* Size of function is specified later. */
95     __aia_insert_label_instr(aia, func_name, 0, 4, AIA_LINKAGE_PRIVATE,
96         AIA_LABEL_TYPE_FUNC, 0, aia_get_null_location(aia));
97
98     return NULL;
99 }
100
101 void __aia_block_for_each_depth2_visit(Aia_Block *b,
102     Aia_Block_Callback start_callback,
103     Aia_Block_Callback ret_callback,
104     void *arg)
105 {
106     if (b->visit_count++)
107         goto out;
108
109     start_callback(b, arg);
110
111     Aia_Block *suc;
112     AIA_BLOCK_FOR_EACH_SUCESSOR(b, suc)
113         __aia_block_for_each_depth2_visit(suc, start_callback,
114             ret_callback, arg);
115
116     if (ret_callback)
117         ret_callback(b, arg);
118
119 out:;
120 }
121
122 void __aia_block_for_each_depth_visit(Aia_Block *b,
123     Aia_Block_Callback callback, void *arg)
124 {

```

```

125     __aia_block_for_each_depth2_visit(b, callback, NULL, arg);
126 }
127
128 Aia_Block *aia_block_twin(Aia_Block *b, Aia *aia)
129 {
130     Aia_Block *twin_b = __aia_block_alloc(b->section, b->function);
131     Aia_Instr *in;
132     AIA_BLOCK_FOR_EACH_INSTRUCTION(b, in) {
133         Aia_Instr *twin_in = aia_instr_twin(in, twin_b);
134         __aia_block_append_instr(twin_b, twin_in);
135     }
136     Aia_Instr *lbl_in = __aia_block_peek_first_instr(twin_b);
137     Aia_Operand *lbl_op = aia_instr_get_dest_op(lbl_in);
138
139     String lbl_name = aia_tmp_num_append(aia,
140         aia_operand_label_get_name(lbl_op));
141
142     Aia_Operand *new_lbl_op = __aia_operand_label_alloc(aia, lbl_name,
143         aia_operand_label_get_offset(lbl_op), AIA_OPERAND_LABEL);
144
145     aia_instr_replace_op(lbl_in, -1, new_lbl_op);
146
147     return twin_b;
148 }
149
150 static inline Uns __aia_block_sucessor_count_tmp_jump(Aia_Block *b)
151 {
152     Double_List_Node *ln = double_list_peek_last(&b->instructions);
153     switch (AIA_INSTR_OF(ln)->type) {
154     AIA_CASE_COND_TMP_JUMP:
155         return 2;
156     case __AIA_JUMP:
157         return 1;
158     default:
159         return 0;
160     }
161 }
162
163 static inline Aia_Block *__aia_label_to_block_tmp_jump(Aia *aia,
164     Aia_Operand *lbl)
165 {
166     assert(lbl);
167     return aia_label_to_instruction(aia, lbl)->containing_block;
168 }
169
170 Aia_Instr *__aia_block_peek_first_instr(Aia_Block *b)
171 {
172     Double_List_Node *n = double_list_peek_first(&b->instructions);
173     if (!n)
174         return NULL;
175     return AIA_INSTR_OF(n);
176 }
177
178 Aia_Instr *__aia_block_peek_last_instr(Aia_Block *b)
179 {
180     Double_List_Node *n = double_list_peek_last(&b->instructions);
181     if (!n)
182         return NULL;
183     return AIA_INSTR_OF(n);
184 }
185
186 #define AIA_BLOCK_FOR_EACH_SUCESSOR_TMP_JUMP(block, sucessor, aia) \
187     for (Double_List_Node *__n = \
188         double_list_peek_last(&(block)->instructions); \
189         __n; \
190         __n = NULL) \
191     for (Uns __s = 0, \
192         __c = __aia_block_sucessor_count_tmp_jump(b); \
193         __s < __c; \
194         __s++) \
195         if ((sucessor = __aia_label_to_block_tmp_jump(aia, \
196             aia_instr_get_src_op(AIA_INSTR_OF(__n), __s))) || \
197             !sucessor)
198

```

```

199 static void __aia_block_set_predecessors_start(Aia_Block *b, Aia *aia)
200 {
201     if (b->visit_count)
202         return;
203
204     b->visit_count = 1;
205
206     Aia_Block *suc;
207     AIA_BLOCK_FOR_EACH_SUCESSOR_TMP_JUMP(b, suc, aia) {
208         __aia_block_append_predecessor(suc, b);
209         __aia_block_set_predecessors_start(suc, aia);
210     }
211 }
212
213 static void __aia_block_set_predecessors_end(Aia_Block *b, Aia *aia)
214 {
215     if (!b->visit_count)
216         return;
217
218     b->visit_count = 0;
219
220     Aia_Block *suc;
221     AIA_BLOCK_FOR_EACH_SUCESSOR_TMP_JUMP(b, suc, aia)
222         __aia_block_set_predecessors_end(suc, aia);
223 }
224
225 static inline void __aia_block_replace_jump(Aia_Block *b, Aia *aia)
226 {
227     Aia_Block *block;
228     Aia_Operand *old_lbl;
229     Uns op_count = __aia_block_sucessor_count_tmp_jump(b);
230
231     if (!op_count) // Exit block or .ret instruction
232         return;
233
234     Double_List_Node *n = double_list_peek_last(&b->instructions);
235     Aia_Instr *jin = AIA_INSTR_OF(n);
236     switch (jin->type) {
237     AIA_CASE_COND_TMP_JUMP:
238         old_lbl = aia_instr_get_src_op(jin, 1);
239         assert(old_lbl->op_type == AIA_OPERAND_LABEL &&
240             !old_lbl->iden.op_label->offset);
241
242         block = __aia_label_to_block_tmp_jump(aia, old_lbl);
243         __aia_operand_release(old_lbl);
244         aia_instr_set_src_op(jin, 1, aia_operand_block_alloc(aia, block));
245         /* Fall through. */
246     case __AIA_JMP:
247         old_lbl = aia_instr_get_src_op(jin, 0);
248         assert(old_lbl->op_type == AIA_OPERAND_LABEL &&
249             !old_lbl->iden.op_label->offset);
250
251         block = __aia_label_to_block_tmp_jump(aia, old_lbl);
252         __aia_operand_release(old_lbl);
253         aia_instr_set_src_op(jin, 0, aia_operand_block_alloc(aia, block));
254         break;
255     default:
256         fatal_error(S("Unexpected branch instruction. Aborting...\n"));
257     }
258     jin->type = AIA_LABEL_JUMP_TO_BLOCK_JUMP(jin->type);
259 }
260
261 static void __aia_block_replace_jumps(Double_List *blist, Aia *aia)
262 {
263     Double_List_Node *bnode;
264     DOUBLE_LIST_FOR_EACH(blist, bnode)
265         __aia_block_replace_jump(AIA_BLOCK_OF_DBNODE(bnode), aia);
266 }
267
268 static void __aia_append_nop(Aia_Block *b)
269 {
270     Aia_Instr *tmp = __aia_block_peek_last_instr(b);
271     Aia_Instr *in = aia_instr_alloc_0op(AIA_NOP, b, -1,
272         aia_instr_get_location(tmp));

```

```

273     __aia_block_append_instr(b, in);
274 }
275
276 static void __aia_finish_blocks(Aia *aia)
277 {
278     Aia_Section *sec;
279     AIA_FOR_EACH_SECTION(aia, sec) {
280         Aia_Func *func;
281         __aia_block_set_predecessors_start(sec->entry_block, aia);
282         __aia_block_set_predecessors_end(sec->entry_block, aia);
283         __aia_block_replace_jumps(&sec->sec_blist, aia);
284         __aia_append_nop(sec->exit_block);
285
286         AIA_SECTION_FOR_EACH_FUNC(sec, func) {
287             __aia_block_set_predecessors_start(func->entry_block, aia);
288             __aia_block_set_predecessors_end(func->entry_block, aia);
289             __aia_block_replace_jumps(&func->blist, aia);
290             __aia_append_nop(func->exit_block);
291
292             Aia_Func_Trampoline *tramp;
293             AIA_FUNC_FOR_EACH_TRAMPOLINE(func, tramp) {
294                 __aia_block_replace_jump(tramp->block, aia);
295                 __aia_append_nop(tramp->block);
296             }
297         }
298     }
299 }
300
301 #if 0
302 /* Returns suc if it's removed, else return NULL. */
303 Aia_Block *__aia_block_remove_sucessor(Aia_Block *b, Aia_Block *suc)
304 {
305     Aia_Operand *block_op;
306     Aia_Instr *last_instr = __aia_block_peek_last_instr(b);
307     Aia_Block *ret = NULL;
308
309     switch (last_instr->type) {
310     AIA_CASE_COND_JUMP:
311         block_op = aia_instr_get_src_op(last_instr, 1);
312         if (block_op->iden.op_block == suc) {
313             ret = suc;
314             block_op->iden.op_block = NULL;
315             break;
316         }
317         /* Fall through. */
318
319     case AIA_JMP:
320         block_op = aia_instr_get_src_op(last_instr, 0);
321         if (block_op->iden.op_block == suc) {
322             ret = suc;
323             block_op->iden.op_block = NULL;
324         }
325         break;
326
327     default:
328         fatal_error(S("unable to find jump instruction in the end of a block. "
329                     "Aborting...\n"));
330     }
331     return ret;
332 }
333
334 /* Returns pred if it's removed, else return NULL. */
335 Aia_Block *__aia_block_remove_predecessor(Aia_Block *b, Aia_Block *pred)
336 {
337     if (vector_remove_ptr(&b->predecessors, pred))
338         return pred;
339     return NULL;
340 }
341
342 void __aia_block_remove_from_predecessors(Aia_Block *b)
343 {
344     Aia_Block *pred;
345     AIA_BLOCK_FOR_EACH_PREDECESSOR(b, pred) {
346         Aia_Block *ret = __aia_block_remove_sucessor(pred, b);

```

```

347     (void)ret;
348     assert(ret);
349 }
350 }
351
352 void __aia_block_remove_from_sucessors(Aia_Block *b)
353 {
354     Aia_Block *suc;
355     AIA_BLOCK_FOR_EACH_SUCESSOR(b, suc) {
356         Aia_Block *ret = __aia_block_remove_predecessor(suc, b);
357         (void)ret;
358         assert(ret);
359     }
360 }
361
362 void __aia_block_for_each_sucessor_destroy(Aia_Block *entry)
363 {
364     void recursive_destroy(Aia_Block *b)
365     {
366         __aia_block_remove_from_predecessors(b);
367
368         Aia_Block *suc;
369         AIA_BLOCK_FOR_EACH_SUCESSOR(b, suc)
370             recursive_destroy(suc);
371
372         __aia_block_lone_destroy(b);
373     }
374
375     recursive_destroy(entry);
376 }
377 #endif
378
379 void aia_block_remove_from_predecessors(Aia_Block *b)
380 {
381     Aia_Block *suc;
382     AIA_BLOCK_FOR_EACH_SUCESSOR(b, suc)
383         vector_remove_ptr(&suc->predecessors, b);
384 }
385
386 void aia_block_forget_predecessor(Aia_Block *b, Aia_Block *pred)
387 {
388     bool res = vector_remove_ptr(&b->predecessors, pred);
389     (void)res;
390     assert(res);
391 }
392
393 void aia_block_blist_remove_destroy(Aia_Block *b)
394 {
395     double_list_remove(&b->blist_node);
396     __aia_block_lone_destroy(b);
397 }
398
399 void __aia_block_lone_db_destroy(Double_List_Node *n)
400 {
401     __aia_block_lone_destroy(AIA_BLOCK_OF_DBNODE(n));
402 }
403
404 static void aia_insert_default_funcs(Aia *aia)
405 {
406     Aia_Func *func;
407     Uns hash;
408
409     func = __aia_func_alloc(aia, AIA_FUNC_WRITEINI);
410     hash = string_hash_code(aia_func_get_name(func));
411     hash_map_insert(&aia->default_funcs, &func->hash_slot, hash);
412
413     func = __aia_func_alloc(aia, AIA_FUNC_WRITEINB);
414     hash = string_hash_code(aia_func_get_name(func));
415     hash_map_insert(&aia->default_funcs, &func->hash_slot, hash);
416
417     func = __aia_func_alloc(aia, AIA_FUNC_WRITEINC);
418     hash = string_hash_code(aia_func_get_name(func));
419     hash_map_insert(&aia->default_funcs, &func->hash_slot, hash);
420

```

```

421     func = __aia_func_alloc(aia, AIA_FUNC_WRITE_LNS);
422     hash = string_hash_code(aia_func_get_name(func));
423     hash_map_insert(&aia->default_funcs, &func->hash_slot, hash);
424
425     func = __aia_func_alloc(aia, AIA_FUNC_WRITE_LNR);
426     hash = string_hash_code(aia_func_get_name(func));
427     hash_map_insert(&aia->default_funcs, &func->hash_slot, hash);
428
429     func = __aia_func_alloc(aia, AIA_FUNC_WRITE_LNA);
430     hash = string_hash_code(aia_func_get_name(func));
431     hash_map_insert(&aia->default_funcs, &func->hash_slot, hash);
432
433     func = __aia_func_alloc(aia, AIA_FUNC_WRITE_LNN);
434     hash = string_hash_code(aia_func_get_name(func));
435     hash_map_insert(&aia->default_funcs, &func->hash_slot, hash);
436
437     func = __aia_func_alloc(aia, AIA_FUNC_ALLOCATEAB);
438     hash = string_hash_code(aia_func_get_name(func));
439     hash_map_insert(&aia->default_funcs, &func->hash_slot, hash);
440
441     func = __aia_func_alloc(aia, AIA_FUNC_ALLOCATEAL);
442     hash = string_hash_code(aia_func_get_name(func));
443     hash_map_insert(&aia->default_funcs, &func->hash_slot, hash);
444
445     func = __aia_func_alloc(aia, AIA_FUNC_ALLOCATE);
446     hash = string_hash_code(aia_func_get_name(func));
447     hash_map_insert(&aia->default_funcs, &func->hash_slot, hash);
448
449     func = __aia_func_alloc(aia, AIA_FUNC_DELETEA);
450     hash = string_hash_code(aia_func_get_name(func));
451     hash_map_insert(&aia->default_funcs, &func->hash_slot, hash);
452
453     func = __aia_func_alloc(aia, AIA_FUNC_DELETE);
454     hash = string_hash_code(aia_func_get_name(func));
455     hash_map_insert(&aia->default_funcs, &func->hash_slot, hash);
456 }
457
458 void __aia_finish(Aia *aia)
459 {
460     __aia_set_exit_block(aia);
461     __aia_finish_blocks(aia);
462     aia_insert_default_funcs(aia);
463 }
464
465 Aia_Operand *aia_operand_addr_ref_alloc(Aia *aia UNUSED,
466     Aia_Operand *label,
467     Aia_Operand *disp,
468     Aia_Operand *base,
469     Aia_Operand *index,
470     Aia_Operand *scale)
471 {
472     Aia_Operand *op = ALLOC_NEW(Aia_Operand);
473     op->iden.addr_ref = ALLOC_NEW(Aia_Operand_Addr_Ref);
474
475     DEBUGT(def,
476         assert(base);
477         if (label)
478             assert(label->op_type == AIA_OPERAND_LABEL);
479         if (disp)
480             assert(disp->op_type == AIA_OPERAND_CONST_INT);
481         if (scale) {
482             assert(scale->op_type == AIA_OPERAND_CONST_INT);
483             switch (scale->iden.int_const) {
484                 case 1:
485                 case 2:
486                 case 4:
487                     break;
488                 default:
489                     assert(false);
490             }
491         }
492     );
493     if (label)

```

```

495     ++label->ref_count;
496     if (disp)
497         ++disp->ref_count;
498     ++base->ref_count;
499     if (index)
500         ++index->ref_count;
501     if (scale)
502         ++scale->ref_count;
503
504     *op->iden.addr_ref = (Aia_Operand_Addr_Ref){
505         label, disp, base, index, scale
506     };
507
508     op->op_type = AIA_OPERAND_ADDR_REF;
509     op->ref_count = 0;
510     return op;
511 }
512
513 void __aia_insert_label_instr(Aia *aia, Const_String lbl_name,
514     int32_t lbl_offset, uint8_t alignment, Aia_Linkage linkage,
515     Aia_Label_Type lbl_type, int32_t obj_byte_size,
516     File_Location *loc)
517 {
518     Aia_Instr *lbl_instr = __aia_2op_instr(aia, __AIA_LABEL,
519         AIA_LONG, AIA_LONG, loc);
520     Aia_Operand *lbl = aia_operand_label_alloc(aia, lbl_name, lbl_offset);
521
522     Aia_Label_Data d;
523     d.alignment = alignment;
524     d.linkage = linkage;
525     d.label_type = lbl_type;
526
527     Aia_Operand *icon = aia_operand_const_int_alloc(aia, d.data);
528     aia_instr_set_src_op(lbl_instr, 0, icon);
529
530     icon = aia_operand_const_int_alloc(aia, obj_byte_size);
531     aia_instr_set_src_op(lbl_instr, 1, icon);
532
533     aia_instr_set_dest_op(lbl_instr, lbl);
534
535     Aia_Label_Instr_Entry *e = ALLOC_NEW(Aia_Label_Instr_Entry);
536     e->label_instr = lbl_instr;
537     Uns hash = string_hash_code(lbl->iden.op_label->label_name);
538
539     assert(!hash_map_contains(&aia->label_instr_map, lbl, hash));
540     hash_map_insert(&aia->label_instr_map, &e->hash_slot, hash);
541 }
542
543 static CONST_STRING(aia_section_init_std, ".init");
544 static CONST_STRING(aia_section_init_lib, ".Vlt_libinit, \"xa\"");
545
546 Const_String __aia_section_names[AIA_SECTION_COUNT] = {
547     [AIA_SECTION_INIT] = NULL,
548     [AIA_SECTION_FINI] = S(".fini"),
549     [AIA_SECTION_TEXT] = S(".text"),
550     [AIA_SECTION_DATA] = S(".data"),
551     [AIA_SECTION_RODATA] = S(".rodata")
552 };
553
554 void __aia_func_trampoline_destroy(Aia_Func_Trampoline *t)
555 {
556     string_destroy(t->trampoline_name);
557     double_list_for_each_destroy(&t->blist, __aia_block_lone_db_destroy);
558     free_mem(t);
559 }
560
561 bool __aia_func_comparator(String search_func, Hash_Map_Slot *maps)
562 {
563     Aia_Func *f = AIA_FUNC_OF(maps);
564     return !string_compare(search_func, f->func_name);
565 }
566
567 Aia *aia_alloc(Const_String src_fname)
568 {

```



```

569     if (cmdopts.library_init)
570         __aia_section_names[AIA_SECTION_INIT] = aia_section_init_lib;
571     else
572         __aia_section_names[AIA_SECTION_INIT] = aia_section_init_std;
573
574     Aia *aia = ALLOC_NEW(Aia);
575     aia->label_instr_map = HASH_MAP_INIT(
576         (Hash_Map_Comparator)aia_label_instr_comparator);
577     aia->default_funcs = HASH_MAP_INIT_SIZE(HASH_MAP_SIZE_23,
578         (Hash_Map_Comparator) __aia_func_comparator);
579
580     for (Uns i = 0; i < AIA_SECTION_COUNT; i++)
581         aia->sections[i] = NULL;
582
583     aia->curr_func = NULL;
584     aia->curr_block = NULL;
585     aia->curr_blist = NULL;
586
587     aia->curr_tmp_reg_num = 0;
588     aia->curr_tmp_label_num = 0;
589
590     aia->aia_instr_dump_callback = NULL;
591
592     aia->source_file_name = string_duplicate(src_fname);
593     aia->source_null_file_loc =
594         FILE_LOCATION_INIT(aia->source_file_name, 0, 0);
595
596     aia->record_self_ptr = aia_operand_local_ref_alloc(aia,
597         SELF_STR/*, AIA_LONG*/);
598     __aia_operand_acquire(aia->record_self_ptr);
599
600     return aia;
601 }
602
603 Aia_Func *aia_func_lookup(Aia *aia, Const_String func_name)
604 {
605     Hash_Map_Slot *fslot;
606     Uns hash = string_hash_code(func_name);
607
608     fslot = hash_map_get(&aia->default_funcs, (String)func_name, hash);
609     if (fslot)
610         return AIA_FUNC_OF(fslot);
611
612     if (aia->sections[AIA_SECTION_TEXT]) {
613         fslot = hash_map_get(
614             &aia->sections[AIA_SECTION_TEXT]->functions,
615             (String)func_name,
616             hash);
617         if (fslot)
618             return AIA_FUNC_OF(fslot);
619     }
620
621     for (Uns i = 0; i < AIA_SECTION_TEXT; i++) {
622         if (aia->sections[i]) {
623             fslot = hash_map_get(
624                 &aia->sections[i]->functions,
625                 (String)func_name,
626                 hash);
627             if (fslot)
628                 return AIA_FUNC_OF(fslot);
629         }
630     }
631     for (Uns i = AIA_SECTION_TEXT + 1;
632          i < AIA_SECTION_COUNT; i++) {
633         if (aia->sections[i]) {
634             fslot = hash_map_get(
635                 &aia->sections[i]->functions,
636                 (String)func_name,
637                 hash);
638             if (fslot)
639                 return AIA_FUNC_OF(fslot);
640         }
641     }
642     return NULL;

```

```

643 }
644
645 static void dump_block(Aia_Block *b, Aia *aia)
646 {
647     FILE *stream = aia->meta_data;
648     file_print_message(stream, S("# AIA block start\n"));
649     Aia_Instr *in;
650     AIA_BLOCK_FOR_EACH_INSTRUCTION(b, in)
651     aia_instr_dump(stream, in, aia);
652     file_print_message(stream, S("# AIA block end\n"));
653 }
654
655 #if 0
656 static void aia_dump_blocks(Aia *aia,
657     Aia_Block *entry_b, Aia_Block *exit_b)
658 {
659     if (entry_b != exit_b)
660         aia_block_for_each_depth(entry_b, exit_b, dump_block, aia);
661     else
662         dump_block(entry_b, aia);
663 }
664 #endif
665
666 static void aia_dump_blocks(Aia *aia, Double_List *blist)
667 {
668     Double_List_Node *dbnode;
669     DOUBLE_LIST_FOR_EACH(blist, dbnode) {
670         Aia_Block *b = AIA_BLOCK_OF_DBNODE(dbnode);
671         dump_block(b, aia);
672     }
673 }
674
675 static void aia_dump_trampoline(Aia *aia, Aia_Func_trampoline *t)
676 {
677     FILE *stream = aia->meta_data;
678     file_print_message(stream, S("\n@trampoline(%S)\n"), t->trampoline_name);
679
680     Const_String str;
681     VECTOR_FOR_EACH_ENTRY(t->func_params, str)
682         file_print_message(stream, S("\t@param(%S)\n"), str);
683
684     aia_dump_blocks(aia, &t->blist);
685
686     file_print_message(stream, S(".size %l$, . - %l$\n"),
687         t->trampoline_name);
688     file_print_message(stream, S("# end %S\n\n"), t->trampoline_name);
689 }
690
691 static void aia_dump_func(Aia *aia, Aia_Func *f)
692 {
693     FILE *stream = aia->meta_data;
694     Aia_Func_trampoline *t;
695     VECTOR_FOR_EACH_ENTRY(&f->trampolines, t)
696         aia_dump_trampoline(aia, t);
697
698     file_print_message(stream, S("\n@procedure(%S)\n"), f->func_name);
699
700     Const_String str;
701     VECTOR_FOR_EACH_ENTRY(&f->parameters, str)
702         file_print_message(stream, S("\t@param(%S)\n"), str);
703
704     ssize_t display;
705     VECTOR_FOR_EACH_ENTRY(&f->preserve_display_indices, display)
706         file_print_message(stream, S("\t@push_display(%zd)\n"), display);
707
708     VECTOR_FOR_EACH_ENTRY(&f->locals, str)
709         file_print_message(stream, S("\t@var(%S)\n"), str);
710
711     aia_dump_blocks(aia, &f->blist);
712
713     file_print_message(stream, S(".size %l$, . - %l$\n"), f->func_name);
714     file_print_message(stream, S("# end %S\n\n"), f->func_name);
715 }
716

```

```

717 Aia_Instr *aia_private_label_before(Aia *aia, Aia_Operand *lbl,
718     Aia_Instr *sucessor)
719 {
720     Aia_Instr *lbl_instr = aia_instr_alloc_2op(__AIA_LABEL,
721         aia_instr_get_block(sucessor), AIA_LONG, AIA_LONG,
722         aia_instr_get_location(sucessor));
723
724     Aia_Operand *icon = aia_operand_const_int_alloc(aia, 0);
725     aia_instr_set_src_op(lbl_instr, 0, icon);
726     icon = aia_operand_const_int_alloc(aia, AIA_LINKAGE_PRIVATE);
727     aia_instr_set_src_op(lbl_instr, 1, icon);
728     aia_instr_set_dest_op(lbl_instr, lbl);
729
730     Aia_Label_Instr_Entry *e = ALLOC_NEW(Aia_Label_Instr_Entry);
731     e->label_instr = lbl_instr;
732     Uns hash = string_hash_code(lbl->iden.op_label->label_name);
733
734     assert(!hash_map_contains(&aia->label_instr_map, lbl, hash));
735     hash_map_insert(&aia->label_instr_map, &e->hash_slot, hash);
736
737     aia_instr_insert_before(lbl_instr, sucessor);
738
739     return lbl_instr;
740 }
741
742 Aia_Instr *aia_type_mov_before(uint16_t mov_instr, Aia_Operand *src_op,
743     Aia_Operand *dest_op, Aia_Instr *sucessor,
744     uint8_t src_size, uint8_t dest_size)
745 {
746     DEBUGT(def,
747         switch (mov_instr) {
748             case AIA_MOV:
749             case AIA_MOVS:
750             case AIA_MOVZ:
751                 break;
752             default:
753                 assert(false);
754         }
755     );
756
757     Aia_Instr *mov_in = aia_instr_alloc_lop(mov_instr,
758         aia_instr_get_block(sucessor), dest_size, src_size,
759         aia_instr_get_location(sucessor));
760     aia_instr_set_src_op(mov_in, 0, src_op);
761     aia_instr_set_dest_op(mov_in, dest_op);
762
763     aia_instr_insert_before(mov_in, sucessor);
764
765     return mov_in;
766 }
767
768 Aia_Instr *aia_mov_before(Aia_Operand *src_op, Aia_Operand *dest_op,
769     Aia_Instr *sucessor, uint8_t op_sizes)
770 {
771     Aia_Instr *mov_in = aia_instr_alloc_lop(AIA_MOV,
772         aia_instr_get_block(sucessor), op_sizes, op_sizes,
773         aia_instr_get_location(sucessor));
774     aia_instr_set_src_op(mov_in, 0, src_op);
775     aia_instr_set_dest_op(mov_in, dest_op);
776
777     aia_instr_insert_before(mov_in, sucessor);
778
779     return mov_in;
780 }
781
782 Aia_Instr *aia_mov_after(Aia_Operand *src_op, Aia_Operand *dest_op,
783     Aia_Instr *predecessor, uint8_t op_sizes)
784 {
785     Aia_Instr *mov_in = aia_instr_alloc_lop(AIA_MOV,
786         aia_instr_get_block(predecessor), op_sizes, op_sizes,
787         aia_instr_get_location(predecessor));
788     aia_instr_set_src_op(mov_in, 0, src_op);
789     aia_instr_set_dest_op(mov_in, dest_op);
790

```

```

791     aia_instr_insert_after(mov_in, predecessor);
792
793     return mov_in;
794 }
795
796 void aia_dump(Aia *aia, FILE *stream)
797 {
798     if (!aia_is_valid(aia))
799         return;
800
801     aia->meta_data = stream;
802
803     Aia_Section *sec;
804     AIA_FOR_EACH_SECTION(aia, sec) {
805         file_print_message(stream, S(".section %S\n"), sec->section_name);
806
807         aia_dump_blocks(aia, &sec->sec_blist);
808
809         Aia_Func *func;
810         AIA_SECTION_FOR_EACH_FUNC(sec, func)
811             aia_dump_func(aia, func);
812         file_print_message(stream, S("\n"));
813     }
814 }

```

:

A.7.2 src/aia/aia.h

```

1  #ifndef AIA_H
2  #define AIA_H
3
4  #include <std_include.h>
5  #include <hash_map.h>
6  #include <vector.h>
7  #include <double_list.h>
8  #include <debug.h>
9  #include "aia_operand.h"
10 #include "aia_instr.h"
11
12 #undef DEBUG_TYPE
13 #define DEBUG_TYPE aia-gen
14
15 #define VMT_CSTR ".vmt"
16 #define VMT_STR S(VMT_CSTR)
17
18 #define SELF_STR S(".slf")
19 #define DISP_CSTR_PREFIX ".disp."
20
21 #define AIA_FUNC_WRITEINI S("_Vit_writelni")
22 #define AIA_FUNC_WRITEINS S("_Vit_writelns")
23 #define AIA_FUNC_WRITEINC S("_Vit_writelnc")
24 #define AIA_FUNC_WRITEINB S("_Vit_writelnb")
25 #define AIA_FUNC_WRITEINR S("_Vit_writelnr")
26 #define AIA_FUNC_WRITEINA S("_Vit_writelna")
27 #define AIA_FUNC_WRITEINN S("_Vit_writelnn")
28 #define AIA_FUNC_ALLOCATEAB S("_Vit_allocab")
29 #define AIA_FUNC_ALLOCATEAL S("_Vit_allocal")
30 #define AIA_FUNC_ALLOCATE S("_Vit_alloc")
31 #define AIA_FUNC_DELETEA S("_Vit_deletea")
32 #define AIA_FUNC_DELETE S("_Vit_delete")
33
34 #define AIA_NUM_DEFAULT_FUNCS 8
35
36 typedef struct Aia_Section Aia_Section;
37
38 typedef struct Aia_Func Aia_Func;
39
40 typedef enum Aia_Linkage {
41     AIA_LINKAGE_PRIVATE,
42     AIA_LINKAGE_GLOBAL,

```

```

43     AIA_LINKAGE_WEAK
44 } Aia_Linkage;
45
46 typedef enum Aia_Label_Type {
47     AIA_LABEL_TYPE_NONE,
48     AIA_LABEL_TYPE_FUNC,
49     AIA_LABEL_TYPE_OBJ
50 } Aia_Label_Type;
51
52 typedef union Aia_Label_Data {
53     struct {
54         unsigned alignment    : 8;
55         Aia_Linkage linkage    : 8;
56         Aia_Label_Type label_type : 8;
57         unsigned              : 8;
58     };
59     uint32_t data;
60 } Aia_Label_Data;
61
62 typedef enum Aia_Section_Type {
63     AIA_SECTION_INIT,
64     AIA_SECTION_FINI,
65     AIA_SECTION_TEXT,
66     AIA_SECTION_DATA,
67     AIA_SECTION_RODATA,
68     AIA_SECTION_COUNT // Must be last (not really a section)
69 } Aia_Section_Type;
70
71 struct Aia_Block {
72     Vector predecessors;
73     Double_List instructions;
74     Aia_Section *section;
75     Aia_Func *function;
76     void *meta_data;
77     Uns visit_count;
78     Double_List_Node blist_node;
79 };
80
81 #define AIA_BLOCK_OF_DBNODE(node) \
82     DOUBLE_LIST_ENTRY(node, Aia_Block, blist_node)
83
84 typedef struct Aia_Func_Trampoline {
85     String trampoline_name;
86     Const_String func_name;
87     Vector *func_params;
88     Aia_Block *block;
89     void *meta_data;
90     Double_List blist;
91 } Aia_Func_Trampoline;
92
93 typedef struct Aia_Func {
94     String func_name;
95     File_Location location;
96     Vector preserve_display_indices;
97     Vector parameters;
98     Vector locals;
99     Vector trampolines;
100     Hash_Map_Slot hash_slot;
101     Aia_Block *entry_block;
102     Aia_Block *exit_block;
103     Aia_Func *parent_func;
104     void *meta_data;
105     void *func_access_struct;
106     void *func_kills_struct;
107     Double_List blist;
108     Int num_display_params;
109 } Aia_Func;
110
111 #define AIA_FUNC_OF(slot) HASH_MAP_ENTRY(slot, Aia_Func, hash_slot)
112
113 typedef struct Aia_Section {
114     Const_String section_name;
115     Aia_Block *entry_block;
116     Aia_Block *exit_block;

```

```

117     Aia *aia;
118     void *meta_data;
119     Hash_Map functions;
120     Double_List sec_blist;
121     Aia_Section_Type sec_type;
122 } Aia_Section;
123
124 typedef struct Aia {
125     String source_file_name;
126     File_Location source_null_file_loc;
127     Aia_Section *sections[AIA_SECTION_COUNT];
128     Aia_Func *curr_func;
129     Aia_Block *curr_block;
130     Double_List *curr_blist;
131     void *meta_data;
132     void (*aia_instr_dump_callback)(FILE *stream, Aia_Instr *in);
133     Aia_Operand *record_self_ptr;
134     Hash_Map label_instr_map;
135     Hash_Map default_funcs;
136     Aia_Section_Type curr_sec;
137     Uns curr_tmp_reg_num;
138     Uns curr_tmp_label_num;
139 } Aia;
140
141 #define AIA_FOR_EACH_SECTION(aia, sec) \
142     for (Uns __i = 0; __i < AIA_SECTION_COUNT; __i++) \
143         if ((sec = (aia)->sections[__i]))
144
145 static inline Aia_Operand *aia_operand_display_ref_alloc(Aia *aia UNUSED,
146     Aia_Operand *display_reg, Const_String var_name,
147     Const_String func_name /*, uint8_t var_size */)
148 {
149     Aia_Operand *op = ALLOC_NEW(Aia_Operand);
150     op->iden.display_ref = ALLOC_NEW(Aia_Operand_Display_Ref);
151     *op->iden.display_ref = (Aia_Operand_Display_Ref){
152         display_reg,
153         string_duplicate(var_name),
154         string_duplicate(func_name) /*,
155         var_size */
156     };
157     ++display_reg->ref_count;
158     op->op_type = AIA_OPERAND_DISPLAY_REF;
159     op->ref_count = 0;
160     return op;
161 }
162
163 static inline Aia_Operand *__aia_operand_local_ref_alloc(Aia *aia UNUSED,
164     String var_name /*, uint8_t var_size */)
165 {
166     Aia_Operand *op = ALLOC_NEW(Aia_Operand);
167     op->iden.local_ref = ALLOC_NEW(Aia_Operand_Local_Ref);
168     *op->iden.local_ref = (Aia_Operand_Local_Ref){
169         var_name /*,
170         var_size */
171     };
172     op->op_type = AIA_OPERAND_LOCAL_REF;
173     op->ref_count = 0;
174     return op;
175 }
176
177 static inline Aia_Operand *aia_operand_local_ref_alloc(Aia *aia,
178     Const_String var_name /*, uint8_t var_size */)
179 {
180     return __aia_operand_local_ref_alloc(aia, string_duplicate(var_name) /*,
181     var_size */);
182 }
183
184 static inline Aia_Operand *aia_operand_arg_alloc(Aia *aia UNUSED,
185     int32_t idx)
186 {
187     Aia_Operand *op = ALLOC_NEW(Aia_Operand);
188     op->iden.int_const = idx;
189     op->op_type = AIA_OPERAND_ARG;
190     op->ref_count = 0;

```

```

191     return op;
192 }
193
194 static inline Aia_Operand *aia_operand_const_int_alloc(Aia *aia UNUSED,
195     int32_t int_val)
196 {
197     Aia_Operand *op = ALLOC_NEW(Aia_Operand);
198     op->iden.int_const = int_val;
199     op->op_type = AIA_OPERAND_CONST_INT;
200     op->ref_count = 0;
201     return op;
202 }
203
204 static inline Aia_Operand *aia_operand_const_string_alloc(Aia *aia UNUSED,
205     Const_String op_name)
206 {
207     Aia_Operand *op = ALLOC_NEW(Aia_Operand);
208     op->iden.op_name = string_duplicate(op_name);
209     op->op_type = AIA_OPERAND_CONST_STRING;
210     op->ref_count = 0;
211     return op;
212 }
213
214 static inline Aia_Operand *aia_operand_block_alloc(Aia *aia UNUSED,
215     Aia_Block *block)
216 {
217     Aia_Operand *op = ALLOC_NEW(Aia_Operand);
218     op->iden.op_block = block;
219     op->op_type = AIA_OPERAND_BLOCK;
220     op->ref_count = 0;
221     return op;
222 }
223
224 static inline Aia_Operand *__aia_operand_label_alloc(Aia *aia UNUSED,
225     String name, Int offset, Aia_Operand_Type lbl_type)
226 {
227     assert(lbl_type == AIA_OPERAND_LABEL ||
228         lbl_type == AIA_OPERAND_LABEL_ADDR);
229
230     Aia_Operand *op = ALLOC_NEW(Aia_Operand);
231     op->iden.op_label = ALLOC_NEW(Aia_Operand_Label);
232     op->op_type = lbl_type;
233     op->iden.op_label->offset = offset;
234     op->iden.op_label->label_name = name;
235     op->ref_count = 0;
236     return op;
237 }
238
239 static inline Aia_Operand *aia_operand_label_alloc(Aia *aia,
240     Const_String name, Int offset)
241 {
242     return __aia_operand_label_alloc(aia, string_duplicate(name),
243         offset, AIA_OPERAND_LABEL);
244 }
245
246 static inline Aia_Operand *aia_operand_label_addr_alloc(Aia *aia,
247     Const_String name, Int offset)
248 {
249     return __aia_operand_label_alloc(aia, string_duplicate(name),
250         offset, AIA_OPERAND_LABEL_ADDR);
251 }
252
253 static inline Aia_Operand *__aia_operand_reg_alloc(Aia *aia UNUSED,
254     String reg)
255 {
256     Aia_Operand *op = ALLOC_NEW(Aia_Operand);
257     op->iden.op_name = reg;
258     op->op_type = AIA_OPERAND_REG;
259     op->ref_count = 0;
260     return op;
261 }
262
263 static inline Aia_Operand *aia_operand_reg_alloc(Aia *aia, Const_String reg)
264 {

```

```

265     return __aia_operand_reg_alloc(aia, string_duplicate(reg));
266 }
267
268 static inline Aia_Operand *aia_operand_tmp_reg_alloc(Aia *aia)
269 {
270     return __aia_operand_reg_alloc(aia, string_from_format(S("%U"),
271     aia->curr_tmp_reg_num++));
272 }
273
274 static inline String aia_tmp_num_append(Aia *aia, Const_String prefix)
275 {
276     return string_from_format(S("%S.%U"), prefix, aia->curr_tmp_label_num++);
277 }
278
279 static inline String aia_tmp_name_gen(Aia *aia, Const_String prefix)
280 {
281     return string_from_format(S("%.S.%U"), prefix, aia->curr_tmp_label_num++);
282 }
283
284 static inline Aia_Operand *aia_operand_tmp_label_alloc(Aia *aia,
285     Const_String prefix)
286 {
287     String lbl_name = aia_tmp_name_gen(aia, prefix);
288     return __aia_operand_label_alloc(aia, lbl_name, 0, AIA_OPERAND_LABEL);
289 }
290
291 Aia_Operand *aia_operand_addr_ref_alloc(Aia *aia,
292     Aia_Operand *label,
293     Aia_Operand *disp,
294     Aia_Operand *base,
295     Aia_Operand *index,
296     Aia_Operand *scale);
297
298 static inline void __aia_block_append_instr(Aia_Block *b, Aia_Instr *in);
299
300 static inline void __aia_curr_block_append_instr(Aia *aia, Aia_Instr *in)
301 {
302     __aia_block_append_instr(aia->curr_block, in);
303 }
304
305 static inline Const_String aia_get_file_name(Aia *aia);
306
307 static inline Aia_Instr *__aia_0op_instr(Aia *aia, uint32_t type,
308     uint8_t dest_op_size, File_Location *loc)
309 {
310     File_Location tmp = FILE_LOCATION_INIT(aia_get_file_name(aia),
311     loc->line, loc->column);
312     Aia_Instr *instr = aia_instr_alloc_vop(type, aia->curr_block,
313     dest_op_size, -1, 0, &tmp);
314     __aia_curr_block_append_instr(aia, instr);
315     return instr;
316 }
317
318 static inline Aia_Instr *__aia_1op_instr(Aia *aia, uint32_t type,
319     uint8_t dest_op_size, uint8_t src_op_size, File_Location *loc)
320 {
321     File_Location tmp = FILE_LOCATION_INIT(aia_get_file_name(aia),
322     loc->line, loc->column);
323     Aia_Instr *instr = aia_instr_alloc_vop(type, aia->curr_block,
324     dest_op_size, src_op_size, 1, &tmp);
325     __aia_curr_block_append_instr(aia, instr);
326     return instr;
327 }
328
329 static inline Aia_Instr *__aia_2op_instr(Aia *aia, uint32_t type,
330     uint8_t dest_op_size, uint8_t src_ops_size, File_Location *loc)
331 {
332     File_Location tmp = FILE_LOCATION_INIT(aia_get_file_name(aia),
333     loc->line, loc->column);
334     Aia_Instr *instr = aia_instr_alloc_vop(type, aia->curr_block,
335     dest_op_size, src_ops_size, 2, &tmp);
336     __aia_curr_block_append_instr(aia, instr);
337     return instr;
338 }

```



```

339
340 static inline Aia_Instr *__aia_vop_instr(Aia *aia, uint32_t type,
341     uint8_t dest_op_size, uint8_t src_ops_size, Uns nops,
342     File_Location *loc)
343 {
344     File_Location tmp = FILE_LOCATION_INIT(aia_get_file_name(aia),
345     loc->line, loc->column);
346     Aia_Instr *instr = aia_instr_alloc_vop(type, aia->curr_block,
347     dest_op_size, src_ops_size, nops, &tmp);
348     __aia_curr_block_append_instr(aia, instr);
349     return instr;
350 }
351
352 static inline void __aia_block_append_instr(Aia_Block *b, Aia_Instr *in)
353 {
354     double_list_append(&b->instructions, &in->dbnode);
355 }
356
357 static inline void __aia_block_append_predecessor(Aia_Block *b,
358     Aia_Block *pred)
359 {
360     vector_append(&b->predecessors, pred);
361 }
362
363 static inline Aia_Block *__aia_block_alloc(Aia_Section *sec,
364     Aia_Func *func)
365 {
366     assert(sec);
367     Aia_Block *b = ALLOC_NEW(Aia_Block);
368     b->instructions = DOUBLE_LIST_INIT(b->instructions);
369     b->predecessors = VECTOR_INIT_SIZE(2);
370     b->section = sec;
371     b->function = func;
372     b->visit_count = 0;
373     return b;
374 }
375
376 static inline Aia_Block *__aia_block(Aia *aia)
377 {
378     Aia_Block *b = __aia_block_alloc(aia->sections[aia->curr_sec],
379     aia->curr_func);
380     aia->curr_block = b;
381     double_list_append(aia->curr_blist, &b->blist_node);
382     return b;
383 }
384
385 static inline Aia_Block *__aia_block_alloc_entry(Aia_Section *sec,
386     Aia_Func *f)
387 {
388     Aia_Block *entry = __aia_block_alloc(sec, f);
389     return entry;
390 }
391
392 Aia_Block *aia_block_twin(Aia_Block *b, Aia *aia);
393
394 static inline Uns aia_block_predecessor_count(Aia_Block *b)
395 {
396     return vector_size(&b->predecessors);
397 }
398
399 static inline Uns aia_block_sucessor_count(Aia_Block *b)
400 {
401     Double_List_Node *ln = double_list_peek_last(&b->instructions);
402     switch (AIA_INSTR_OF(ln)->type) {
403     AIA_CASE_COND_JUMP:
404         return 2;
405     case AIA_JMP:
406         return 1;
407     default:
408         return 0;
409     }
410 }
411
412 Aia_Instr *__aia_block_peek_first_instr(Aia_Block *b);

```

```

413
414 Aia_Instr *__aia_block_peek_last_instr(Aia_Block *b);
415
416 #define AIA_BLOCK_FOR_EACH_SUCESSOR(block, sucessor) \
417     for (Double_List_Node *__n = \
418         double_list_peek_last(&(block)->instructions); \
419         __n; \
420         __n = NULL) \
421     for (Int __c = aia_block_sucessor_count(b) - 1; \
422         __c >= 0; \
423         __c--) \
424         if ((sucessor = aia_instr_get_src_op(AIA_INSTR_OF(__n), \
425             __c)->iden.op_block))
426
427 #define AIA_BLOCK_FOR_EACH_PREDECESSOR(block, predecessor) \
428     VECTOR_FOR_EACH_ENTRY(&block->predecessors, predecessor)
429
430 #define AIA_BLOCK_FOR_EACH_INSTRUCTION(block, instr) \
431     for (Double_List_Node *__n = NULL; __n == NULL;) \
432     DOUBLE_LIST_FOR_EACH(&block->instructions, __n) \
433     if ((instr = AIA_INSTR_OF(__n)) || !instr)
434
435 #define AIA_BLOCK_FOR_EACH_INSTRUCTION_REVERSED(block, instr) \
436     for (Double_List_Node *__n = NULL; __n == NULL;) \
437     DOUBLE_LIST_FOR_EACH_REVERSED(&block->instructions, __n) \
438     if ((instr = AIA_INSTR_OF(__n)) || !instr)
439
440 void aia_block_remove_from_predecessors(Aia_Block *b);
441
442 void aia_block_blist_remove_destroy(Aia_Block *b);
443
444 static inline void __aia_block_lone_destroy(Aia_Block *b)
445 {
446     double_list_for_each_destroy(&b->instructions, __aia_instr_db_destroy);
447     vector_clear(&b->predecessors);
448     free_mem(b);
449 }
450
451 #if 0
452 static inline bool aia_block_is_empty(Aia_Block *b)
453 {
454     bool ret = false;
455     if (aia_block_predecessor_count(b) != 1 ||
456         aia_block_sucessor_count(b) != 1)
457         goto out;
458
459     Uns idx = 0;
460     Aia_Instr *in;
461     AIA_BLOCK_FOR_EACH_INSTRUCTION(b, in) {
462         if (idx == 1) {
463             switch (aia_instr_get_type(in)) {
464                 AIA_CASE_COND_JUMP:
465                 case AIA_JMP:
466                     ret = true;
467
468             default:
469                 goto out;
470             }
471         } else if (idx > 1) {
472             ret = false;
473             goto out;
474         }
475         ++idx;
476     }
477 out:
478     return ret;
479 }
480 #endif
481
482 typedef void (*Aia_Block_Callback)(Aia_Block *block, void *arg);
483
484 static inline void __aia_block_zero_visit_count(Double_List *blist)
485 {
486     Double_List_Node *bnode;

```

```

487     DOUBLE_LIST_FOR_EACH(blist, bnode)
488         AIA_BLOCK_OF_DBNODE(bnode)->visit_count = 0;
489 }
490
491 /* Don't call this function. Use aia_func_for_each_block_depth(),
492  * or aia_section_for_each_block_depth() instead. */
493 static inline void __aia_block_for_each_depth(Aia_Block *entry_block,
494     Aia_Block_Callback callback, void *arg)
495 {
496     extern void __aia_block_for_each_depth_visit(Aia_Block *b,
497         Aia_Block_Callback callback, void *arg);
498
499     __aia_block_for_each_depth_visit(entry_block, callback, arg);
500 }
501
502 static inline void __aia_block_for_each_depth2(Aia_Block *entry_block,
503     Aia_Block_Callback start_callback,
504     Aia_Block_Callback ret_callback,
505     void *arg)
506 {
507     extern void __aia_block_for_each_depth2_visit(Aia_Block *b,
508         Aia_Block_Callback start_callback,
509         Aia_Block_Callback ret_callback,
510         void *arg);
511
512     __aia_block_for_each_depth2_visit(entry_block, start_callback,
513         ret_callback, arg);
514 }
515
516 /* Don't call this function. Use aia_func_for_each_block_depth2(),
517  * or aia_section_for_each_block_depth2() instead. */
518 static inline void aia_func_for_each_block_depth2(Aia_Func *f,
519     Aia_Block_Callback start_cb, Aia_Block_Callback ret_cb, void *arg)
520 {
521     __aia_block_zero_visit_count(&f->blist);
522     __aia_block_for_each_depth2(f->entry_block, start_cb, ret_cb, arg);
523 }
524
525 static inline void aia_func_for_each_block_depth(Aia_Func *f,
526     Aia_Block_Callback callback, void *arg)
527 {
528     __aia_block_zero_visit_count(&f->blist);
529     __aia_block_for_each_depth(f->entry_block, callback, arg);
530 }
531
532 void __aia_block_lone_db_destroy(Double_List_Node *n);
533
534 void aia_block_forget_predecessor(Aia_Block *b, Aia_Block *pred);
535
536 void __aia_block_recursive_destroy(Aia_Block *root);
537
538 Aia_Func *__aia_func_alloc(Aia *aia, Const_String func_name);
539
540 static inline void __aia_set_curr_func_parent(Aia *aia, Aia_Func *parent)
541 {
542     assert(aia->curr_func);
543     aia->curr_func->parent_func = parent;
544 }
545
546 static inline void aia_set_curr_func_location(Aia *aia, File_Location *loc)
547 {
548     assert(aia->curr_func);
549     aia->curr_func->location = FILE_LOCATION_INIT(aia_get_file_name(aia),
550         loc->line, loc->column);
551 }
552
553 static inline Aia_Func *aia_func_get_parent_func(Aia_Func *f)
554 {
555     return f->parent_func;
556 }
557
558 static inline File_Location *aia_func_get_last_location(Aia_Func *f)
559 {
560     Aia_Instr *last_in = __aia_block_peek_last_instr(f->exit_block);

```

```

561     return aia_instr_get_location(last_in);
562 }
563
564 static inline bool aia_func_is_nested(Aia_Func *f)
565 {
566     return aia_func_get_parent_func(f);
567 }
568
569 static inline Const_String aia_func_get_name(Aia_Func *f)
570 {
571     return f->func_name;
572 }
573
574 static inline File_Location *aia_func_get_location(Aia_Func *f)
575 {
576     return &f->location;
577 }
578
579 #define AIA_FUNC_GET_SOURCE_NAME(aia_func) \
580     STRING_AFTER_LAST((aia_func)->func_name, '.')
581
582 static inline Aia_Block *aia_func_get_exit_block(Aia_Func *f)
583 {
584     return f->exit_block;
585 }
586
587 static inline Aia_Block *aia_func_get_entry_block(Aia_Func *f)
588 {
589     return f->entry_block;
590 }
591
592 Aia_Func_Trampoline *__aia_func_trampoline_alloc(Aia *aia,
593     Const_String tramp_name, Vector *params, Aia_Func *func);
594
595 void __aia_func_trampoline_destroy(Aia_Func_Trampoline *t);
596
597 void __aia_func_destroy(Aia_Func *f);
598
599 /* Get function with name func_name. Return NULL if func does not exist. */
600 Aia_Func *aia_func_lookup(Aia *aia, Const_String func_name);
601
602 bool __aia_func_comparator(String search_func, Hash_Map_Slot *maps);
603
604 void __aia_func_hash_destroy(Hash_Map_Slot *s);
605
606 static inline void __aia_section_destroy(Aia_Section *s)
607 {
608     hash_map_for_each_destroy(&s->functions, __aia_func_hash_destroy);
609     //__aia_block_for_each_successor_destroy(s->entry_block);
610     double_list_for_each_destroy(&s->sec_blist, __aia_block_lone_db_destroy);
611     free_mem(s);
612 }
613
614 extern Const_String __aia_section_names[AIA_SECTION_COUNT];
615
616 static inline Const_String __aia_section_name(
617     Aia_Section_Type sec)
618 {
619     return __aia_section_names[sec];
620 }
621
622 static inline void __aia_section_init(Aia_Section *s,
623     Aia_Section_Type sec, Aia *aia)
624 {
625     s->section_name = __aia_section_name(sec);
626     Aia_Block *entry = __aia_block_alloc_entry(s, NULL);
627     s->entry_block = s->exit_block = entry;
628     s->functions = HASH_MAP_INIT_SIZE(HASH_MAP_SIZE_23,
629     (Hash_Map_Comparator) __aia_func_comparator);
630     s->sec_blist = DOUBLE_LIST_INIT(s->sec_blist);
631     s->sec_type = sec;
632     s->aia = aia;
633     double_list_append(&s->sec_blist, &s->entry_block->blist_node);
634 }

```

```

635
636 static inline Const_String aia_section_get_name(Aia_Section *s)
637 {
638     return s->section_name;
639 }
640
641 static inline Aia_Block *aia_section_get_entry_block(Aia_Section *s)
642 {
643     return s->entry_block;
644 }
645
646 static inline Aia_Block *aia_section_get_exit_block(Aia_Section *s)
647 {
648     return s->exit_block;
649 }
650
651 static inline void aia_section_for_each_block_depth(Aia_Section *s,
652     Aia_Block_Callback callback, void *arg)
653 {
654     __aia_block_zero_visit_count(&s->sec_blist);
655     __aia_block_for_each_depth(s->entry_block, callback, arg);
656 }
657
658 static inline void aia_section_for_each_block_depth2(Aia_Section *s,
659     Aia_Block_Callback start_cb, Aia_Block_Callback ret_cb, void *arg)
660 {
661     __aia_block_zero_visit_count(&s->sec_blist);
662     __aia_block_for_each_depth2(s->entry_block, start_cb, ret_cb, arg);
663 }
664
665 /* As long as sec < AIA_SECTION_COUNT this function
666  * returns a valid section. */
667 Aia_Section *__aia_get_section(Aia *aia,
668     Aia_Section_Type sec);
669
670 // Warning use goto to break this loop
671 #define AIA_SECTION_FOR_EACH_FUNC(sec, fun) \
672     for (Hash_Map_Slot *__fs = INT_TO_PTR(1); __fs; __fs = NULL) \
673         HASH_MAP_FOR_EACH(&(sec)->functions, __fs) \
674             if ((fun = AIA_FUNC_OF(__fs)) || !fun)
675
676 #define AIA_SECTION_FOR_EACH_BLOCK(sec, block) \
677     for (Double_List_Node *__bnode = NULL; __bnode == NULL;) \
678         DOUBLE_LIST_FOR_EACH(&(sec)->sec_blist, __bnode) \
679             if ((block = AIA_BLOCK_OF_DBNODE(__bnode)) || !block)
680
681 #define AIA_SECTION_FOR_EACH_BLOCK_REVERSED(sec, block) \
682     for (Double_List_Node *__bnode = NULL; __bnode == NULL;) \
683         DOUBLE_LIST_FOR_EACH_REVERSED(&(sec)->sec_blist, __bnode) \
684             if ((block = AIA_BLOCK_OF_DBNODE(__bnode)) || !block)
685
686 static inline void __aia_set_exit_block(Aia *aia)
687 {
688     if (aia->curr_func)
689         aia->curr_func->exit_block = aia->curr_block;
690     else if (aia->curr_block)
691         aia->sections[aia->curr_sec]->exit_block = aia->curr_block;
692 }
693
694 static inline void __aia_set_section(Aia *aia,
695     Aia_Section_Type sec, Aia_Section *s)
696 {
697     __aia_set_exit_block(aia);
698     aia->sections[sec] = s;
699     aia->curr_sec = sec;
700     aia->curr_func = NULL;
701     aia->curr_block = aia->sections[sec]->exit_block;
702     aia->curr_blist = &aia->sections[sec]->sec_blist;
703 }
704
705 static inline Aia_Section *aia_get_section(Aia *aia, Aia_Section_Type t)
706 {
707     return aia->sections[t];
708 }

```

```

709
710 static inline void __aia_switch_section(Aia *aia, Aia_Section_Type sec)
711 {
712     Aia_Section *s = __aia_get_section(aia, sec);
713     if (s)
714         __aia_set_section(aia, sec, s);
715     // else __aia_get_section() called __aia_set_section() for us.
716 }
717
718 static inline Aia_Section_Type __aia_get_curr_section_type(Aia *aia)
719 {
720     return aia->curr_sec;
721 }
722
723 Aia_Func *__aia_get_func(Aia *aia, Const_String func_name);
724
725 static inline void __aia_set_func(Aia *aia, Aia_Func *func)
726 {
727     aia->curr_func = func;
728     aia->curr_block = aia->curr_func->exit_block;
729     aia->curr_blist = &func->blist;
730 }
731
732 static inline void __aia_switch_func(Aia *aia, Const_String func_name)
733 {
734     __aia_set_exit_block(aia);
735     if (func_name) {
736         /* Currently it's only allowed to add functions to .text section. */
737         assert(!string_compare(
738             aia->sections[aia->curr_sec]->section_name,
739             __aia_section_name(AIA_SECTION_TEXT)));
740
741         Aia_Func *func = __aia_get_func(aia, func_name);
742         if (func)
743             __aia_set_func(aia, func);
744     } else {
745         aia->curr_func = NULL;
746         aia->curr_block = aia->sections[aia->curr_sec]->exit_block;
747         aia->curr_blist = &aia->sections[aia->curr_sec]->sec_blist;
748     }
749 }
750
751 static inline Aia_Block *__aia_get_curr_block(Aia *aia)
752 {
753     return aia->curr_block;
754 }
755
756 static inline Aia_Func *__aia_get_curr_func(Aia *aia)
757 {
758     return aia->curr_func;
759 }
760
761 static inline void __aia_set_func_trampoline_block(Aia *aia,
762     Aia_Func_trampoline *t)
763 {
764     aia->curr_block = t->block;
765     aia->curr_blist = &t->blist;
766 }
767
768 static inline void __aia_set_curr_func(Aia *aia, Aia_Func *f)
769 {
770     aia->curr_blist = &f->blist;
771     aia->curr_func = f;
772 }
773
774 /* idx == -1 means preserve own display. */
775 static inline void aia_func_append_preserve_display(Aia *aia, Int idx)
776 {
777     assert(aia->curr_func);
778     vector_append(&aia->curr_func->preserve_display_indices, INT_TO_PTR(idx));
779 }
780
781 /* idx == -1 means preserve own display. */
782 static inline void aia_func_prepend_preserve_display(Aia *aia, Int idx)

```

```

783 {
784     assert(aia->curr_func);
785     vector_insert(&aia->curr_func->preserve_display_indices, 0,
786                 INT_TO_PTR(idx));
787 }
788
789 static inline void __aia_func_append_param(Aia *aia, String param)
790 {
791     assert(aia->curr_func);
792     vector_append(&aia->curr_func->parameters, param);
793 }
794
795 static inline void __aia_func_append_display_param(Aia *aia, String param)
796 {
797     ++aia->curr_func->num_display_params;
798     __aia_func_append_param(aia, param);
799 }
800
801 static inline void aia_func_append_param(Aia *aia, Const_String param)
802 {
803     __aia_func_append_param(aia, string_duplicate(param));
804 }
805
806 static inline Int aia_func_get_num_display_params(Aia_Func *f)
807 {
808     return f->num_display_params;
809 }
810
811 static inline void aia_func_append_local(Aia_Func *f, Const_String var_name)
812 {
813     vector_append(&f->locals, string_duplicate(var_name));
814 }
815
816 static inline void __aia_func_append_local(Aia *aia, Const_String var_name)
817 {
818     assert(aia->curr_func);
819     aia_func_append_local(aia->curr_func, var_name);
820 }
821
822 static inline void aia_func_set_linkage(Aia *aia, Aia_Linkage linkage)
823 {
824     assert(aia->curr_func);
825     Aia_Block *b = aia->curr_func->entry_block;
826     Double_List_Node *n = double_list_peek_first(&b->instructions);
827     Aia_Instr *in = AIA_INSTR_OF(n);
828     assert(in->type == __AIA_LABEL);
829     Aia_Operand *op = aia_instr_get_src_op(in, 0);
830     Aia_Label_Data lbld;
831     lbld.data = aia_operand_const_int_get_val(op);
832     lbld.linkage = linkage;
833     op->iden.int_const = lbld.data;
834 }
835
836 static inline Aia_Linkage aia_func_get_linkage(Aia_Func *f)
837 {
838     Aia_Block *b = f->entry_block;
839     Double_List_Node *n = double_list_peek_first(&b->instructions);
840     Aia_Instr *in = AIA_INSTR_OF(n);
841     assert(in->type == __AIA_LABEL);
842     Aia_Operand *op = aia_instr_get_src_op(in, 1);
843     return op->iden.int_const;
844 }
845
846 /* Returns NULL if not currently inside function. */
847 static inline Const_String __aia_get_curr_func_name(Aia *aia)
848 {
849     return aia->curr_func ? aia->curr_func->func_name : NULL;
850 }
851
852 #define AIA_FUNC_FOR_EACH_TRAMPOLINE(func, tramp) \
853     VECTOR_FOR_EACH_ENTRY(&(func)->trampolines, tramp)
854
855 #define AIA_FUNC_FOR_EACH_BLOCK(func, block) \
856     for (Double_List_Node *__bnode = NULL; __bnode == NULL;) \

```

```

857     DOUBLE_LIST_FOR_EACH(&(func)->blist, ____bnode) \
858     if ((block = AIA_BLOCK_OF_DBNODE(____bnode)) || !block)
859
860 #define AIA_FUNC_FOR_EACH_BLOCK_REVERSED(func, block) \
861     for (Double_List_Node *____bnode = NULL; ____bnode == NULL;) \
862     DOUBLE_LIST_FOR_EACH_REVERSED(&(func)->blist, ____bnode) \
863     if ((block = AIA_BLOCK_OF_DBNODE(____bnode)) || !block)
864
865 #define AIA_FUNC_TRAMPOLINE_FOR_EACH_BLOCK(tramp, block) \
866     if ((block = (tramp)->block) || !block)
867
868 #define AIA_FUNC_TRAMPOLINE_FOR_EACH_BLOCK_REVERSED(tramp, block) \
869     AIA_FUNC_TRAMPOLINE_FOR_EACH_BLOCK(tramp, block)
870
871 static inline Aia_Instr *aia_label_to_instruction(Aia *aia, Aia_Operand *lbl)
872 {
873     assert(lbl);
874     Hash_Map_Slot *s = hash_map_get(&aia->label_instr_map, lbl,
875     string_hash_code(lbl->iden.op_label->label_name));
876     if (!s)
877         return NULL;
878     return AIA_LABEL_INSTR_ENTRY_OF(s)->label_instr;
879 }
880
881 void ____aia_insert_label_instr(Aia *aia, Const_String lbl_name,
882     int32_t lbl_offset, uint8_t alignment, Aia_Linkage linkage,
883     Aia_Label_Type lbl_type, int32_t obj_byte_size,
884     File_Location *loc);
885
886 static inline void ____aia_insert_jump_label_instr(Aia *aia,
887     Const_String lbl_name)
888 {
889     ____aia_insert_label_instr(aia, lbl_name, 0, 0, AIA_LINKAGE_PRIVATE,
890     AIA_LABEL_TYPE_NONE, 0, &aia->source_null_file_loc);
891 }
892
893 static inline void ____aia_insert_string_instr(Aia *aia, Aia_Operand *str_op,
894     File_Location *loc)
895 {
896     File_Location tmp = FILE_LOCATION_INIT(aia_get_file_name(aia),
897     loc->line, loc->column);
898     Aia_Instr *str_instr = ____aia_lob_instr(aia, ____AIA_STRING,
899     0, AIA_LONG, &tmp);
900     aia_instr_set_src_op(str_instr, 0, str_op);
901 }
902
903 static inline void ____aia_insert_const_val_instr(Aia *aia, Aia_Operand *op,
904     uint8_t aia_size /* AIA_BYTE or AIA_LONG. */, File_Location *loc)
905 {
906     File_Location tmp = FILE_LOCATION_INIT(aia_get_file_name(aia),
907     loc->line, loc->column);
908     Aia_Instr *instr = ____aia_lob_instr(aia, ____AIA_INTEGER,
909     0, aia_size, &tmp);
910     aia_instr_set_src_op(instr, 0, op);
911 }
912
913 Aia_Instr *aia_private_label_before(Aia *aia, Aia_Operand *lbl,
914     Aia_Instr *sucessor);
915
916 Aia_Instr *aia_type_mov_before(uint16_t mov_instr, Aia_Operand *src_op,
917     Aia_Operand *dest_op, Aia_Instr *sucessor,
918     uint8_t src_size, uint8_t dest_size);
919
920 Aia_Instr *aia_mov_before(Aia_Operand *src_op, Aia_Operand *dest_op,
921     Aia_Instr *sucessor, uint8_t op_sizes);
922
923 Aia_Instr *aia_mov_after(Aia_Operand *src_op, Aia_Operand *dest_op,
924     Aia_Instr *predecessor, uint8_t op_sizes);
925
926 static inline Aia_Instr *aia_mov_to_tmp_reg_before(Aia *aia,
927     Aia_Operand *src_op, Aia_Instr *sucessor, uint8_t op_sizes)
928 {
929     Aia_Operand *tmp = aia_operand_tmp_reg_alloc(aia);
930     return aia_mov_before(src_op, tmp, sucessor, op_sizes);

```



```

931 }
932
933 Aia *aia_alloc(Const_String src_fname);
934
935 void __aia_finish(Aia *aia);
936
937 static inline bool aia_is_valid(Aia *aia)
938 {
939     return aia;
940 }
941
942 static inline void aia_destroy(Aia *aia)
943 {
944     if (!aia_is_valid(aia))
945         return;
946
947     string_destroy(aia->source_file_name);
948
949     Aia_Section *sec;
950     AIA_FOR_EACH_SECTION(aia, sec)
951         __aia_section_destroy(sec);
952     hash_map_for_each_destroy(&aia->label_instr_map, aia_label_entry_destroy);
953     hash_map_for_each_destroy(&aia->default_funcs, __aia_func_hash_destroy);
954
955     __aia_operand_release(aia->record_self_ptr);
956     free_mem(aia);
957 }
958
959 /* Remember to call aia_clear_instr_dump_callback() once AIA is dumped. */
960 static inline void aia_set_instr_dump_callback(Aia *aia,
961     void (*cb)(FILE *stream, Aia_Instr *in))
962 {
963     aia->aia_instr_dump_callback = cb;
964 }
965
966 static inline void aia_clear_instr_dump_callback(Aia *aia)
967 {
968     aia->aia_instr_dump_callback = NULL;
969 }
970
971 void aia_dump(Aia *aia, FILE *stream);
972
973 static inline Const_String aia_get_file_name(Aia *aia)
974 {
975     assert(aia_is_valid(aia));
976     return aia->source_file_name;
977 }
978
979 static inline File_Location *aia_get_null_location(Aia *aia)
980 {
981     assert(aia_is_valid(aia));
982     return &aia->source_null_file_loc;
983 }
984
985 #define AIA_FOR_EACH_FUNC(aia, func) \
986     for (Aia_Section *sec = INT_TO_PTR(1); sec; sec = NULL) \
987         AIA_FOR_EACH_SECTION(aia, sec) \
988             AIA_SECTION_FOR_EACH_FUNC(sec, func)
989
990 #undef DEBUG_TYPE
991 #define DEBUG_TYPE def
992
993 #endif // AIA_H

```

:

A.7.3 src/aia/aia_functions_return.c

```

1  #include "aia.h"
2
3  #undef DEBUG_TYPE

```

```

4  #define DEBUG_TYPE func-ret
5
6  static bool block_control_returns(Aia_Block *b)
7  {
8      if (b->visit_count)
9          return true;
10     b->visit_count = 1;
11
12     Aia_Instr *in;
13     AIA_BLOCK_FOR_EACH_INSTRUCTION(b, in)
14         if (aia_instr_get_type(in) == AIA_RET)
15             return true;
16
17     bool ret = false;
18     Aia_Block *suc;
19     AIA_BLOCK_FOR_EACH_SUCESSOR(b, suc) {
20         if (!block_control_returns(suc))
21             return false;
22         else
23             ret = true;
24     }
25
26     return ret;
27 }
28
29 static bool aia_func_returns(Aia_Func *f)
30 {
31     Aia_Block *b;
32     AIA_FUNC_FOR_EACH_BLOCK(f, b)
33         b->visit_count = 0;
34
35     if (block_control_returns(aia_func_get_entry_block(f)))
36         return true;
37
38     report_error_location(aia_func_get_last_location(f),
39         S("control flow reaches end of function " QFY("%S")
40         " without " QFY("return") "\n"),
41         AIA_FUNC_GET_SOURCE_NAME(f));
42     return false;
43 }
44
45 bool aia_functions_return(Aia *aia)
46 {
47     bool ret = true;
48
49     if (!aia_is_valid(aia))
50         goto out;
51
52     Aia_Func *func;
53     AIA_FOR_EACH_FUNC(aia, func)
54         ret &= aia_func_returns(func);
55
56 out:
57     return ret;
58 }

```

:

A.7.4 src/aia/aia_functions_return.h

```

1  #ifndef AIA_FUNCTIONS_RETURN_H
2  #define AIA_FUNCTIONS_RETURN_H
3
4  bool aia_functions_return(Aia *aia);
5
6  #endif // AIA_FUNCTIONS_RETURN_H

```

:

A.7.5 src/aia/aia_instr.c

```

1  #include "aia.h"
2
3  #undef DEBUG_TYPE
4  #define DEBUG_TYPE aia-instr
5
6  static void aia_instr_label_aia_remove(Aia_Instr *instr)
7  {
8      if (aia_instr_get_type(instr) != __AIA_LABEL)
9          return;
10
11     Aia_Block *b = aia_instr_get_block(instr);
12     Aia_Section *sec = b->section;
13     Aia *aia = sec->aia;
14
15     Aia_Operand *op = aia_instr_get_dest_op(instr);
16     Hash_Map_Slot *s = hash_map_remove(&aia->label_instr_map, op,
17         string_hash_code(aia_operand_label_get_name(op)));
18     assert(s);
19     free_mem(AIA_LABEL_INSTR_ENTRY_OF(s));
20 }
21
22 void __aia_instr_destroy(Aia_Instr *instr)
23 {
24     Aia_Operand *op;
25     AIA_INSTR_FOR_EACH_OPERAND(instr, op)
26         __aia_operand_release(op);
27     free_mem(instr);
28 }
29
30 void aia_instr_destroy(Aia_Instr *instr)
31 {
32     aia_instr_label_aia_remove(instr);
33     __aia_instr_destroy(instr);
34 }
35
36 void __aia_instr_db_destroy(Double_List_Node *n)
37 {
38     aia_instr_destroy(AIA_INSTR_OF(n));
39 }
40
41 void aia_instr_label_dump(FILE *stream, Aia_Instr *instr)
42 {
43     STRING(str, "");
44
45     Aia_Operand *tmp = aia_instr_get_src_op(instr, 0);
46     Aia_Label_Data lbl_data;
47     lbl_data.data = aia_operand_const_int_get_val(tmp);
48
49     tmp = aia_instr_get_dest_op(instr);
50     Const_String lbl_name = aia_operand_label_get_name(tmp);
51
52     switch (lbl_data.linkage) {
53     case AIA_LINKAGE_PRIVATE:
54         break;
55     case AIA_LINKAGE_WEAK:
56         string_append_format(str, S(".weak %S\n"), lbl_name);
57         break;
58     case AIA_LINKAGE_GLOBAL:
59         string_append_format(str, S(".globl %S\n"), lbl_name);
60         break;
61     }
62
63     switch (lbl_data.label_type) {
64     case AIA_LABEL_TYPE_NONE:
65         break;
66     case AIA_LABEL_TYPE_FUNC:
67         string_append_format(str, S(".type %S, @function\n"), lbl_name);
68         break;
69     case AIA_LABEL_TYPE_OBJ:
70         string_append_format(str, S(".type %S, @object\n"), lbl_name);
71         break;

```

```

72     }
73
74     tmp = aia_instr_get_src_op(instr, 1);
75     int32_t size = aia_operand_const_int_get_val(tmp);
76     if (size)
77         string_append_format(str, S(".size %S, %" PRId32 "\n"),
78                             lbl_name, size);
79
80     if (lbl_data.alignment)
81         string_append_format(str, S(".align %u\n"), lbl_data.alignment);
82
83     string_append_format(str, S("%S:"),
84                         aia_instr_get_dest_op(instr)->iden.op_label->label_name);
85     file_print_message(stream, str);
86     string_clear(str);
87 }
88
89 void aia_instr_string_dump(FILE *stream, Aia_Instr *instr)
90 {
91     Const_String name = aia_instr_get_src_op(instr, 0)->iden.op_name;
92     file_print_message(stream, S("\t.byte "));
93     Uns len = string_length(name);
94     for (Uns i = 0; i < len; i++)
95         file_print_message(stream, S("%d,"), string_get(name, i));
96     file_print_message(stream, S("0"));
97 }
98
99 void aia_instr_integer_dump(FILE *stream, Aia_Instr *instr)
100 {
101     Aia_Operand *op = aia_instr_get_src_op(instr, 0);
102     if (instr->src_ops_size == AIA_LONG) {
103         file_print_message(stream, S("\t.long "));
104     } else if (aia_operand_get_type(op) == AIA_OPERAND_CONST_INT) {
105         assert((int8_t)aia_instr_get_src_op(instr, 0)->iden.int_const ==
106              aia_instr_get_src_op(instr, 0)->iden.int_const);
107
108         file_print_message(stream, S("\t.byte "));
109     }
110     aia_operand_dump(stream, op, false);
111 }
112
113 void aia_instr_call_dump(FILE *stream, Aia_Instr *instr,
114                          Const_String name)
115 {
116     file_print_message(stream, S("\t%S"), name);
117     if (instr->src_op_count) {
118         file_print_message(stream, S(" "));
119         assert(instr->src_ops_size == AIA_LONG);
120         aia_operand_dump_size(stream, instr->src_ops_size);
121         Int i;
122         Aia_Operand *call_op = aia_instr_get_src_op(instr, 0);
123         if (call_op->op_type != AIA_OPERAND_LABEL)
124             file_print_message(stream, S("*"));
125         aia_operand_dump(stream, call_op, true);
126
127         if (instr->src_op_count > 1) {
128             file_print_message(stream, S(", "));
129             for (i = 1; i < (Int)instr->src_op_count - 1; i++) {
130                 aia_operand_dump(stream, aia_instr_get_src_op(instr, i), true);
131                 file_print_message(stream, S(", "));
132             }
133             aia_operand_dump(stream, aia_instr_get_src_op(instr, i), true);
134         }
135     }
136 }
137
138 if (instr->dest_op) {
139     file_print_message(stream, S(" -> "));
140     aia_operand_dump_size(stream, instr->dest_op_size);
141     aia_operand_dump(stream, instr->dest_op, true);
142 }
143
144 void aia_instr_nop_dump(FILE *stream, Aia_Instr *instr UNUSED,
145                        Const_String name)

```

```

146 {
147     file_print_message(stream, S("\t%S"), name);
148 }
149
150 void aia_instr_vop_dump(FILE *stream, Aia_Instr *instr,
151     Const_String name)
152 {
153     file_print_message(stream, S("\t%S"), name);
154     if (instr->src_op_count) {
155         file_print_message(stream, S(" "));
156         aia_operand_dump_size(stream, instr->src_ops_size);
157         Int i;
158         if (instr->src_op_count > 0) {
159             for (i = 0; i < (Int)instr->src_op_count - 1; i++) {
160                 aia_operand_dump(stream, aia_instr_get_src_op(instr, i), true);
161                 file_print_message(stream, S(", "));
162             }
163             aia_operand_dump(stream, aia_instr_get_src_op(instr, i), true);
164         }
165     }
166     if (instr->dest_op) {
167         file_print_message(stream, S(" -> "));
168         aia_operand_dump_size(stream, instr->dest_op_size);
169         aia_operand_dump(stream, instr->dest_op, true);
170     }
171 }
172
173 void aia_instr_jump_dump(FILE *stream, Aia_Instr *instr,
174     Const_String name)
175 {
176     Aia_Operand *op;
177     file_print_message(stream, S("\t%S "), name);
178
179     assert(!instr->dest_op);
180     assert(instr->src_op_count > 0 && instr->src_op_count <= 2);
181     assert(instr->src_ops_size == AIA_LONG);
182
183     aia_operand_dump_size(stream, instr->src_ops_size);
184     op = aia_instr_get_src_op(instr, 0);
185     aia_operand_dump(stream, op, true);
186
187     if (instr->src_op_count == 2) {
188         file_print_message(stream, S(", "));
189         op = aia_instr_get_src_op(instr, 1);
190         aia_operand_dump(stream, op, true);
191     }
192 }
193
194 void aia_instr_dump(FILE *stream, Aia_Instr *instr, Aia *aia)
195 {
196     switch (instr->type) {
197     case AIA_MOV:
198         aia_instr_vop_dump(stream, instr, S(".mov"));
199         break;
200     case AIA_MOVS:
201         aia_instr_vop_dump(stream, instr, S(".movs"));
202         break;
203     case AIA_MOVZ:
204         aia_instr_vop_dump(stream, instr, S(".movz"));
205         break;
206     case AIA_ADD:
207         aia_instr_vop_dump(stream, instr, S(".add"));
208         break;
209     case AIA_SUB:
210         aia_instr_vop_dump(stream, instr, S(".sub"));
211         break;
212     case AIA_IMUL:
213         aia_instr_vop_dump(stream, instr, S(".imul"));
214         break;
215     case AIA_IDIV:
216         aia_instr_vop_dump(stream, instr, S(".idiv"));
217         break;
218     case AIA_CMP:
219         aia_instr_vop_dump(stream, instr, S(".cmp"));

```

```

220     break;
221 case AIA_SETE:
222     aia_instr_vop_dump(stream, instr, S(".sete"));
223     break;
224 case AIA_SETNE:
225     aia_instr_vop_dump(stream, instr, S(".setne"));
226     break;
227 case AIA_SETL:
228     aia_instr_vop_dump(stream, instr, S(".setl"));
229     break;
230 case AIA_SETG:
231     aia_instr_vop_dump(stream, instr, S(".setg"));
232     break;
233 case AIA_SETLE:
234     aia_instr_vop_dump(stream, instr, S(".setle"));
235     break;
236 case AIA_SETGE:
237     aia_instr_vop_dump(stream, instr, S(".setge"));
238     break;
239 case AIA_RET:
240     aia_instr_vop_dump(stream, instr, S(".ret"));
241     break;
242 case AIA_CALL:
243     aia_instr_call_dump(stream, instr, S(".call"));
244     break;
245 case AIA_NEG:
246     aia_instr_vop_dump(stream, instr, S(".neg"));
247     break;
248 case __AIA_LABEL:
249     aia_instr_label_dump(stream, instr);
250     break;
251 case __AIA_STRING:
252     aia_instr_string_dump(stream, instr);
253     break;
254 case __AIA_INTEGER:
255     aia_instr_integer_dump(stream, instr);
256     break;
257 case AIA_JNE:
258     aia_instr_jump_dump(stream, instr, S(".jne"));
259     break;
260 case AIA_JE:
261     aia_instr_jump_dump(stream, instr, S(".je"));
262     break;
263 case AIA_JGE:
264     aia_instr_jump_dump(stream, instr, S(".jge"));
265     break;
266 case AIA_JG:
267     aia_instr_jump_dump(stream, instr, S(".jg"));
268     break;
269 case AIA_JLE:
270     aia_instr_jump_dump(stream, instr, S(".jle"));
271     break;
272 case AIA_JL:
273     aia_instr_jump_dump(stream, instr, S(".jl"));
274     break;
275 case AIA_JMP:
276     aia_instr_jump_dump(stream, instr, S(".jmp"));
277     break;
278 case AIA_CDQ:
279     aia_instr_vop_dump(stream, instr, S(".cdq"));
280     break;
281 case AIA_NOP:
282     aia_instr_nop_dump(stream, instr, S(".nop"));
283     break;
284
285 case __AIA_JNE:
286     /* Fall */
287 case __AIA_JE:
288     /* Fall */
289 case __AIA_JMP:
290     fatal_error(S("Unable to dump unexpected jump instruction. "
291                  "Aborting...\n"));
292 default:
293     fatal_error(S("Unable to dump unexpected instruction. "

```

```

294         "Aborting...\n"));
295     }
296
297     if (aia->aia_instr_dump_callback)
298         aia->aia_instr_dump_callback(stream, instr);
299
300     file_print_message(stream, S("\n"));
301 }
302
303 void aia_label_entry_destroy(Hash_Map_Slot *s)
304 {
305     free_mem(AIA_LABEL_INSTR_ENTRY_OF(s));
306 }
307
308 bool aia_label_instr_comparator(Aia_Operand *search_lbl, Hash_Map_Slot *mslot)
309 {
310     Aia_Label_Instr_Entry *e = AIA_LABEL_INSTR_ENTRY_OF(mslot);
311     return !string_compare(search_lbl->iden.op_label->label_name,
312         aia_instr_get_dest_op(e->label_instr)->iden.op_label->label_name);
313 }
314
315 void aia_instr_append_sucessors(Aia_Instr *in, Vector *sucessors)
316 {
317     Aia_Instr *suc = aia_instr_get_sucessor(in);
318     if (suc) {
319         vector_append(sucessors, suc);
320     } else {
321         Aia_Block *b = aia_instr_get_block(in);
322         Aia_Block *suc_b;
323         AIA_BLOCK_FOR_EACH_SUCESSOR(b, suc_b) {
324             suc = __aia_block_peek_first_instr(suc_b);
325             vector_append(sucessors, suc);
326         }
327     }
328 }
329
330 void aia_instr_cond_jump_to_jump(Aia_Instr *cond_jump, Aia_Instr *jmp,
331     Aia_Operand *preserve_block)
332 {
333     Aia_Operand *op = aia_instr_get_src_op(cond_jump, 0);
334     Aia_Block *b = aia_operand_block_get_block(op);
335
336     if (!aia_operands_equal(op, preserve_block)) {
337         aia_block_forget_predecessor(b, aia_instr_get_block(cond_jump));
338     } else {
339         op = aia_instr_get_src_op(cond_jump, 1);
340         b = aia_operand_block_get_block(op);
341         assert(!aia_operands_equal(op, preserve_block));
342         aia_block_forget_predecessor(b, aia_instr_get_block(cond_jump));
343     }
344
345     __aia_instr_replace(cond_jump, jmp);
346 }
347
348 Aia_Instr *aia_instr_twin(Aia_Instr *in, Aia_Block *contain_block)
349 {
350     uint16_t in_type = aia_instr_get_type(in);
351     Aia_Instr *twin = aia_instr_alloc_vop(in_type,
352         contain_block,
353         aia_instr_get_dest_op_size(in),
354         aia_instr_get_src_ops_size(in),
355         aia_instr_get_src_op_count(in),
356         aia_instr_get_location(in));
357
358     Int idx = 0;
359     Aia_Operand *op;
360     AIA_INSTR_FOR_EACH_SRC(in, op)
361         aia_instr_set_src_op(twin, idx++, op);
362
363     op = aia_instr_get_dest_op(in);
364     if (op)
365         aia_instr_set_dest_op(twin, op);
366
367     return twin;

```

```

368 }
369
370 void aia_instr_get_predecessors(Aia_Instr *in, Vector *preds)
371 {
372     Aia_Instr *p = aia_instr_get_predecessor(in);
373     if (p) {
374         vector_append(preds, p);
375         return;
376     }
377     Aia_Block *b = aia_instr_get_block(in);
378     Aia_Block *pred_b;
379     AIA_BLOCK_FOR_EACH_PREDECESSOR(b, pred_b) {
380         p = __aia_block_peek_last_instr(pred_b);
381         vector_append(preds, p);
382     }
383 }
384
385 void aia_instr_get_sucessors(Aia_Instr *in, Vector *sucs)
386 {
387     Aia_Instr *s = aia_instr_get_sucessor(in);
388     if (s) {
389         vector_append(sucs, s);
390         return;
391     }
392     Aia_Block *b = aia_instr_get_block(in);
393     Aia_Block *suc_b;
394     AIA_BLOCK_FOR_EACH_PREDECESSOR(b, suc_b) {
395         s = __aia_block_peek_last_instr(suc_b);
396         vector_append(sucs, s);
397     }
398 }

```

:

A.7.6 src/aia/aia_instr.h

```

1  #ifndef AIA_INSTR_H
2  #define AIA_INSTR_H
3
4  #include <std_include.h>
5  #include <double_list.h>
6  #include <vector.h>
7  #include <hash_map.h>
8  #include "aia_operand.h"
9
10 #define AIA_CASE_INCONCRETE \
11     case __AIA_JNE: \
12     case __AIA_JL: \
13     case __AIA_JMP: \
14     case __AIA_JGE: \
15     case __AIA_STRING: \
16     case __AIA_INTEGER: \
17     case __AIA_LABEL: \
18     case AIA_NOP
19
20 #define AIA_CASE_SET \
21     case AIA_SETG: \
22     case AIA_SETGE: \
23     case AIA_SETL: \
24     case AIA_SETLE: \
25     case AIA_SETE: \
26     case AIA_SETNE
27
28 #define AIA_CASE_COND_TMP_JUMP \
29     case __AIA_JNE: \
30     case __AIA_JGE: \
31     case __AIA_JL: \
32     case __AIA_JE
33
34 #define AIA_CASE_COND_JUMP \
35     case AIA_JNE: \

```



```

36     case AIA_JGE:      \
37     case AIA_JL:       \
38     case AIA_JE:       \
39     case AIA_JG:       \
40     case AIA_JLE
41
42 #define AIA_LABEL_JUMP_TO_BLOCK_JUMP(jmp_type) ({ \
43     Aia_Operand_Type ____r; \
44     switch (jmp_type) { \
45     case ____AIA_JNE: \
46         ____r = AIA_JNE; \
47         break; \
48     case ____AIA_JE: \
49         ____r = AIA_JE; \
50         break; \
51     case ____AIA_JGE: \
52         ____r = AIA_JGE; \
53         break; \
54     case ____AIA_JMP: \
55         ____r = AIA_JMP; \
56         break; \
57     default: \
58         fatal_error(S("unexpected jump instruction. Aborting...\n")); \
59     } \
60     ____r; \
61 })
62
63 typedef struct Aia Aia;
64
65 /* AIA instructions. */
66 enum {
67     // .mov @size(x) src -> @size(x) dest
68     AIA_MOV,
69     // .movs @size(4) src -> @size(1) dest
70     AIA_MOVS,
71     // .movz @size(4) src -> @size(1) dest
72     AIA_MOVZ,
73     // .add @size(4) src1, src2 -> @size(4) dest
74     AIA_ADD,
75     // .sub @size(4) src1, src2 -> @size(4) dest # src1 - src2
76     AIA_SUB,
77     // .imul @size(4) src1, src2 -> @size(4) dest
78     AIA_IMUL,
79     // .idiv @size(4) src1, src2 -> @size(4) dest # src1 / src2
80     AIA_IDIV,
81     // .cmp @size(x) src1, src2
82     AIA_CMP,
83     // .jne @size(4) ne_label -> @size(4) e_label
84     AIA_JNE,
85     // .je @size(4) e_label -> @size(4) ne_label
86     AIA_JE,
87     // .jge @size(4) ge_label -> @size(4) l_label
88     AIA_JGE,
89     // .jg @size(4) g_label -> @size(4) le_label
90     AIA_JG,
91     // .jle @size(4) le_label -> @size(4) g_label
92     AIA_JLE,
93     // .jl @size(4) l_label -> @size(4) ge_label
94     AIA_JL,
95     // .jmp -> @size(4) label
96     AIA_JMP,
97     // .sete -> @size(1) dest
98     AIA_SETE,
99     // .setne -> @size(1) dest
100    AIA_SETNE,
101    // .setl -> @size(1) dest
102    AIA_SETL,
103    // .setg -> @size(1) dest
104    AIA_SETG,
105    // .setle -> @size(1) dest
106    AIA_SETLE,
107    // .setge -> @size(1) dest
108    AIA_SETGE,
109    // .ret [@size(x) src] # opt return value, no dest operand

```

```

110     AIA_RET,
111     // .call @size(4) mem/reg -> @size(x) result
112     AIA_CALL,
113     // .neg @size(4) src -> @size(4) dest
114     AIA_NEG,
115     // .cdq @size(4) src -> @size(4) dest # sign extend src into dest
116     AIA_CDQ,
117
118     // .nop
119     AIA_NOP,
120
121     // .tmp.jne @size(4) ne_label, e_label (gets replaced by AIA_JNE)
122     __AIA_JNE,
123     // .tmp.je @size(4) e_label, ne_label (gets replaced by AIA_JE)
124     __AIA_JE,
125     // .tmp.jge @size(4) ge_label, nge_label (gets replaced by AIA_JGE)
126     __AIA_JGE,
127     // .tmp.jl @size(4) l_label, nl_label (gets replaced by AIA_JL)
128     __AIA_JL,
129     // .tmp.jmp @size(4) label (gets replaced by AIA_JMP)
130     __AIA_JMP,
131
132     // __AIA_LABEL, dest is label operand,
133     // 1st src is an Aia_Label_Data union where
134     // the fields have the meaning:
135     // when alignment != 0:
136     // [.align alignment]
137     // when linkage != PRIVATE:
138     // [|.globl dest_label|.weak dest_label]
139     // when label_type != NONE
140     // [|.type dest_lbl @function|.type dest_lbl @object]
141     // 2nd src operand is specifying an object size
142     // when size != 0:
143     // [.size dest_lbl x]
144     // dest_label:
145     __AIA_LABEL, // Dest is a label (AIA_OPERAND_LABEL).
146     // .byte c1, c2, ..., cn
147     __AIA_STRING, // Src is a static string (AIA_OPERAND_CONST_STRING).
148     // [.byte|.long] int_val
149     __AIA_INTEGER // Src is a static int (AIA_OPERAND_CONST_INT).
150 };
151
152 typedef enum Aia_Cmp_Result {
153     AIA_CMP_NONE,
154     AIA_CMP_UNKNOWN,
155     AIA_CMP_LESS,
156     AIA_CMP_EQUAL,
157     AIA_CMP_GREAT
158 } Aia_Cmp_Result;
159
160 /* Source operand sizes. */
161 enum {
162     AIA_BYTE,
163     AIA_LONG
164 };
165
166 typedef struct Aia_Instr {
167     File_Location location;
168     Double_List_Node dbnode;
169     uint16_t type;
170     uint8_t dest_op_size;
171     uint8_t src_ops_size;
172     int32_t src_op_count;
173     void *meta_data;
174     int32_t flags;
175     Aia_Block *containing_block;
176     Aia_Operand *dest_op;
177     Aia_Operand *src_ops[0]; /* Optional source operands. */
178 } Aia_Instr;
179
180 #define AIA_INSTR_FLAG_NORMALIZED 1
181
182 #define AIA_INSTR_OF(node) DOUBLE_LIST_ENTRY(node, Aia_Instr, dbnode)
183

```

```

184 typedef struct Aia_Label_Instr_Entry {
185     Aia_Instr *label_instr;
186     Hash_Map_Slot hash_slot;
187 } Aia_Label_Instr_Entry;
188
189 #define AIA_LABEL_INSTR_ENTRY_OF(slot) \
190     HASH_MAP_ENTRY(slot, Aia_Label_Instr_Entry, hash_slot)
191
192 #define AIA_INSTR_FOR_EACH_SRC(instr, op) \
193     for (int32_t __i = 0; __i < (instr)->src_op_count; __i++) \
194         if ((op = (instr)->src_ops[__i]) || !op)
195
196 #define AIA_INSTR_FOR_EACH_OPERAND(instr, op) \
197     for (Aia_Operand *__op = aia_instr_get_dest_op(instr), \
198          *__i = INT_TO_PTR(-1); \
199          (PTR_TO_INT(__i) < (instr)->src_op_count); \
200          __i = (Aia_Operand *) ((char *) __i + 1), \
201          __op = (PTR_TO_INT(__i) < (instr)->src_op_count) ? \
202                aia_instr_get_src_op(instr, PTR_TO_INT(__i)) : NULL) \
203          if ((op = __op) || !op)
204
205 static inline File_Location *aia_instr_get_location(Aia_Instr *in)
206 {
207     return &in->location;
208 }
209
210 static inline Aia_Instr *__aia_instr_alloc(uint16_t type, Aia_Block *b,
211      uint8_t dest_op_size, uint8_t src_ops_size, int32_t nsrc_ops,
212      File_Location *loc)
213 {
214     Aia_Instr *ret = alloc_mem(sizeof(Aia_Instr) +
215                                nsrc_ops * sizeof(Aia_Operand *));
216     ret->type = type;
217     ret->src_op_count = nsrc_ops;
218     ret->dest_op_size = dest_op_size;
219     ret->src_ops_size = src_ops_size;
220     ret->containing_block = b;
221     ret->dest_op = NULL;
222     ret->meta_data = NULL;
223     ret->flags = 0;
224     ret->location = FILE_LOCATION_INIT(file_location_get_file_name(loc),
225                                       loc->line, loc->column);
226     for (int32_t i = 0; i < nsrc_ops; i++)
227         ret->src_ops[i] = NULL;
228     return ret;
229 }
230
231 static inline bool aia_instr_is_normalized(Aia_Instr *in)
232 {
233     return in->flags & AIA_INSTR_FLAG_NORMALIZED;
234 }
235
236 static inline void aia_instr_set_normalized(Aia_Instr *in)
237 {
238     in->flags |= AIA_INSTR_FLAG_NORMALIZED;
239 }
240
241 static inline Aia_Instr *aia_instr_alloc_0op(uint16_t type,
242      Aia_Block *contain_block, uint8_t dest_op_size,
243      File_Location *loc)
244 {
245     return __aia_instr_alloc(type, contain_block, dest_op_size, -1, 0, loc);
246 }
247
248 static inline Aia_Instr *aia_instr_alloc_lop(uint16_t type,
249      Aia_Block *contain_block, uint8_t dest_op_size, uint8_t src_ops_size,
250      File_Location *loc)
251 {
252     return __aia_instr_alloc(type, contain_block, dest_op_size,
253                               src_ops_size, 1, loc);
254 }
255
256 static inline Aia_Instr *aia_instr_alloc_2op(uint16_t type,
257      Aia_Block *contain_block, uint8_t dest_op_size, uint8_t src_ops_size,

```

```

258     File_Location *loc)
259 {
260     return __aia_instr_alloc(type, contain_block, dest_op_size,
261                             src_ops_size, 2, loc);
262 }
263
264 static inline Aia_Instr *aia_instr_alloc_vop(uint16_t type,
265     Aia_Block *contain_block, uint8_t dest_op_size,
266     uint8_t src_ops_size, Uns num_src_ops, File_Location *loc)
267 {
268     return __aia_instr_alloc(type, contain_block, dest_op_size,
269                             src_ops_size, num_src_ops, loc);
270 }
271
272 static inline uint16_t aia_instr_get_type(Aia_Instr *in)
273 {
274     return in->type;
275 }
276
277 static inline Aia_Block *aia_instr_get_block(Aia_Instr *in)
278 {
279     return in->containing_block;
280 }
281
282 static inline void aia_instr_set_src_op(Aia_Instr *instr, int32_t idx,
283     Aia_Operand *src_op)
284 {
285     assert(instr->src_op_count > idx);
286     ++src_op->ref_count;
287     instr->src_ops[idx] = src_op;
288 }
289
290 static inline Aia_Operand *aia_instr_get_src_op(Aia_Instr *instr, int32_t idx)
291 {
292     assert(instr->src_op_count > idx);
293     return instr->src_ops[idx];
294 }
295
296 static inline int32_t aia_instr_get_src_op_count(Aia_Instr *in)
297 {
298     return in->src_op_count;
299 }
300
301 static inline void aia_instr_set_dest_op(Aia_Instr *instr,
302     Aia_Operand *dest_op)
303 {
304     assert(dest_op);
305     assert(aia_operand_is_dest(dest_op));
306     if (!dest_op->ref_count)
307         assert(!dest_op->ref_count);
308     ++dest_op->ref_count;
309     instr->dest_op = dest_op;
310 }
311
312 // idx == -1 => set dest op
313 static inline void aia_instr_set_op(Aia_Instr *in, Int idx, Aia_Operand *op)
314 {
315     if (idx == -1)
316         aia_instr_set_dest_op(in, op);
317     else
318         aia_instr_set_src_op(in, idx, op);
319 }
320
321 static inline Aia_Operand *aia_instr_get_dest_op(Aia_Instr *instr)
322 {
323     return instr->dest_op;
324 }
325
326 // idx == -1 => set dest op
327 static inline Aia_Operand *aia_instr_get_op(Aia_Instr *in, Int idx)
328 {
329     if (idx == -1)
330         return aia_instr_get_dest_op(in);
331     return aia_instr_get_src_op(in, idx);

```

```

332 }
333
334 static inline uint8_t aia_instr_get_src_ops_size(Aia_Instr *in)
335 {
336     return in->src_ops_size;
337 }
338
339 static inline uint8_t aia_instr_get_dest_op_size(Aia_Instr *in)
340 {
341     return in->dest_op_size;
342 }
343
344 static inline bool aia_instr_is_set_instr(Aia_Instr *in)
345 {
346     switch (aia_instr_get_type(in)) {
347     AIA_CASE_SET:
348         return true;
349     default:
350         return false;
351     }
352 }
353
354 static inline bool aia_instr_is_cond_jump(Aia_Instr *in)
355 {
356     switch (aia_instr_get_type(in)) {
357     AIA_CASE_COND_JUMP:
358         return true;
359     default:
360         return false;
361     }
362 }
363
364 static inline bool aia_instr_is_movx(Aia_Instr *in)
365 {
366     switch (aia_instr_get_type(in)) {
367     case AIA_MOVZ:
368     case AIA_MOVS:
369     case AIA_MOV:
370         return true;
371     default:
372         return false;
373     }
374 }
375
376 static inline bool aia_instr_is_concrete(Aia_Instr *in)
377 {
378     switch (aia_instr_get_type(in)) {
379     AIA_CASE_INCONCRETE:
380         return false;
381     default:
382         return true;
383     }
384 }
385
386
387 static inline void aia_instr_replace_op(Aia_Instr *in, Int old_op_idx,
388     Aia_Operand *new_op)
389 {
390     Aia_Operand *old_op = aia_instr_get_op(in, old_op_idx);
391     aia_instr_set_op(in, old_op_idx, new_op);
392     __aia_operand_release(old_op);
393 }
394
395 static inline void aia_instr_swap_ops(Aia_Instr *in, Int i1, Int i2)
396 {
397     assert(i1 != i2);
398
399     Aia_Operand *tmp;
400     if (i1 == -1) {
401         tmp = aia_instr_get_dest_op(in);
402         in->dest_op = aia_instr_get_src_op(in, i2);
403         in->src_ops[i2] = tmp;
404     } else if (i2 == -1) {
405         tmp = aia_instr_get_dest_op(in);

```

```

406     in->dest_op = aia_instr_get_src_op(in, i1);
407     in->src_ops[i1] = tmp;
408 } else {
409     tmp = aia_instr_get_src_op(in, i2);
410     in->src_ops[i2] = aia_instr_get_src_op(in, i1);
411     in->src_ops[i1] = tmp;
412 }
413 }
414
415 void aia_instr_destroy(Aia_Instr *instr);
416
417 void __aia_instr_destroy(Aia_Instr *instr);
418
419 /* Returns NULL if in is the last instruction in the block. */
420 static inline Aia_Instr *aia_instr_get_sucessor(Aia_Instr *in)
421 {
422     extern Aia_Instr *__aia_block_peek_last_instr(Aia_Block *b);
423
424     if (in == __aia_block_peek_last_instr(aia_instr_get_block(in)))
425         return NULL;
426
427     Aia_Instr *suc = AIA_INSTR_OF(in->dbnode.next);
428     return suc;
429 }
430
431 void aia_instr_append_sucessors(Aia_Instr *in, Vector *sucessors);
432
433 /* Returns NULL if in is the last instruction in the block. */
434 static inline Aia_Instr *aia_instr_get_predecessor(Aia_Instr *in)
435 {
436     extern Aia_Instr *__aia_block_peek_first_instr(Aia_Block *b);
437
438     if (in == __aia_block_peek_first_instr(aia_instr_get_block(in)))
439         return NULL;
440
441     Aia_Instr *pred = AIA_INSTR_OF(in->dbnode.prev);
442     return pred;
443 }
444
445 static inline void __aia_instr_remove(Aia_Instr *in)
446 {
447     double_list_remove(&in->dbnode);
448 }
449
450 static inline void aia_instr_remove(Aia_Instr *in)
451 {
452     DEBUGT(def,
453         switch (aia_instr_get_type(in)) {
454             AIA_CASE_COND_JUMP:
455             case AIA_JMP:
456                 assert(false);
457             default:
458                 break;
459         }
460     );
461     __aia_instr_remove(in);
462 }
463
464 static inline void aia_instr_remove_destroy(Aia_Instr *in)
465 {
466     aia_instr_remove(in);
467     aia_instr_destroy(in);
468 }
469
470 static inline void aia_instr_insert_before(Aia_Instr *in, Aia_Instr *sucessor)
471 {
472     double_list_insert(&in->dbnode, sucessor->dbnode.prev, &sucessor->dbnode);
473 }
474
475 static inline void aia_instr_insert_after(Aia_Instr *in, Aia_Instr *pred)
476 {
477     double_list_insert(&in->dbnode, &pred->dbnode, pred->dbnode.next);
478 }
479

```

```

480 static inline void __aia_instr_replace(Aia_Instr *old_in, Aia_Instr *new_in)
481 {
482     aia_instr_insert_before(new_in, old_in);
483     __aia_instr_remove(old_in);
484 }
485
486 static inline void __aia_instr_replace_destroy(Aia_Instr *old_in,
487     Aia_Instr *new_in)
488 {
489     aia_instr_insert_before(new_in, old_in);
490     __aia_instr_remove(old_in);
491     aia_instr_destroy(old_in);
492 }
493
494 static inline void aia_instr_replace(Aia_Instr *old_in, Aia_Instr *new_in)
495 {
496     aia_instr_insert_before(new_in, old_in);
497     aia_instr_remove(old_in);
498 }
499
500 static inline void aia_instr_replace_destroy(Aia_Instr *old_in,
501     Aia_Instr *new_in)
502 {
503     aia_instr_replace(old_in, new_in);
504     aia_instr_destroy(old_in);
505 }
506
507 void aia_instr_get_predecessors(Aia_Instr *in, Vector *preds);
508
509 void aia_instr_get_sucessors(Aia_Instr *in, Vector *sucs);
510
511 void aia_instr_cond_jump_to_jump(Aia_Instr *cond_jump, Aia_Instr *jmp,
512     Aia_Operand *preserve_block);
513
514 void aia_instr_dump(FILE *stream, Aia_Instr *instr, Aia *aia);
515
516 void __aia_instr_db_destroy(Double_List_Node *dbnode);
517
518 void aia_instr_label_dump(FILE *stream, Aia_Instr *instr);
519
520 void aia_instr_string_dump(FILE *stream, Aia_Instr *instr);
521
522 void aia_instr_integer_dump(FILE *stream, Aia_Instr *instr);
523
524 void aia_instr_call_dump(FILE *stream, Aia_Instr *instr,
525     Const_String name);
526
527 void aia_instr_vop_dump(FILE *stream, Aia_Instr *instr,
528     Const_String name);
529
530 void aia_instr_jump_dump(FILE *stream, Aia_Instr *instr,
531     Const_String name);
532
533 bool aia_label_instr_comparator(Aia_Operand *search_lbl, Hash_Map_Slot *mslot);
534
535 void aia_label_entry_destroy(Hash_Map_Slot *s);
536
537 Aia_Instr *aia_instr_twin(Aia_Instr *in, Aia_Block *contain_block);
538
539 #endif // AIA_INSTR_H

```

:

A.7.7 src/aia/aia_normalize_addr.c

```

1 #include "aia_normalize_addr.h"
2 #include "aia.h"
3 #include <main.h>
4
5 static Aia_Operand *aia_mov_to_temp_reg(Aia *aia, Aia_Operand *src_op,
6     Aia_Instr *sucessor)

```

```

7  {
8      Aia_Instr *mov_in = aia_mov_to_tmp_reg_before(aia, src_op, sucessor,
9          AIA_LONG);
10     aia_normalize_if_addr(aia, mov_in);
11     return aia_instr_get_dest_op(mov_in);
12 }
13
14 static void do_normalize_addr(Aia *aia, Aia_Instr *in, Int op_idx)
15 {
16     Aia_Operand *op = aia_instr_get_op(in, op_idx);
17
18     Aia_Operand *scale_op = aia_operand_addr_ref_get_scale(op);
19     Aia_Operand *disp_op = aia_operand_addr_ref_get_disp(op);
20
21     bool delete_disp;
22     if (!disp_op)
23         delete_disp = true;
24     else
25         delete_disp = false;
26
27     Aia_Operand *index_reg = aia_operand_addr_ref_get_index(op);
28     if (index_reg) {
29         if (aia_operand_get_type(index_reg) == AIA_OPERAND_CONST_INT) {
30             int32_t disp;
31             if (!scale_op)
32                 disp = aia_operand_const_int_get_val(index_reg);
33             else
34                 disp = aia_operand_const_int_get_val(index_reg) *
35                     aia_operand_const_int_get_val(scale_op);
36             if (!disp_op) {
37                 disp_op = aia_operand_const_int_alloc(aia, disp);
38             } else {
39                 Aia_Operand *tmp = aia_operand_const_int_alloc(aia,
40                     aia_operand_const_int_get_val(disp_op) + disp);
41                 __aia_operand_release(disp_op);
42                 __aia_operand_acquire(tmp);
43                 op->iden.addr_ref->disp = tmp;
44             }
45             index_reg = scale_op = NULL;
46         } else if (aia_operand_get_type(index_reg) != AIA_OPERAND_REG) {
47             index_reg = aia_mov_to_temp_reg(aia, index_reg, in);
48         }
49     }
50
51     Aia_Operand *base_reg = aia_operand_addr_ref_get_base(op);
52     if (base_reg) {
53         if (aia_operand_get_type(base_reg) == AIA_OPERAND_CONST_INT) {
54             int32_t disp = base_reg->iden.int_const;
55             if (!disp_op) {
56                 disp_op = aia_operand_const_int_alloc(aia, disp);
57             } else {
58                 Aia_Operand *tmp = aia_operand_const_int_alloc(aia,
59                     aia_operand_const_int_get_val(disp_op) + disp);
60                 __aia_operand_release(disp_op);
61                 __aia_operand_acquire(tmp);
62                 op->iden.addr_ref->disp = tmp;
63             }
64             base_reg = NULL;
65         } else if (aia_operand_get_type(base_reg) != AIA_OPERAND_REG) {
66             base_reg = aia_mov_to_temp_reg(aia, base_reg, in);
67         }
68     }
69
70     if (disp_op && !aia_operand_const_int_get_val(disp_op)) {
71         if (delete_disp)
72             __aia_operand_destroy(disp_op);
73         disp_op = NULL;
74     }
75
76     Aia_Operand *label_op = aia_operand_addr_ref_get_label(op);
77     Aia_Operand *new_op = aia_operand_addr_ref_alloc(aia, label_op, disp_op,
78         base_reg, index_reg, scale_op);
79
80     aia_instr_replace_op(in, op_idx, new_op);

```



```

81 }
82
83 static void do_normalize_display(Aia *aia, Aia_Instr *in, Int op_idx)
84 {
85     Aia_Operand *op = aia_instr_get_op(in, op_idx);
86     if (aia_operand_get_type(op) != AIA_OPERAND_REG) {
87         Aia_Operand *reg = aia_mov_to_temp_reg(aia,
88         aia_operand_display_ref_get_display_reg(op),
89         in);
90         Aia_Operand *new_op = aia_operand_display_ref_alloc(aia,
91         reg,
92         aia_operand_display_ref_get_var_name(op),
93         aia_operand_display_ref_get_func_name(op)
94         /*, aia_operand_display_ref_get_var_size(op) */);
95         aia_instr_replace_op(in, op_idx, new_op);
96     }
97 }
98
99 void aia_normalize_if_addr(Aia *aia, Aia_Instr *in)
100 {
101     Int op_idx = -1;
102     Aia_Operand *op;
103     AIA_INSTR_FOR_EACH_OPERAND(in, op) {
104         if (op) {
105             if (aia_operand_get_type(op) == AIA_OPERAND_ADDR_REF)
106                 do_normalize_addr(aia, in, op_idx);
107             else if (aia_operand_get_type(op) == AIA_OPERAND_DISPLAY_REF)
108                 do_normalize_display(aia, in, op_idx);
109         }
110         ++op_idx;
111     }
112 }
113
114 static void aia_norm_addr_dump(Aia *aia)
115 {
116     String fname = string_from_format(S("%S.vitaly.norm-addr-ic"),
117     aia_get_file_name(aia));
118     FILE *dfname = file_open(fname, S("w"));
119     if (!dfname)
120         fatal_error(S("unable to create file %S for intermediate "
121         "code dump [%m]\n"), fname);
122     string_destroy(fname);
123     aia_dump(aia, dfname);
124     file_close(dfname);
125 }
126
127 void aia_normalize_block_callback(Aia_Block *b, void *arg)
128 {
129     Aia *aia = arg;
130     Aia_Instr *in;
131     AIA_BLOCK_FOR_EACH_INSTRUCTION(b, in)
132         aia_normalize_if_addr(aia, in);
133 }
134
135 void aia_normalize_addr(Aia *aia)
136 {
137     if (!aia_is_valid(aia))
138         return;
139
140     Aia_Section *sec;
141     AIA_FOR_EACH_SECTION(aia, sec) {
142         aia_section_for_each_block_depth(sec,
143         aia_normalize_block_callback, aia);
144
145         Aia_Func *f;
146         AIA_SECTION_FOR_EACH_FUNC(sec, f) {
147             aia_func_for_each_block_depth(
148             f, aia_normalize_block_callback, aia);
149
150             Aia_Func_Trampoline *t;
151             AIA_FUNC_FOR_EACH_TRAMPOLINE(f, t)
152                 aia_normalize_block_callback(t->block, aia);
153         }
154     }

```

```

155
156     if (cmdopts.dump_norm_addr_ic)
157         aia_norm_addr_dump(aia);
158 }

```

:

A.7.8 src/aia/aia_normalize_addr.h

```

1  #ifndef AIA_NORMALIZE_ADDR_H
2  #define AIA_NORMALIZE_ADDR_H
3
4  #include <vector.h>
5
6  typedef struct Aia Aia;
7
8  typedef struct Aia_Instr Aia_Instr;
9
10 void aia_normalize_addr(Aia *aia);
11
12 void aia_normalize_if_addr(Aia *aia, Aia_Instr *in);
13
14 static inline void aia_normalize_addr_instructions(Aia *aia, Vector *v)
15 {
16     Aia_Instr *in;
17     VECTOR_FOR_EACH_ENTRY(v, in)
18         aia_normalize_if_addr(aia, in);
19 }
20
21 #endif // AIA_NORMALIZE_ADDR_H

```

:

A.7.9 src/aia/aia_operand.c

```

1  #include "aia.h"
2  #include <hash_map.h>
3
4  bool aia_operands_equal(Aia_Operand *lhs, Aia_Operand *rhs);
5
6  static inline bool aia_operand_regs_equal(Aia_Operand *lhs, Aia_Operand *rhs)
7  {
8      return !string_compare(aia_operand_reg_get_name(lhs),
9                          aia_operand_reg_get_name(rhs));
10 }
11
12 static inline bool aia_operand_labels_equal(Aia_Operand *lhs, Aia_Operand *rhs)
13 {
14     if (string_compare(aia_operand_label_get_name(lhs),
15                     aia_operand_label_get_name(rhs))
16         return false;
17     return aia_operand_label_get_offset(lhs) ==
18         aia_operand_label_get_offset(rhs);
19 }
20
21 static inline bool aia_operand_const_ints_equal(Aia_Operand *lhs,
22         Aia_Operand *rhs)
23 {
24     return aia_operand_const_int_get_val(lhs) ==
25         aia_operand_const_int_get_val(rhs);
26 }
27
28 static inline bool aia_operand_const_strings_equal(Aia_Operand *lhs,
29         Aia_Operand *rhs)
30 {
31     return !string_compare(aia_operand_const_string_get_val(lhs),
32         aia_operand_const_string_get_val(rhs));

```

```

33 }
34
35 static inline bool aia_operand_addr_ref_operands_equal(Aia_Operand *lhs,
36 Aia_Operand *rhs)
37 {
38     if (lhs && rhs)
39         return aia_operands_equal(lhs, rhs);
40     return lhs == rhs;
41 }
42
43 static inline bool aia_operand_addr_refs_equal(Aia_Operand *lhs,
44 Aia_Operand *rhs)
45 {
46     if (!aia_operand_addr_ref_operands_equal(
47         aia_operand_addr_ref_get_index(lhs),
48         aia_operand_addr_ref_get_index(rhs)))
49         return false;
50     if (!aia_operand_addr_ref_operands_equal(
51         aia_operand_addr_ref_get_base(lhs),
52         aia_operand_addr_ref_get_base(rhs)))
53         return false;
54     if (!aia_operand_addr_ref_operands_equal(
55         aia_operand_addr_ref_get_disp(lhs),
56         aia_operand_addr_ref_get_disp(rhs)))
57         return false;
58     if (!aia_operand_addr_ref_operands_equal(
59         aia_operand_addr_ref_get_scale(lhs),
60         aia_operand_addr_ref_get_scale(rhs)))
61         return false;
62     if (!aia_operand_addr_ref_operands_equal(
63         aia_operand_addr_ref_get_label(lhs),
64         aia_operand_addr_ref_get_label(rhs)))
65         return false;
66     return true;
67 }
68
69 static inline bool aia_operand_local_refs_equal(Aia_Operand *lhs,
70 Aia_Operand *rhs)
71 {
72     if (string_compare(aia_operand_local_ref_get_var_name(lhs),
73         aia_operand_local_ref_get_var_name(rhs)))
74         return false;
75     return true;
76 }
77
78 static inline bool aia_operand_args_equal(Aia_Operand *lhs,
79 Aia_Operand *rhs)
80 {
81     return aia_operand_arg_get_idx(lhs) ==
82         aia_operand_arg_get_idx(rhs);
83 }
84
85 static inline bool aia_operand_display_refs_equal(Aia_Operand *lhs,
86 Aia_Operand *rhs)
87 {
88     #if 0
89     if (aia_operand_display_ref_get_var_size(lhs) !=
90         aia_operand_display_ref_get_var_size(rhs))
91         return false;
92     #endif
93
94     if (string_compare(aia_operand_display_ref_get_var_name(lhs),
95         aia_operand_display_ref_get_var_name(rhs)))
96         return false;
97
98     if (string_compare(aia_operand_display_ref_get_func_name(lhs),
99         aia_operand_display_ref_get_func_name(rhs)))
100         return false;
101
102     return aia_operands_equal(aia_operand_display_ref_get_display_reg(lhs),
103         aia_operand_display_ref_get_display_reg(rhs));
104 }
105
106 static inline bool aia_operand_blocks_equal(Aia_Operand *lhs, Aia_Operand *rhs)

```

```

107 {
108     return aia_operand_block_get_block(lhs) ==
109         aia_operand_block_get_block(rhs);
110 }
111
112 bool aia_operands_equal(Aia_Operand *lhs, Aia_Operand *rhs)
113 {
114     Aia_Operand_Type t = aia_operand_get_type(lhs);
115     if (t != aia_operand_get_type(rhs))
116         return false;
117     switch (t) {
118     case AIA_OPERAND_REG:
119         return aia_operand_regs_equal(lhs, rhs);
120     case AIA_OPERAND_LABEL:
121         /* Fall through. */
122     case AIA_OPERAND_LABEL_ADDR:
123         return aia_operand_labels_equal(lhs, rhs);
124     case AIA_OPERAND_CONST_INT:
125         return aia_operand_const_ints_equal(lhs, rhs);
126     case AIA_OPERAND_CONST_STRING:
127         return aia_operand_const_strings_equal(lhs, rhs);
128     case AIA_OPERAND_ADDR_REF:
129         return aia_operand_addr_refs_equal(lhs, rhs);
130     case AIA_OPERAND_LOCAL_REF:
131         return aia_operand_local_refs_equal(lhs, rhs);
132     case AIA_OPERAND_ARG:
133         return aia_operand_args_equal(lhs, rhs);
134     case AIA_OPERAND_DISPLAY_REF:
135         return aia_operand_display_refs_equal(lhs, rhs);
136     case AIA_OPERAND_BLOCK:
137         return aia_operand_blocks_equal(lhs, rhs);
138     default:
139         fatal_error(S("unexpected operand type. Aborting...\n"));
140     }
141 }
142
143 void aia_operand_dump_size(FILE *stream, uint8_t type)
144 {
145     switch (type) {
146     case AIA_BYTE:
147         file_print_message(stream, S("@size(1) "));
148         break;
149     default: // just assume AIA_LONG
150         file_print_message(stream, S("@size(4) "));
151         break;
152     }
153 }
154
155 void aia_operand_dump(FILE *stream, Aia_Operand *op,
156     bool print_int_dollar)
157 {
158     if (!op)
159         return;
160
161     switch (op->op_type) {
162     case AIA_OPERAND_LABEL_ADDR:
163         if (op->iden.op_label->offset)
164             file_print_message(stream, S("%S+% PRIu32",
165                 op->iden.op_label->label_name,
166                 op->iden.op_label->offset));
167         else
168             file_print_message(stream, S("%S",
169                 op->iden.op_label->label_name));
170         break;
171
172     case AIA_OPERAND_REG:
173         file_print_message(stream, S("%%%S", op->iden.op_name));
174         break;
175
176     case AIA_OPERAND_LABEL:
177         if (op->iden.op_label->offset)
178             file_print_message(stream, S("%S+% PRIu32",
179                 op->iden.op_label->label_name,
180                 op->iden.op_label->offset));

```

```

181     else
182         file_print_message(stream, S("%S"),
183             op->iden.op_label->label_name);
184     break;
185
186 case AIA_OPERAND_CONST_INT:
187     if (print_int_dollar)
188         file_print_message(stream, S("$%" PRId32), op->iden.int_const);
189     else
190         file_print_message(stream, S("%" PRId32), op->iden.int_const);
191     break;
192
193 case AIA_OPERAND_ADDR_REF:
194     if (op->iden.addr_ref->label)
195         aia_operand_dump(stream, op->iden.addr_ref->label, false);
196     if (op->iden.addr_ref->disp && op->iden.addr_ref->label)
197         file_print_message(stream, S("+"));
198     if (op->iden.addr_ref->disp)
199         aia_operand_dump(stream, op->iden.addr_ref->disp, false);
200     file_print_message(stream, S("("));
201     if (op->iden.addr_ref->base)
202         aia_operand_dump(stream, op->iden.addr_ref->base, false);
203     if (op->iden.addr_ref->index || op->iden.addr_ref->scale)
204         file_print_message(stream, S(", "));
205     if (op->iden.addr_ref->index) {
206         aia_operand_dump(stream, op->iden.addr_ref->index, false);
207         if (op->iden.addr_ref->scale)
208             file_print_message(stream, S(", "));
209     }
210     if (op->iden.addr_ref->scale)
211         aia_operand_dump(stream, op->iden.addr_ref->scale, false);
212     file_print_message(stream, S(")"));
213     break;
214
215 case AIA_OPERAND_LOCAL_REF:
216     file_print_message(stream, S("@local(%S)"),
217         op->iden.local_ref->var_name);
218     break;
219
220 case AIA_OPERAND_ARG:
221     file_print_message(stream, S("@arg(%U)"),
222         aia_operand_arg_get_idx(op));
223     break;
224
225 case AIA_OPERAND_DISPLAY_REF:
226     file_print_message(stream, S("@display_ref("));
227     aia_operand_dump(stream,
228         aia_operand_display_ref_get_display_reg(op), false);
229     file_print_message(stream, S(":::%S::%S)"),
230         op->iden.display_ref->func_name,
231         op->iden.display_ref->var_name);
232     break;
233
234 case AIA_OPERAND_BLOCK:
235     Aia_Instr *lbl_instr =
236         __aia_block_peek_first_instr(op->iden.op_block);
237     aia_operand_dump(stream, aia_instr_get_dest_op(lbl_instr), false);
238     break;
239
240 default:
241     fatal_error(S("Unexpected operand type for dump\n"));
242 }
243 }
244
245 void __aia_operand_destroy(Aia_Operand *op)
246 {
247     Aia_Operand_Addr_Ref *addr_ref;
248     switch (op->op_type) {
249     case AIA_OPERAND_CONST_INT:
250         break;
251     case AIA_OPERAND_ADDR_REF:
252         addr_ref = op->iden.addr_ref;
253         __aia_operand_release(addr_ref->index);
254         __aia_operand_release(addr_ref->scale);

```

```

255     __aia_operand_release(addr_ref->base);
256     __aia_operand_release(addr_ref->disp);
257     __aia_operand_release(addr_ref->label);
258     free_mem(addr_ref);
259     break;
260 case AIA_OPERAND_LOCAL_REF:
261     string_destroy(op->iden.local_ref->var_name);
262     free_mem(op->iden.local_ref);
263     break;
264 case AIA_OPERAND_ARG:
265     break;
266 case AIA_OPERAND_DISPLAY_REF:
267     string_destroy(op->iden.display_ref->func_name);
268     string_destroy(op->iden.display_ref->var_name);
269     __aia_operand_release(op->iden.display_ref->display_reg);
270     free_mem(op->iden.display_ref);
271     break;
272 case AIA_OPERAND_LABEL:
273 case AIA_OPERAND_LABEL_ADDR:
274     string_destroy(op->iden.op_label->label_name);
275     free_mem(op->iden.op_label);
276     break;
277 case AIA_OPERAND_BLOCK:
278     break;
279 default:
280     string_destroy(op->iden.op_name);
281     break;
282 }
283 free_mem(op);
284 }
285
286 Uns aia_operand_block_hash_code(Aia_Operand *op)
287 {
288     return hash_map_aligned_ptr_hash(aia_operand_block_get_block(op));
289 }
290
291 Aia_Operand *aia_operand_display_ref_with_replaced_reg(Aia *aia,
292     Aia_Operand *disp_ref, Aia_Operand *new_reg)
293 {
294     assert(disp_ref->op_type == AIA_OPERAND_DISPLAY_REF);
295     return aia_operand_display_ref_alloc(aia, new_reg,
296     aia_operand_display_ref_get_var_name(disp_ref),
297     aia_operand_display_ref_get_func_name(disp_ref)
298     /* , aia_operand_display_ref_get_var_size(disp_ref) */);
299 }
300
301 Aia_Operand *aia_operand_addr_ref_with_replaced_index(Aia *aia,
302     Aia_Operand *addr_ref, Aia_Operand *new_op)
303 {
304     assert(aia_operand_get_type(addr_ref) == AIA_OPERAND_ADDR_REF);
305     return aia_operand_addr_ref_alloc(aia,
306     aia_operand_addr_ref_get_label(addr_ref),
307     aia_operand_addr_ref_get_disp(addr_ref),
308     aia_operand_addr_ref_get_base(addr_ref),
309     new_op,
310     aia_operand_addr_ref_get_scale(addr_ref));
311 }
312
313 Aia_Operand *aia_operand_addr_ref_with_replaced_base(Aia *aia,
314     Aia_Operand *addr_ref, Aia_Operand *new_op)
315 {
316     assert(aia_operand_get_type(addr_ref) == AIA_OPERAND_ADDR_REF);
317     return aia_operand_addr_ref_alloc(aia,
318     aia_operand_addr_ref_get_label(addr_ref),
319     aia_operand_addr_ref_get_disp(addr_ref),
320     new_op,
321     aia_operand_addr_ref_get_index(addr_ref),
322     aia_operand_addr_ref_get_scale(addr_ref));
323 }

```

:

A.7.10 src/aia/aia_operand.h

```

1  #ifndef AIA_OPERAND_H
2  #define AIA_OPERAND_H
3
4  #include <std_include.h>
5  #include <str.h>
6
7  /* Aia operand allocators are located in aia.h. */
8
9  typedef struct Aia_Operand Aia_Operand;
10
11 typedef struct Aia_Block Aia_Block;
12
13 typedef struct Aia Aia;
14
15 typedef struct Aia_Operand_Label {
16     String label_name;
17     Int offset;
18 } Aia_Operand_Label;
19
20 typedef struct Aia_Operand_Addr_Ref {
21     Aia_Operand *label;
22     Aia_Operand *disp;
23     Aia_Operand *base;
24     Aia_Operand *index;
25     Aia_Operand *scale;
26 } Aia_Operand_Addr_Ref;
27
28 typedef struct Aia_Operand_Local_Ref {
29     String var_name; // Unique variable name.
30     // uint8_t var_size; // AIA_BYTE or AIA_LONG.
31 } Aia_Operand_Local_Ref;
32
33 typedef struct Aia_Operand_Display_Ref {
34     Aia_Operand *display_reg; // Register with function display.
35     String var_name; // Unique variable name.
36     String func_name; // Unique function name.
37     //uint8_t var_size; // AIA_BYTE or AIA_LONG.
38 } Aia_Operand_Display_Ref;
39
40 typedef union Aia_Operand_Iden {
41     String op_name;
42     int32_t int_const;
43     Aia_Operand_Addr_Ref *addr_ref;
44     Aia_Operand_Local_Ref *local_ref;
45     Aia_Operand_Display_Ref *display_ref;
46     Aia_Operand_Label *op_label;
47     Aia_Block *op_block;
48 } Aia_Operand_Iden;
49
50 typedef enum Aia_Operand_Type {
51     AIA_OPERAND_REG, // %reg
52     AIA_OPERAND_LABEL, // label[+displacement]
53     AIA_OPERAND_LABEL_ADDR, // $label[+displacement]
54     AIA_OPERAND_CONST_INT, // 42
55     AIA_OPERAND_CONST_STRING, // "my-string"
56     AIA_OPERAND_ADDR_REF, // lbl+4(%1,%2,4)
57     AIA_OPERAND_LOCAL_REF, // local function variable or parameter
58     AIA_OPERAND_ARG, // argument to function call
59     AIA_OPERAND_DISPLAY_REF, // reference to function display
60     AIA_OPERAND_BLOCK // Reference to an Aia_Block
61 } Aia_Operand_Type;
62
63 struct Aia_Operand {
64     Aia_Operand_Iden iden;
65     Aia_Operand_Type op_type;
66     Uns ref_count;
67 };
68
69 static inline Aia_Operand_Type aia_operand_get_type(Aia_Operand *op)
70 {
71     return op->op_type;

```

```

72 }
73
74 static inline bool aia_operand_is_dest(Aia_Operand *op)
75 {
76     switch (aia_operand_get_type(op)) {
77         case AIA_OPERAND_REG:
78         case AIA_OPERAND_LABEL:
79         case AIA_OPERAND_LOCAL_REF:
80         case AIA_OPERAND_ARG:
81         case AIA_OPERAND_DISPLAY_REF:
82         case AIA_OPERAND_ADDR_REF:
83             return true;
84         default:
85             return false;
86     }
87 }
88
89 static inline bool aia_operand_is_mem(Aia_Operand *op)
90 {
91     switch (aia_operand_get_type(op)) {
92         case AIA_OPERAND_LABEL:
93         case AIA_OPERAND_LOCAL_REF:
94         case AIA_OPERAND_ARG:
95         case AIA_OPERAND_DISPLAY_REF:
96         case AIA_OPERAND_ADDR_REF:
97             return true;
98         default:
99             return false;
100     }
101 }
102
103 static inline bool aia_operand_is_reg(Aia_Operand *op)
104 {
105     switch (aia_operand_get_type(op)) {
106         case AIA_OPERAND_REG:
107             return true;
108         default:
109             return false;
110     }
111 }
112
113 static inline bool aia_operand_is_integer(Aia_Operand *op)
114 {
115     switch (aia_operand_get_type(op)) {
116         case AIA_OPERAND_CONST_INT:
117         case AIA_OPERAND_LABEL_ADDR:
118             return true;
119         default:
120             return false;
121     }
122 }
123
124 bool aia_operands_equal(Aia_Operand *lhs, Aia_Operand *rhs);
125
126 void __aia_operand_destroy(Aia_Operand *op);
127
128 static inline void __aia_operand_release(Aia_Operand *op)
129 {
130     if (op && !--op->ref_count)
131         __aia_operand_destroy(op);
132 }
133
134 static inline void __aia_operand_acquire(Aia_Operand *op)
135 {
136     assert(op);
137     ++op->ref_count;
138 }
139
140 static inline Const_String aia_operand_local_ref_get_var_name(Aia_Operand *op)
141 {
142     assert(op->op_type == AIA_OPERAND_LOCAL_REF);
143     return op->iden.local_ref->var_name;
144 }
145

```



```

146 static inline int32_t aia_operand_arg_get_idx(Aia_Operand *op)
147 {
148     assert(op->op_type == AIA_OPERAND_ARG);
149     return op->iden.int_const;
150 }
151
152 #if 0
153 static inline uint8_t aia_operand_local_ref_get_var_size(Aia_Operand *op)
154 {
155     assert(op->op_type == AIA_OPERAND_LOCAL_REF);
156     return op->iden.local_ref->var_size;
157 }
158 #endif
159
160 static inline Const_String aia_operand_display_ref_get_func_name(Aia_Operand *op)
161 {
162     assert(op->op_type == AIA_OPERAND_DISPLAY_REF);
163     return op->iden.display_ref->func_name;
164 }
165
166 static inline Const_String aia_operand_display_ref_get_var_name(Aia_Operand *op)
167 {
168     assert(op->op_type == AIA_OPERAND_DISPLAY_REF);
169     return op->iden.display_ref->var_name;
170 }
171
172 static inline Aia_Operand *aia_operand_display_ref_get_display_reg(
173     Aia_Operand *op)
174 {
175     assert(op->op_type == AIA_OPERAND_DISPLAY_REF);
176     return op->iden.display_ref->display_reg;
177 }
178
179 Aia_Operand *aia_operand_display_ref_with_replaced_reg(Aia *aia,
180     Aia_Operand *disp_ref, Aia_Operand *new_reg);
181
182 #if 0
183 static inline uint8_t aia_operand_display_ref_get_var_size(Aia_Operand *op)
184 {
185     assert(op->op_type == AIA_OPERAND_DISPLAY_REF);
186     return op->iden.display_ref->var_size;
187 }
188 #endif
189
190 static inline Aia_Block *aia_operand_block_get_block(Aia_Operand *op)
191 {
192     assert(op->op_type == AIA_OPERAND_BLOCK);
193     return op->iden.op_block;
194 }
195
196 static inline Const_String aia_operand_const_string_get_val(Aia_Operand *op)
197 {
198     assert(op->op_type == AIA_OPERAND_CONST_STRING);
199     return op->iden.op_name;
200 }
201
202 static inline Const_String aia_operand_reg_get_name(Aia_Operand *op)
203 {
204     assert(op->op_type == AIA_OPERAND_REG);
205     return op->iden.op_name;
206 }
207
208 static inline int32_t aia_operand_const_int_get_val(Aia_Operand *op)
209 {
210     assert(op->op_type == AIA_OPERAND_CONST_INT);
211     return op->iden.int_const;
212 }
213
214 static inline Aia_Operand *aia_operand_addr_ref_get_label(Aia_Operand *op)
215 {
216     assert(aia_operand_get_type(op) == AIA_OPERAND_ADDR_REF);
217     return op->iden.addr_ref->label;
218 }
219

```

```

220 static inline Aia_Operand *aia_operand_addr_ref_get_disp(Aia_Operand *op)
221 {
222     assert(aia_operand_get_type(op) == AIA_OPERAND_ADDR_REF);
223     return op->iden.addr_ref->disp;
224 }
225
226 static inline Aia_Operand *aia_operand_addr_ref_get_base(Aia_Operand *op)
227 {
228     assert(aia_operand_get_type(op) == AIA_OPERAND_ADDR_REF);
229     return op->iden.addr_ref->base;
230 }
231
232 static inline Aia_Operand *aia_operand_addr_ref_get_index(Aia_Operand *op)
233 {
234     assert(aia_operand_get_type(op) == AIA_OPERAND_ADDR_REF);
235     return op->iden.addr_ref->index;
236 }
237
238 static inline Aia_Operand *aia_operand_addr_ref_get_scale(Aia_Operand *op)
239 {
240     assert(aia_operand_get_type(op) == AIA_OPERAND_ADDR_REF);
241     return op->iden.addr_ref->scale;
242 }
243
244 Aia_Operand *aia_operand_addr_ref_with_replaced_index(Aia *aia,
245     Aia_Operand *addr_ref, Aia_Operand *new_op);
246
247 Aia_Operand *aia_operand_addr_ref_with_replaced_base(Aia *aia,
248     Aia_Operand *addr_ref, Aia_Operand *new_op);
249
250 static inline Const_String aia_operand_label_get_name(Aia_Operand *lblop)
251 {
252     assert(lblop->op_type == AIA_OPERAND_LABEL || lblop->op_type ==
253         AIA_OPERAND_LABEL_ADDR);
254     return lblop->iden.op_label->label_name;
255 }
256
257 static inline Int aia_operand_label_get_offset(Aia_Operand *lblop)
258 {
259     assert(lblop->op_type == AIA_OPERAND_LABEL || lblop->op_type ==
260         AIA_OPERAND_LABEL_ADDR);
261     return lblop->iden.op_label->offset;
262 }
263
264 void aia_operand_dump_size(FILE *stream, uint8_t type);
265
266 void aia_operand_dump(FILE *stream, Aia_Operand *op,
267     bool print_int_dollar);
268
269 static inline Uns aia_operand_hash_code(Aia_Operand *op);
270
271 static inline Uns aia_operand_reg_hash_code(Aia_Operand *op)
272 {
273     return string_hash_code(aia_operand_reg_get_name(op));
274 }
275
276 static inline Uns aia_operand_label_hash_code(Aia_Operand *op)
277 {
278     return string_hash_code(aia_operand_label_get_name(op));
279 }
280
281 static inline Uns aia_operand_const_int_hash_code(Aia_Operand *op)
282 {
283     return (Uns)aia_operand_const_int_get_val(op);
284 }
285
286 static inline Uns aia_operand_const_string_hash_code(Aia_Operand *op)
287 {
288     return string_hash_code(aia_operand_const_string_get_val(op));
289 }
290
291 static inline Uns aia_operand_addr_ref_hash_code(Aia_Operand *op)
292 {
293     Uns hash = 0;

```

```

294
295 inline void add_hash_code(Aia_Operand *ref_op)
296 {
297     if (ref_op)
298         hash += aia_operand_hash_code(ref_op);
299 }
300
301 add_hash_code(aia_operand_addr_ref_get_base(op));
302 add_hash_code(aia_operand_addr_ref_get_index(op));
303 add_hash_code(aia_operand_addr_ref_get_disp(op));
304 add_hash_code(aia_operand_addr_ref_get_label(op));
305 add_hash_code(aia_operand_addr_ref_get_scale(op));
306
307 return hash;
308 }
309
310 static inline Uns aia_operand_local_ref_hash_code(Aia_Operand *op)
311 {
312     return string_hash_code(aia_operand_local_ref_get_var_name(op));
313 }
314
315 static inline Uns aia_operand_arg_hash_code(Aia_Operand *op)
316 {
317     return aia_operand_arg_get_idx(op);
318 }
319
320 static inline Uns aia_operand_display_ref_hash_code(Aia_Operand *op)
321 {
322     return string_hash_code(aia_operand_display_ref_get_var_name(op)) +
323         string_hash_code(aia_operand_display_ref_get_func_name(op));
324 }
325
326 Uns aia_operand_block_hash_code(Aia_Operand *op);
327
328 static inline Uns aia_operand_hash_code(Aia_Operand *op)
329 {
330     switch (op->op_type) {
331     case AIA_OPERAND_REG:
332         return aia_operand_reg_hash_code(op);
333     case AIA_OPERAND_LABEL:
334         return aia_operand_label_hash_code(op);
335     case AIA_OPERAND_LABEL_ADDR:
336         return aia_operand_label_hash_code(op);
337     case AIA_OPERAND_CONST_INT:
338         return aia_operand_const_int_hash_code(op);
339     case AIA_OPERAND_CONST_STRING:
340         return aia_operand_const_string_hash_code(op);
341     case AIA_OPERAND_ADDR_REF:
342         return aia_operand_addr_ref_hash_code(op);
343     case AIA_OPERAND_LOCAL_REF:
344         return aia_operand_local_ref_hash_code(op);
345     case AIA_OPERAND_ARG:
346         return aia_operand_arg_hash_code(op);
347     case AIA_OPERAND_DISPLAY_REF:
348         return aia_operand_display_ref_hash_code(op);
349     case AIA_OPERAND_BLOCK:
350         return aia_operand_block_hash_code(op);
351     }
352     fatal_error(S("taking hash code of unexpected operand type. "
353                 "Aborting...\n"));
354 }
355
356 #endif // AIA_OPERAND_H

```

:

A.7.11 src/aia/aia_operand_map.c

```

1 #include "aia_operand_map.h"
2
3 void aia_operand_entry_hash_destroy(Hash_Map_Slot *slot)

```

```

4  {
5      aia_operand_entry_destroy(AIA_OPERAND_MAP_ENTRY_OF(slot));
6  }
7
8  bool aia_operand_map_compare(Aia_Operand *lookup, Hash_Map_Slot *search_slot)
9  {
10     Aia_Operand_Map_Entry *e = AIA_OPERAND_MAP_ENTRY_OF(search_slot);
11     return aia_operands_equal(e->operand, lookup);
12 }

```

:

A.7.12 src/aia/aia_operand_map.h

```

1  #ifndef AIA_OPERAND_MAP_H
2  #define AIA_OPERAND_MAP_H
3
4  #include "aia.h"
5  #include <hash_map.h>
6
7  typedef Hash_Map Aia_Operand_Map;
8
9  typedef struct Aia_Operand_Map_Entry {
10     Aia_Operand *operand;
11     Aia_Operand *value;
12     Hash_Map_Slot hash_slot;
13 } Aia_Operand_Map_Entry;
14
15 #define AIA_OPERAND_MAP_ENTRY_OF(slot) \
16     HASH_MAP_ENTRY(slot, Aia_Operand_Map_Entry, hash_slot)
17
18 bool aia_operand_map_compare(Aia_Operand *lookup, Hash_Map_Slot *search_slot);
19
20 static inline void aia_operand_entry_destroy(Aia_Operand_Map_Entry *e)
21 {
22     __aia_operand_release(e->operand);
23     __aia_operand_release(e->value);
24     free_mem(e);
25 }
26
27 /* Use as dtor argument to AIA_OPERAND_MAP_INIT* see aia_operand_map.c */
28 void aia_operand_entry_hash_destroy(Hash_Map_Slot *slot);
29
30 #define AIA_OPERAND_MAP_INIT() \
31     HASH_MAP_INIT((Hash_Map_Comparator) aia_operand_map_compare)
32
33 #define AIA_OPERAND_MAP_FOR_EACH_ENTRY(opmap, ent) \
34     for (Hash_Map_Slot *___s = NULL; !___s; ___s = INT_TO_PTR(1)) \
35         HASH_MAP_FOR_EACH(opmap, ___s) \
36             if ((ent = AIA_OPERAND_MAP_ENTRY_OF(___s)) || !ent)
37
38 static inline Aia_Operand_Map *aia_operand_map_alloc()
39 {
40     Aia_Operand_Map *ret = ALLOC_NEW(Aia_Operand_Map);
41     *ret = AIA_OPERAND_MAP_INIT();
42     return ret;
43 }
44
45 static inline void aia_operand_map_insert(Aia_Operand_Map *map,
46     Aia_Operand *op, Aia_Operand *value)
47 {
48     Aia_Operand_Map_Entry *e = ALLOC_NEW(Aia_Operand_Map_Entry);
49     __aia_operand_acquire(op);
50     __aia_operand_acquire(value);
51     e->operand = op;
52     e->value = value;
53     hash_map_insert(map, &e->hash_slot, aia_operand_hash_code(op));
54 }
55
56 static inline bool aia_operand_map_contains(Aia_Operand_Map *map,
57     Aia_Operand *op)

```

```

58 {
59     return hash_map_contains(map, op, aia_operand_hash_code(op));
60 }
61
62 static inline Aia_Operand *aia_operand_map_get_value(Aia_Operand_Map *map,
63     Aia_Operand *op)
64 {
65     Hash_Map_Slot *slot = hash_map_get(map, op,
66         aia_operand_hash_code(op));
67     if (!slot)
68         return NULL;
69     Aia_Operand_Map_Entry *e = AIA_OPERAND_MAP_ENTRY_OF(slot);
70     return e->value;
71 }
72
73 static inline void aia_operand_map_for_each_destroy(Aia_Operand_Map *map)
74 {
75     hash_map_for_each_destroy(map, aia_operand_entry_hash_destroy);
76 }
77
78 static inline bool aia_operand_map_is_empty(Aia_Operand_Map *map)
79 {
80     return !hash_map_size(map);
81 }
82
83 static inline Aia_Operand_Map_Entry *__aia_operand_map_remove(
84     Aia_Operand_Map *map, Aia_Operand *op)
85 {
86     Hash_Map_Slot *slot = hash_map_remove(map, op,
87         aia_operand_hash_code(op));
88     if (!slot)
89         return NULL;
90     return AIA_OPERAND_MAP_ENTRY_OF(slot);
91 }
92
93 static inline bool aia_operand_map_remove(Aia_Operand_Map *map,
94     Aia_Operand *op)
95 {
96     Aia_Operand_Map_Entry *e = __aia_operand_map_remove(map, op);
97     if (e) {
98         aia_operand_entry_destroy(e);
99         return true;
100     }
101     return false;
102 }
103
104 static inline void aia_operand_map_destroy(Aia_Operand_Map *map)
105 {
106     if (map) {
107         aia_operand_map_for_each_destroy(map);
108         free_mem(map);
109     }
110 }
111
112 #endif // AIA_OPERAND_MAP_H

```

:

A.7.13 src/aia/aia_operand_set.c

```

1 #include "aia_operand_set.h"
2
3 void aia_operand_set_entry_hash_destroy(Hash_Map_Slot *slot)
4 {
5     aia_operand_set_entry_destroy(AIA_OPERAND_SET_ENTRY_OF(slot));
6 }
7
8 bool aia_operand_set_compare(Aia_Operand *lookup, Hash_Map_Slot *search_slot)
9 {
10     Aia_Operand_Set_Entry *e = AIA_OPERAND_SET_ENTRY_OF(search_slot);
11     return aia_operands_equal(e->operand, lookup);

```

12 }

:

A.7.14 src/aia/aia_operand_set.h

```

1  #ifndef AIA_OPERAND_SET_H
2  #define AIA_OPERAND_SET_H
3
4  #include "aia.h"
5  #include <hash_map.h>
6
7  typedef Hash_Map Aia_Operand_Set;
8
9  typedef struct Aia_Operand_Set_Entry {
10     Aia_Operand *operand;
11     Hash_Map_Slot hash_slot;
12 } Aia_Operand_Set_Entry;
13
14 #define AIA_OPERAND_SET_ENTRY_OF(slot) \
15     HASH_MAP_ENTRY(slot, Aia_Operand_Set_Entry, hash_slot)
16
17 bool aia_operand_set_compare(Aia_Operand *lookup, Hash_Map_Slot *search_slot);
18
19 static inline void aia_operand_set_entry_destroy(Aia_Operand_Set_Entry *e)
20 {
21     __aia_operand_release(e->operand);
22     free_mem(e);
23 }
24
25 void aia_operand_set_entry_hash_destroy(Hash_Map_Slot *slot);
26
27 #define AIA_OPERAND_SET_INIT() \
28     HASH_MAP_INIT((Hash_Map_Comparator) aia_operand_set_compare)
29
30 #define AIA_OPERAND_SET_FOR_EACH_OPERAND(opmap, op) \
31     for (Hash_Map_Slot *___s = NULL; !___s; ___s = INT_TO_PTR(1)) \
32         HASH_MAP_FOR_EACH(opmap, ___s) \
33             if ((op = AIA_OPERAND_SET_ENTRY_OF(___s)->operand) || !op)
34
35 static inline Aia_Operand_Set *aia_operand_set_alloc()
36 {
37     Aia_Operand_Set *ret = ALLOC_NEW(Aia_Operand_Set);
38     *ret = AIA_OPERAND_SET_INIT();
39     return ret;
40 }
41
42 static inline bool aia_operand_set_insert(Aia_Operand_Set *set,
43     Aia_Operand *op)
44 {
45     Uns hash = aia_operand_hash_code(op);
46     if (hash_map_contains(set, op, hash))
47         return false;
48
49     Aia_Operand_Set_Entry *e = ALLOC_NEW(Aia_Operand_Set_Entry);
50     __aia_operand_acquire(op);
51     e->operand = op;
52     hash_map_insert(set, &e->hash_slot, hash);
53     return true;
54 }
55
56 static inline bool aia_operand_set_contains(Aia_Operand_Set *set,
57     Aia_Operand *op)
58 {
59     return hash_map_contains(set, op, aia_operand_hash_code(op));
60 }
61
62 static inline void aia_operand_set_for_each_destroy(Aia_Operand_Set *set)
63 {
64     hash_map_for_each_destroy(set, aia_operand_set_entry_hash_destroy);
65 }

```

```

66
67 static inline bool aia_operand_set_is_empty(Aia_Operand_Set *set)
68 {
69     return !hash_map_size(set);
70 }
71
72 static inline Aia_Operand_Set_Entry *__aia_operand_set_remove(
73     Aia_Operand_Set *set, Aia_Operand *op)
74 {
75     Hash_Map_Slot *slot = hash_map_remove(set, op,
76     aia_operand_hash_code(op));
77     if (!slot)
78         return NULL;
79     return AIA_OPERAND_SET_ENTRY_OF(slot);
80 }
81
82 static inline bool aia_operand_set_remove(Aia_Operand_Set *set,
83     Aia_Operand *op)
84 {
85     Aia_Operand_Set_Entry *e = __aia_operand_set_remove(set, op);
86     if (e) {
87         aia_operand_set_entry_destroy(e);
88         return true;
89     }
90     return false;
91 }
92
93 static inline void aia_operand_set_destroy(Aia_Operand_Set *set)
94 {
95     if (set) {
96         aia_operand_set_for_each_destroy(set);
97         free_mem(set);
98     }
99 }
100
101
102 #endif // AIA_OPERAND_SET_H

```

:

A.7.15 src/aia/aia_warn_undefined.c

```

1 #include "aia_warn_undefined.h"
2 #include "aia_instr_live_sets.h"
3 #include <main.h>
4
5 static inline bool aia_warn_undefined_is_param(Aia_Operand *op,
6     Aia_Func *func)
7 {
8     Vector *params = &func->parameters;
9     Const_String name = aia_operand_local_ref_get_var_name(op);
10    if (vector_contains(params, (Vector_Comparator)string_compare, name))
11        return true;
12    return false;
13 }
14
15 static void aia_warn_undefined_do(Aia_Operand *op, Aia_Func *func)
16 {
17     Const_String loc_name = aia_operand_local_ref_get_var_name(op);
18     report_warning_location(aia_func_get_location(func),
19     S("varibale " QFY("%S") " declared in function "
20     QFY("%S") " might be uninitialized before use\n"),
21     STRING_AFTER_DOT(loc_name),
22     STRING_AFTER_DOT(aia_func_get_name(func)));
23 }
24
25 static void aia_warn_undefined_local(Aia_Func *func)
26 {
27     Aia_Instr *in = __aia_block_peek_first_instr(func->entry_block);
28     Aia_Operand *op;
29

```

```

30     AIA_INSTR_LIVE_SET_FOR_EACH_OP(in, op) {
31         if (aia_operand_get_type(op) == AIA_OPERAND_LOCAL_REF &&
32             !aia_warn_undefined_is_param(op, func))
33             aia_warn_undefined_do(op, func);
34     }
35 }
36
37 void aia_warn_undefined(Aia *aia)
38 {
39     if (!cmdopts.warn_uninitialized)
40         return;
41
42     aia_instr_live_sets(aia, NULL, true);
43
44     if (cmdopts.dump_warn_uninit_liveness_ic)
45         aia_instr_live_sets_dump(aia, S("warn-uninit-liveness-ic"));
46
47     Aia_Func *func;
48     AIA_FOR_EACH_FUNC(aia, func)
49         aia_warn_undefined_local(func);
50
51     aia_instr_live_sets_destroy(aia);
52 }

```

:

A.7.16 src/aia/aia_warn_undefined.h

```

1  #ifndef AIA_WARN_UNDEFINED_H
2  #define AIA_WARN_UNDEFINED_H
3
4  #include "aia.h"
5
6  void aia_warn_undefined(Aia *aia);
7
8  #endif // AIA_WARN_UNDEFINED_H

```

A.8 Optimization

:

A.8.1 src/aia/aia_block_elim.c

```

1  #include "aia_block_elim.h"
2  #include <main.h>
3
4  #undef DEBUG_TYPE
5  #define DEBUG_TYPE block-elim
6
7  #define BLOCK_DEAD INT_TO_PTR(0)
8  #define BLOCK_LIVE INT_TO_PTR(1)
9
10 static void aia_live_block_callback(Aia_Block *b, void *arg UNUSED)
11 {
12     DEBUG(
13         Aia_Instr *lbl = __aia_block_peek_first_instr(b);
14         Aia_Operand *lbl_op = aia_instr_get_dest_op(lbl);
15         LOG("live block: ");
16         aia_operand_dump(stderr, lbl_op, false);
17         LOG("\n");
18     );
19     b->meta_data = BLOCK_LIVE;
20 }
21
22 static void aia_section_dead_blocks(Aia_Section *sec, Aia *aia)

```

```

23 {
24     VECTOR(dead_blocks);
25
26     inline void add_unused_block(Aia_Block *b)
27     {
28         if (b->meta_data == BLOCK_DEAD) {
29             vector_append(&dead_blocks, b);
30             DEBUG(
31                 Aia_Instr *lbl = __aia_block_peek_first_instr(b);
32                 Aia_Operand *lbl_op = aia_instr_get_dest_op(lbl);
33                 DLOG("dead block: ");
34                 aia_operand_dump(stderr, lbl_op, false);
35                 DLOG("\n");
36             );
37         }
38     }
39
40     inline void remove_dead_blocks(Aia_Block *entry_b, Aia_Block *exit_b)
41     {
42         Aia_Block *b;
43         VECTOR_FOR_EACH_ENTRY(&dead_blocks, b) {
44             if (b != entry_b && b != exit_b) {
45                 aia_block_remove_from_predecessors(b);
46                 aia->meta_data = INT_TO_PTR(true);
47             }
48         }
49
50         VECTOR_FOR_EACH_ENTRY(&dead_blocks, b) {
51             if (b != entry_b && b != exit_b)
52                 aia_block_blist_remove_destroy(b);
53         }
54         vector_clear(&dead_blocks);
55     }
56
57     Aia_Block *b;
58     AIA_SECTION_FOR_EACH_BLOCK(sec, b)
59         b->meta_data = BLOCK_DEAD;
60
61     aia_section_for_each_block_depth(sec, aia_live_block_callback, aia);
62
63     AIA_SECTION_FOR_EACH_BLOCK(sec, b)
64         add_unused_block(b);
65
66     remove_dead_blocks(sec->exit_block, sec->exit_block);
67
68     Aia_Func *func;
69     AIA_SECTION_FOR_EACH_FUNC(sec, func) {
70         AIA_FUNC_FOR_EACH_BLOCK(func, b) {
71             b->meta_data = BLOCK_DEAD;
72             DEBUG(
73                 Aia_Instr *lbl = __aia_block_peek_first_instr(b);
74                 Aia_Operand *lbl_op = aia_instr_get_dest_op(lbl);
75                 DLOG("existing block: ");
76                 aia_operand_dump(stderr, lbl_op, false);
77                 DLOG("\n");
78             );
79         }
80
81         aia_func_for_each_block_depth(func, aia_live_block_callback, aia);
82
83         AIA_FUNC_FOR_EACH_BLOCK(func, b) {
84             add_unused_block(b);
85             DEBUG(
86                 Aia_Instr *lbl = __aia_block_peek_first_instr(b);
87                 Aia_Operand *lbl_op = aia_instr_get_dest_op(lbl);
88                 DLOG("2 existing block: ");
89                 aia_operand_dump(stderr, lbl_op, false);
90                 DLOG("\n");
91             );
92         }
93         remove_dead_blocks(func->exit_block, func->exit_block);
94     }
95 }
96 }

```

```

97
98 static void aia_elim_dead_blocks(Aia *aia)
99 {
100     Aia_Section *sec;
101     AIA_FOR_EACH_SECTION(aia, sec)
102         aia_section_dead_blocks(sec, aia);
103 }
104
105 static UNUSED void aia_block_elim_dump(Aia *aia, Const_String post)
106 {
107     String fname = string_from_format(S("%S.vitaly.block-elim-ic-%S"),
108         aia_get_file_name(aia), post);
109     FILE *f = file_open(fname, S("w"));
110     if (!f)
111         fatal_error(S("cannot open file " QFY("%S")
112             " for code elimination dump [%m]\n"),
113             fname);
114     string_destroy(fname);
115
116     aia_dump(aia, f);
117     file_close(f);
118 }
119
120 bool aia_block_elim(Aia *aia)
121 {
122     bool ret = false;
123
124     do {
125         aia->meta_data = NULL;
126         aia_elim_dead_blocks(aia);
127         if (aia->meta_data)
128             ret = true;
129     } while (aia->meta_data);
130
131     return ret;
132 }

```

:

A.8.2 src/aia/aia_block_elim.h

```

1 #ifndef AIA_BLOCK_ELIM_H
2 #define AIA_BLOCK_ELIM_H
3
4 #include "aia.h"
5
6 bool aia_block_elim(Aia *aia);
7
8 #endif // AIA_BLOCK_ELIM_H

```

:

A.8.3 src/aia/aia_const_prop.c

```

1 #include "aia_const_prop.h"
2 #include "aia_block_elim.h"
3 #include "aia_func_kills.h"
4 #include <main.h>
5
6 #undef DEBUG_TYPE
7 #define DEBUG_TYPE const-prop
8
9 static void const_opentry_insert(Aia_Const_Prop_Bmeta *bmeta,
10     Aia_Operand *op, Aia_Operand *value)
11 {
12     Aia_Operand_Map *map = &bmeta->const_ops;
13     aia_operand_map_insert(map, op, value);

```

```

14 }
15
16 static Aia_Operand *const_opentry_lookup_val(Aia_Const_Prop_Bmeta *bmeta,
17 Aia_Operand *op)
18 {
19     if (aia_operand_get_type(op) == AIA_OPERAND_CONST_INT)
20         return op;
21
22     Aia_Operand_Map *map = &bmeta->const_ops;
23
24     return aia_operand_map_get_value(map, op);
25 }
26
27 static void const_opentry_remove(Aia_Const_Prop_Bmeta *bmeta, Aia_Operand *op)
28 {
29     if (!op)
30         return;
31
32     Aia_Operand_Map *map = &bmeta->const_ops;
33     aia_operand_map_remove(map, op);
34 }
35
36 static void aia_const_prop_remove_instr(Aia_Instr *in,
37 Aia_Const_Prop_Bmeta *bmeta)
38 {
39     aia_instr_remove(in);
40     vector_append(&bmeta->removed_instrs, in);
41 }
42
43 static void aia_const_prop_mov(Aia_Instr *in,
44 Aia_Const_Prop_Bmeta *bmeta, Aia_Const_Prop_Ameta *ameta)
45 {
46     Aia_Operand *dest = aia_instr_get_dest_op(in);
47     Aia_Operand *src = aia_instr_get_src_op(in, 0);
48     Aia_Operand *val = const_opentry_lookup_val(bmeta, src);
49     if (val) {
50         DEBUG(
51             DLOGT(def, "value: ");
52             aia_operand_dump(stderr, val, false);
53             DLOGT(def, "\n");
54         );
55
56         Aia_Operand *dest_val = const_opentry_lookup_val(bmeta, dest);
57         const_opentry_remove(bmeta, dest);
58         if (dest_val && aia_operands_equal(dest_val, val)) {
59             switch (aia_operand_get_type(dest)) {
60                 case AIA_OPERAND_ARG:
61                     break;
62
63                 default:
64                     aia_const_prop_remove_instr(in, bmeta);
65                     return;
66             }
67         }
68         assert(!aia_operand_map_contains(&bmeta->const_ops, dest));
69         DEBUG(
70             DLOGT(def, "new dest: ");
71             aia_operand_dump(stderr, dest, false);
72             DLOGT(def, "\n");
73         );
74         const_opentry_insert(bmeta, dest, val);
75
76         if (aia_operand_get_type(src) != AIA_OPERAND_CONST_INT) {
77             aia_type_mov_before(aia_instr_get_type(in), val, dest, in,
78                 aia_instr_get_src_ops_size(in),
79                 aia_instr_get_dest_op_size(in));
80             aia_const_prop_remove_instr(in, bmeta);
81             ameta->any_update = true;
82         }
83     }
84     else {
85         const_opentry_remove(bmeta, dest);
86     }
87 }

```

```

88
89 static void aia_const_prop_2op_arith(Aia_Instr *in, Aia *aia,
90     Aia_Const_Prop_Bmeta *bmeta, Aia_Const_Prop_Ameta *ameta)
91 {
92     Aia_Operand *src = aia_instr_get_src_op(in, 0);
93     Aia_Operand *lhs = const_opentry_lookup_val(bmeta, src);
94
95     if (lhs) {
96         Aia_Operand *src_val = const_opentry_lookup_val(bmeta, src);
97         if (src_val && aia_operands_equal(lhs, src_val))
98             aia_instr_replace_op(in, 0, src_val);
99     }
100
101     src = aia_instr_get_src_op(in, 1);
102     Aia_Operand *rhs = const_opentry_lookup_val(bmeta, src);
103
104     if (rhs) {
105         Aia_Operand *src_val = const_opentry_lookup_val(bmeta, src);
106         if (src_val && aia_operands_equal(rhs, src_val))
107             aia_instr_replace_op(in, 1, src_val);
108     }
109
110     if (!rhs || !lhs)
111         return;
112
113     int32_t lhs_val = aia_operand_const_int_get_val(lhs);
114     int32_t rhs_val = aia_operand_const_int_get_val(rhs);
115     int32_t result;
116     switch (aia_instr_get_type(in)) {
117     case AIA_ADD:
118         result = lhs_val + rhs_val;
119         break;
120     case AIA_SUB:
121         result = lhs_val - rhs_val;
122         break;
123     case AIA_IMUL:
124         result = lhs_val * rhs_val;
125         break;
126     case AIA_IDIV:
127         if (rhs_val == 0) {
128             File_Location *loc = aia_instr_get_location(in);
129             if (cmdopts.warn_div_zero && !is_warning_reported_here(loc))
130                 report_warning_location(loc, S("division by zero\n"));
131             return;
132         }
133         result = lhs_val / rhs_val;
134         break;
135     default:
136         assert(false);
137         return;
138     }
139
140     Aia_Operand *res_op = aia_operand_const_int_alloc(aia, result);
141     Aia_Operand *dest = aia_instr_get_dest_op(in);
142     aia_mov_before(res_op, dest, in, aia_instr_get_dest_op_size(in));
143
144     const_opentry_insert(bmeta, dest, res_op);
145     aia_const_prop_remove_instr(in, bmeta);
146     ameta->any_update = true;
147 }
148
149 static void aia_const_prop_cmp(Aia_Instr *in, Aia_Const_Prop_Bmeta *bmeta,
150     Aia_Const_Prop_Ameta *ameta)
151 {
152     Aia_Cmp_Result res = AIA_CMP_NONE;
153
154     Aia_Operand *lhs = const_opentry_lookup_val(bmeta,
155         aia_instr_get_src_op(in, 0));
156     if (!lhs)
157         goto out;
158     aia_instr_replace_op(in, 0, lhs);
159
160     Aia_Operand *rhs = const_opentry_lookup_val(bmeta,

```

```

162     aia_instr_get_src_op(in, 1));
163     if (!rhs)
164         goto out;
165     aia_instr_replace_op(in, 1, rhs);
166
167     int32_t lhs_val = aia_operand_const_int_get_val(lhs);
168     int32_t rhs_val = aia_operand_const_int_get_val(rhs);
169
170     if (lhs_val < rhs_val)
171         res = AIA_CMP_LESS;
172     else if (lhs_val > rhs_val)
173         res = AIA_CMP_GREAT;
174     else
175         res = AIA_CMP_EQUAL;
176
177     aia_instr_remove(in);
178     vector_append(&bmeta->removed_instrs, in);
179
180 out:
181     ameta->prev_cmp_result = res;
182 }
183
184 static void aia_const_prop_cond_jump(Aia_Instr *in,
185     Aia_Const_Prop_Bmeta *bmeta, Aia_Const_Prop_Ameta *ameta)
186 {
187     if (ameta->prev_cmp_result == AIA_CMP_NONE ||
188         ameta->prev_cmp_result == AIA_CMP_UNKNOWN)
189         return;
190
191     Aia_Block *b_in = aia_instr_get_block(in);
192
193     Aia_Instr *jmp_in = aia_instr_alloc_lol(AIA_JMP, b_in, -1, AIA_LONG,
194         aia_instr_get_location(in));
195
196     switch (aia_instr_get_type(in)) {
197     case AIA_JG:
198         if (ameta->prev_cmp_result == AIA_CMP_GREAT)
199             aia_instr_set_src_op(jmp_in, 0, aia_instr_get_src_op(in, 0));
200         else
201             aia_instr_set_src_op(jmp_in, 0, aia_instr_get_src_op(in, 1));
202         break;
203
204     case AIA_JLE:
205         if (ameta->prev_cmp_result == AIA_CMP_GREAT)
206             aia_instr_set_src_op(jmp_in, 0, aia_instr_get_src_op(in, 1));
207         else
208             aia_instr_set_src_op(jmp_in, 0, aia_instr_get_src_op(in, 0));
209         break;
210
211     case AIA_JL:
212         if (ameta->prev_cmp_result == AIA_CMP_LESS)
213             aia_instr_set_src_op(jmp_in, 0, aia_instr_get_src_op(in, 0));
214         else
215             aia_instr_set_src_op(jmp_in, 0, aia_instr_get_src_op(in, 1));
216         break;
217
218     case AIA_JGE:
219         if (ameta->prev_cmp_result == AIA_CMP_LESS)
220             aia_instr_set_src_op(jmp_in, 0, aia_instr_get_src_op(in, 1));
221         else
222             aia_instr_set_src_op(jmp_in, 0, aia_instr_get_src_op(in, 0));
223         break;
224
225     case AIA_JE:
226         if (ameta->prev_cmp_result == AIA_CMP_EQUAL)
227             aia_instr_set_src_op(jmp_in, 0, aia_instr_get_src_op(in, 0));
228         else
229             aia_instr_set_src_op(jmp_in, 0, aia_instr_get_src_op(in, 1));
230         break;
231
232     case AIA_JNE:
233         if (ameta->prev_cmp_result == AIA_CMP_EQUAL)
234             aia_instr_set_src_op(jmp_in, 0, aia_instr_get_src_op(in, 1));
235         else

```

```

236     aia_instr_set_src_op(jmp_in, 0, aia_instr_get_src_op(in, 0));
237     break;
238
239     default:
240         assert(false);
241     }
242
243     aia_instr_cond_jump_to_jump(in, jmp_in, aia_instr_get_src_op(jmp_in, 0));
244     vector_append(&bmeta->removed_instrs, in);
245
246     ameta->any_update = true;
247 }
248
249 static void aia_const_prop_set(Aia_Instr *in, Aia *aia,
250     Aia_Const_Prop_Bmeta *bmeta, Aia_Const_Prop_Ameta *ameta)
251 {
252     if (ameta->prev_cmp_result == AIA_CMP_NONE ||
253         ameta->prev_cmp_result == AIA_CMP_UNKNOWN)
254         return;
255
256     int32_t result;
257
258     switch (aia_instr_get_type(in)) {
259     case AIA_SETL:
260         if (ameta->prev_cmp_result == AIA_CMP_LESS)
261             result = 1;
262         else
263             result = 0;
264         break;
265
266     case AIA_SETGE:
267         if (ameta->prev_cmp_result == AIA_CMP_LESS)
268             result = 0;
269         else
270             result = 1;
271         break;
272
273     case AIA_SETG:
274         if (ameta->prev_cmp_result == AIA_CMP_GREAT)
275             result = 1;
276         else
277             result = 0;
278         break;
279
280     case AIA_SETLE:
281         if (ameta->prev_cmp_result == AIA_CMP_GREAT)
282             result = 0;
283         else
284             result = 1;
285         break;
286
287     case AIA_SETGE:
288         if (ameta->prev_cmp_result == AIA_CMP_EQUAL)
289             result = 1;
290         else
291             result = 0;
292         break;
293
294     case AIA_SETNE:
295         if (ameta->prev_cmp_result == AIA_CMP_EQUAL)
296             result = 0;
297         else
298             result = 1;
299         break;
300
301     default:
302         assert(false);
303     }
304
305     Aia_Operand *const_in = aia_operand_const_int_alloc(aia, result);
306     aia_mov_before(const_in, aia_instr_get_dest_op(in), in, AIA_BYTE);
307
308     aia_instr_remove(in);
309     vector_append(&bmeta->removed_instrs, in);

```

```

310     ameta->any_update = true;
311 }
312 }
313
314 static void aia_const_prop_ret(Aia_Instr *in, Aia_Const_Prop_Bmeta *bmeta)
315 {
316     if (!aia_instr_get_src_op_count(in))
317         return;
318
319     Aia_Operand *src = aia_instr_get_src_op(in, 0);
320     if (aia_operand_get_type(src) == AIA_OPERAND_CONST_INT)
321         return;
322
323     Aia_Operand *val = const_opentry_lookup_val(bmeta, src);
324     if (!val)
325         return;
326
327     aia_instr_replace_op(in, 0, val);
328     /* No need to set aia->meta_data->any_update */
329 }
330
331 static void aia_const_prop_neg(Aia_Instr *in, Aia *aia,
332     Aia_Const_Prop_Bmeta *bmeta, Aia_Const_Prop_Ameta *ameta)
333 {
334     Aia_Operand *val = const_opentry_lookup_val(bmeta,
335         aia_instr_get_src_op(in, 0));
336     if (!val)
337         return;
338
339     Aia_Operand *new_val = aia_operand_const_int_alloc(aia,
340         -aia_operand_const_int_get_val(val));
341     Aia_Operand *dest = aia_instr_get_dest_op(in);
342
343     aia_mov_before(new_val, dest, in, aia_instr_get_dest_op_size(in));
344
345     aia_instr_remove(in);
346     vector_append(&bmeta->removed_instrs, in);
347
348     ameta->any_update = true;
349 }
350
351 static void aia_const_prop_call(Aia_Instr *in, Aia *aia,
352     Aia_Const_Prop_Bmeta *bmeta)
353 {
354     Const_String fname;
355     Aia_Func *curr_func = __aia_get_curr_func(aia);
356     Aia_Func *func = NULL;
357
358     VECTOR(delete_ops);
359
360     Aia_Operand *lbl_op = aia_instr_get_src_op(in, 0);
361     if (aia_operand_get_type(lbl_op) == AIA_OPERAND_LABEL) {
362         fname = aia_operand_label_get_name(lbl_op);
363         func = aia_func_lookup(aia, fname);
364     }
365
366     bool kills_global = false;
367
368     if (func) {
369         Func_Kill_Operand fkill_op = FUNC_KILL_OPERAND_INIT(
370             curr_func ? aia_func_get_name(curr_func) : NULL);
371
372         Const_String tmp_func_name = aia_func_get_name(func);
373         if (hash_map_contains(&aia->default_funcs, (String)tmp_func_name,
374             string_hash_code(tmp_func_name)))
375             return;
376
377         Aia_Func_Kill_Meta *fmeta = func->func_kills_struct;
378         Hash_Map *fmap = &fmeta->func_kills;
379
380         Hash_Map_Slot *slot;
381         HASH_MAP_FOR_EACH(&bmeta->const_ops, slot) {
382             Aia_Operand_Map_Entry *e = AIA_OPERAND_MAP_ENTRY_OF(slot);
383             func_kill_operand_set_operand(&fkill_op, e->operand);

```

```

384         if (!cmdopts.opt_func_access) {
385             if (aia_func_is_nested(func))
386                 vector_append(&delete_ops, e->operand);
387         } else if (___func_kill_contains(fmap, &fkill_op)) {
388             vector_append(&delete_ops, e->operand);
389         }
390     }
391
392     if (cmdopts.opt_func_access)
393         kills_global = fmeta->kills_global;
394     else
395         kills_global = true;
396 } else {
397     // All global variables are killed. No local variables are killed.
398     kills_global = true;
399 }
400
401 if (kills_global) {
402     Hash_Map_Slot *slot;
403     HASH_MAP_FOR_EACH(&bmeta->const_ops, slot) {
404         Aia_Operand_Map_Entry *e = AIA_OPERAND_MAP_ENTRY_OF(slot);
405         if (aia_operand_get_type(e->operand) == AIA_OPERAND_LABEL)
406             vector_append(&delete_ops, e->operand);
407     }
408 }
409
410 Aia_Operand *op;
411 VECTOR_FOR_EACH_ENTRY(&delete_ops, op)
412     const_opentry_remove(bmeta, op);
413 vector_clear(&delete_ops);
414 }
415
416 void aia_const_prop_instr(Aia_Instr *in, Aia *aia,
417     Aia_Const_Prop_Bmeta *bmeta, Aia_Const_Prop_Ameta *ameta)
418 {
419     switch (aia_instr_get_type(in)) {
420     case AIA_MOV:
421         /* Fall through. */
422     case AIA_MOVZ:
423         /* Fall through. */
424     case AIA_MOVS:
425         aia_const_prop_mov(in, bmeta, ameta);
426         //ameta->prev_cmp_result = AIA_CMP_NONE;
427         break;
428
429     case AIA_ADD:
430         /* Fall through. */
431     case AIA_SUB:
432         /* Fall through. */
433     case AIA_IMUL:
434         /* Fall through. */
435     case AIA_IDIV:
436         const_opentry_remove(bmeta, aia_instr_get_dest_op(in));
437         aia_const_prop_2op_arith(in, aia, bmeta, ameta);
438         ameta->prev_cmp_result = AIA_CMP_NONE;
439         break;
440
441     case AIA_CMP:
442         aia_const_prop_cmp(in, bmeta, ameta);
443         break;
444
445     AIA_CASE_COND_JUMP:
446         aia_const_prop_cond_jump(in, bmeta, ameta);
447         //ameta->prev_cmp_result = AIA_CMP_NONE;
448         break;
449
450     case AIA_JMP:
451         break;
452
453     case AIA_SETE:
454         /* Fall through. */
455     case AIA_SETNE:
456         /* Fall through. */
457     case AIA_SETL:

```



```

458     /* Fall through. */
459     case AIA_SETG:
460     /* Fall through. */
461     case AIA_SETLE:
462     /* Fall through. */
463     case AIA_SETGE:
464         aia_const_prop_set(in, aia, bmeta, ameta);
465         //ameta->prev_cmp_result = AIA_CMP_NONE;
466         break;
467
468     case AIA_RET:
469         const_opentry_remove(bmeta, aia_instr_get_dest_op(in));
470         aia_const_prop_ret(in, bmeta);
471         ameta->prev_cmp_result = AIA_CMP_NONE;
472         break;
473     case AIA_CALL:
474         const_opentry_remove(bmeta, aia_instr_get_dest_op(in));
475         aia_const_prop_call(in, aia, bmeta);
476         ameta->prev_cmp_result = AIA_CMP_NONE;
477         break;
478     case AIA_NEG:
479         const_opentry_remove(bmeta, aia_instr_get_dest_op(in));
480         aia_const_prop_neg(in, aia, bmeta, ameta);
481         ameta->prev_cmp_result = AIA_CMP_NONE;
482         break;
483     case AIA_CDQ:
484         const_opentry_remove(bmeta, aia_instr_get_dest_op(in));
485         DLOGT(def, "cdq needs constant propagation implementation\n");
486         ameta->prev_cmp_result = AIA_CMP_NONE;
487         break;
488
489     case AIA_NOP:
490     case __AIA_LABEL:
491     case __AIA_STRING:
492     case __AIA_INTEGER:
493         break;
494
495     default:
496         DLOGT(def, "unimplemented instruction for constant propagation\n");
497         break;
498 }
499 }
500
501 static void aia_block_meta_destroy(Aia_Block *b)
502 {
503     Aia_Const_Prop_Bmeta *bmeta = b->meta_data;
504     aia_operand_map_for_each_destroy(&bmeta->const_ops);
505     free_mem(bmeta);
506 }
507
508 static void aia_const_prop_get_pred_consts(Aia_Block *b)
509 {
510     Aia_Block *pred;
511
512     Uns num_preds = vector_size(&b->predecessors);
513     if (!num_preds)
514         return;
515
516     pred = vector_get(&b->predecessors, 0);
517     Aia_Const_Prop_Bmeta *pred_meta = pred->meta_data;
518
519     Hash_Map_Slot *slot;
520     HASH_MAP_FOR_EACH(&pred_meta->const_ops, slot) {
521         Aia_Operand_Map_Entry *e = AIA_OPERAND_MAP_ENTRY_OF(slot);
522
523         for (Uns i = 1; i < num_preds; i++) {
524             pred = vector_get(&b->predecessors, i);
525
526             Aia_Operand *tmp_val = const_opentry_lookup_val(pred->meta_data,
527                 e->operand);
528
529             if (!tmp_val)
530                 goto skip_op;
531             if (!aia_operands_equal(tmp_val, e->value))

```

```

532         goto skip_op;
533     }
534
535     const_opentry_insert(b->meta_data, e->operand, e->value);
536     skip_op++;
537 }
538 }
539
540 static void __aia_const_prop_block(Aia_Block *b, Aia *aia)
541 {
542     Aia_Const_Prop_Bmeta *bmeta = b->meta_data;
543
544     Aia_Instr *in;
545     AIA_BLOCK_FOR_EACH_INSTRUCTION(b, in)
546         aia_const_prop_instr(in, aia, bmeta, aia->meta_data);
547
548     vector_for_each_destroy(&bmeta->removed_instrs,
549         (Vector_Destructor)aia_instr_destroy);
550 }
551
552 static void aia_const_prop_block(Aia_Block *b, Aia *aia)
553 {
554     Aia_Const_Prop_Bmeta *bmeta = b->meta_data;
555     aia_operand_map_for_each_destroy(&bmeta->const_ops);
556
557     aia_const_prop_get_pred_consts(b);
558     __aia_const_prop_block(b, aia);
559 }
560
561 static void aia_const_prop_section_funcs(Aia_Section *sec, Aia *aia)
562 {
563     Aia_Func *func;
564     AIA_SECTION_FOR_EACH_FUNC(sec, func) {
565         __aia_set_curr_func(aia, func);
566         Aia_Block *b;
567         AIA_FUNC_FOR_EACH_BLOCK(func, b)
568             aia_const_prop_block(b, aia);
569
570         Aia_Func_trampoline *tramp;
571         AIA_FUNC_FOR_EACH_TRAMPOLINE(func, tramp)
572             aia_const_prop_block(tramp->block, aia);
573     }
574 }
575
576 static void aia_const_prop_append_data_section(Aia_Section *dsec,
577     Aia_Const_Prop_Bmeta *bmeta)
578 {
579     if (!dsec)
580         return;
581
582     Aia_Block *b;
583     AIA_SECTION_FOR_EACH_BLOCK(dsec, b) {
584         Aia_Instr *prev = NULL;
585         Aia_Instr *curr;
586         AIA_BLOCK_FOR_EACH_INSTRUCTION(b, curr) {
587             if (aia_instr_get_type(curr) == __AIA_INTEGER) {
588                 assert(aia_instr_get_type(prev) == __AIA_LABEL);
589                 Aia_Operand *op = aia_instr_get_src_op(curr, 0);
590                 if (aia_operand_get_type(op) == AIA_OPERAND_CONST_INT) {
591                     Aia_Operand *lbl = aia_instr_get_dest_op(prev);
592                     const_opentry_insert(bmeta, lbl, op);
593                 }
594             }
595             prev = curr;
596         }
597     }
598 }
599
600 static void aia_const_prop_section(Aia_Section *sec, Aia *aia)
601 {
602     __aia_set_curr_func(aia, NULL);
603
604     if (sec->sec_type == AIA_SECTION_INIT) {
605

```

```

606     Aia_Const_Prop_Bmeta *bmeta = sec->entry_block->meta_data;
607     aia_operand_map_for_each_destroy(&bmeta->const_ops);
608
609     aia_const_prop_append_data_section(
610         aia->sections[AIA_SECTION_DATA],
611         sec->entry_block->meta_data);
612
613     __aia_const_prop_block(sec->entry_block, aia);
614
615     Double_List_Node *bnode;
616     DOUBLE_LIST_FOR_EACH_AFTER(&sec->sec_blist,
617         &sec->entry_block->blist_node, bnode)
618         aia_const_prop_block(AIA_BLOCK_OF_DBNODE(bnode), aia);
619 } else {
620     Double_List_Node *bnode;
621     DOUBLE_LIST_FOR_EACH(&sec->sec_blist, bnode)
622         aia_const_prop_block(AIA_BLOCK_OF_DBNODE(bnode), aia);
623 }
624
625 aia_const_prop_section_funcs(sec, aia);
626 }
627
628 static void aia_const_prop_dump(Aia *aia, Int it_count)
629 {
630     String fname = string_from_format(S("%S.vitaly.const-prop-ic-%D"),
631         aia_get_file_name(aia), it_count);
632
633     FILE *f = file_open(fname, S("w"));
634     if (!f)
635         fatal_error(S("cannot open file " QFY("%S")
636             " for constant propagation dump [%m]\n"),
637             fname);
638     string_destroy(fname);
639
640     aia_dump(aia, f);
641     file_close(f);
642 }
643
644 static void aia_const_prop_init_blist(Double_List *blist)
645 {
646     Double_List_Node *bnode;
647     DOUBLE_LIST_FOR_EACH(blist, bnode) {
648         Aia_Block *b = AIA_BLOCK_OF_DBNODE(bnode);
649         b->meta_data = aia_const_prop_bmeta_alloc();
650     }
651 }
652
653 static void aia_const_prop_init(Aia *aia)
654 {
655     Aia_Section *sec;
656     AIA_FOR_EACH_SECTION(aia, sec)
657         aia_const_prop_init_blist(&sec->sec_blist);
658
659     Aia_Func *func;
660     AIA_FOR_EACH_FUNC(aia, func) {
661         aia_const_prop_init_blist(&func->blist);
662
663         Aia_Func_Trampoline *tramp;
664         AIA_FUNC_FOR_EACH_TRAMPOLINE(func, tramp)
665             aia_const_prop_init_blist(&tramp->blist);
666     }
667 }
668
669 static void aia_const_prop_destroy_blist(Double_List *blist)
670 {
671     Double_List_Node *bnode;
672     DOUBLE_LIST_FOR_EACH(blist, bnode)
673         aia_block_meta_destroy(AIA_BLOCK_OF_DBNODE(bnode));
674 }
675
676 static void aia_const_prop_destroy(Aia *aia)
677 {
678     Aia_Section *sec;
679     AIA_FOR_EACH_SECTION(aia, sec)

```

```

680     aia_const_prop_destroy_blist(&sec->sec_blist);
681
682     Aia_Func *func;
683     AIA_FOR_EACH_FUNC(aia, func) {
684         aia_const_prop_destroy_blist(&func->blist);
685
686         Aia_Func_trampoline *tramp;
687         AIA_FUNC_FOR_EACH_TRAMPOLINE(func, tramp)
688             aia_const_prop_destroy_blist(&tramp->blist);
689     }
690 }
691
692 bool aia_const_prop(Aia *aia, Int it_count)
693 {
694     bool ret = false;
695
696     if (!cmdopts.opt_const_prop)
697         goto out;
698
699     aia_collect_func_kills(aia);
700
701     VECTOR(globals);
702
703     bool any_update = false;
704     for (;;) {
705         Aia_Const_Prop_Ameta ameta;
706         ameta.any_update = false;
707         ameta.prev_cmp_result = AIA_CMP_NONE;
708         aia->meta_data = &ameta;
709
710         aia_const_prop_init(aia);
711         for (;;) {
712             Aia_Section *sec;
713             AIA_FOR_EACH_SECTION(aia, sec)
714                 aia_const_prop_section(sec, aia);
715
716             if (ameta.any_update) {
717                 any_update = true;
718                 ameta.any_update = false;
719             } else {
720                 break;
721             }
722         }
723
724         if (any_update) {
725             aia_const_prop_destroy(aia);
726             aia_block_elim(aia);
727             ret = true;
728             any_update = false;
729         } else {
730             if (cmdopts.dump_const_prop_ic)
731                 aia_const_prop_dump(aia, it_count);
732             aia_const_prop_destroy(aia);
733             break;
734         }
735     }
736 }
737
738 vector_clear(&globals);
739 aia_destroy_func_kills(aia);
740
741 out:
742     return ret;
743 }

```

:

A.8.4 src/aia/aia_const_prop.h

```

1 #ifndef AIA_CONST_PROP_H
2 #define AIA_CONST_PROP_H

```

```

3
4 #include <std_include.h>
5 #include "aia.h"
6 #include "aia_operand_map.h"
7 #include "aia_operand_set.h"
8
9 typedef struct Aia_Const_Prop_Bmeta {
10     Aia_Operand_Map const_ops;
11     Vector removed_instrs;
12 } Aia_Const_Prop_Bmeta;
13
14 typedef struct Aia_Const_Prop_Ameta {
15     Aia_Cmp_Result prev_cmp_result;
16     bool any_update;
17 } Aia_Const_Prop_Ameta;
18
19 #define AIA_CONST_PROP_AMETA_INIT() \
20 (Aia_Const_Prop_Ameta){ \
21     .any_update = false, \
22     .prev_cmp_result = AIA_CMP_NONE \
23 }
24
25 #define AIA_CONST_PROP_BMETA_INIT() \
26 (Aia_Const_Prop_Bmeta){ \
27     .const_ops = AIA_OPERAND_MAP_INIT(), \
28     .removed_instrs = VECTOR_INIT() \
29 }
30
31 static inline Aia_Const_Prop_Bmeta *aia_const_prop_bmeta_alloc()
32 {
33     Aia_Const_Prop_Bmeta *bmeta =
34         ALLOC_NEW(Aia_Const_Prop_Bmeta);
35     *bmeta = AIA_CONST_PROP_BMETA_INIT();
36     return bmeta;
37 }
38
39 /* Constant propagate instruction.
40  * Make sure to call __aia_set_curr_func() first. */
41 void aia_const_prop_instr(Aia_Instr *in, Aia *aia,
42     Aia_Const_Prop_Bmeta *bmeta, Aia_Const_Prop_Ameta *ameta);
43
44 /* Constant propagation pass. */
45 bool aia_const_prop(Aia *aia, Int it_count);
46
47 #endif // AIA_CONST_PROP_H

```

:

A.8.5 src/aia/aia_def_to_use.c

```

1 #include "aia_def_to_use.h"
2 #include "aia_instr_live_sets.h"
3 #include "aia_operand_map.h"
4 #include "aia_func_kills.h"
5 #include "aia_func_access.h"
6 #include <main.h>
7
8 typedef struct Replacement_Entry {
9     Aia_Instr *new_suc;
10     Vector instrs;
11     Aia_Instr *in;
12     Hash_Map_Slot hash_slot;
13 } Replacement_Entry;
14
15 #define REPLACEMENT_ENTRY_OF(slot) \
16     HASH_MAP_ENTRY(slot, Replacement_Entry, hash_slot)
17
18 static void replacement_map_insert(Hash_Map *repl_map,
19     Aia_Instr *new_suc, Aia_Instr *in)
20 {
21     Replacement_Entry *e;

```

```

22     Uns hash = hash_map_aligned_ptr_hash(new_suc);
23     Hash_Map_Slot *slot = hash_map_get(repl_map, new_suc, hash);
24     if (slot) {
25         e = REPLACEMENT_ENTRY_OF(slot);
26         DEBUGT(def,
27             Aia_Instr *tmp;
28             VECTOR_FOR_EACH_ENTRY(&e->instrs, tmp)
29                 assert(in != tmp);
30         );
31     } else {
32         e = ALLOC_NEW(Replacement_Entry);
33         e->instrs = VECTOR_INIT_SIZE(2);
34         e->new_suc = new_suc;
35         hash_map_insert(repl_map, &e->hash_slot, hash);
36     }
37     vector_append(&e->instrs, in);
38 }
39
40 static void replacement_entry_destroy(Hash_Map_Slot *slot)
41 {
42     Replacement_Entry *e = REPLACEMENT_ENTRY_OF(slot);
43     vector_clear(&e->instrs);
44     free_mem(e);
45 }
46
47 static bool replacement_entry_compare(Aia_Instr *in,
48     Hash_Map_Slot *slot)
49 {
50     Replacement_Entry *e = REPLACEMENT_ENTRY_OF(slot);
51     return e->new_suc == in;
52 }
53
54 static UNUSED Vector *replacement_map_get(Hash_Map *repl_map, Aia_Instr *in)
55 {
56     Uns hash = hash_map_aligned_ptr_hash(in);
57     Hash_Map_Slot *s = hash_map_get(repl_map, in, hash);
58     if (!s)
59         return NULL;
60     Replacement_Entry *e = REPLACEMENT_ENTRY_OF(s);
61     return &e->instrs;
62 }
63
64 static bool aia_def_to_use_operand_is_addressed(Aia_Operand *test_op,
65     Aia_Operand *use_op)
66 {
67     if (!test_op)
68         return false;
69
70     Aia_Operand *tmp;
71     switch (aia_operand_get_type(test_op)) {
72     case AIA_OPERAND_DISPLAY_REF:
73         tmp = aia_operand_display_ref_get_display_reg(test_op);
74         return aia_operands_equal(tmp, use_op);
75
76     case AIA_OPERAND_ADDR_REF:
77         tmp = aia_operand_addr_ref_get_base(test_op);
78         if (tmp && aia_operands_equal(tmp, use_op))
79             return true;
80         tmp = aia_operand_addr_ref_get_index(test_op);
81         if (tmp && aia_operands_equal(tmp, use_op))
82             return true;
83         break;
84
85     default:
86         break;
87     }
88
89     return false;
90 }
91
92 static bool aia_def_to_use_operand_uses(Aia_Operand *test_op,
93     Aia_Operand *use_op)
94 {
95     if (!test_op)

```

```

96     return false;
97     if (aia_operand_get_type(test_op) == AIA_OPERAND_REG)
98         return aia_operands_equal(test_op, use_op);
99     return aia_def_to_use_operand_is_addressed(test_op, use_op);
100 }
101
102 static Aia_Instr *aia_def_to_use_get_suc(Aia_Instr *in, Aia *aia)
103 {
104     assert(aia_instr_is_concrete(in));
105     Aia_Operand *dest = aia_instr_get_dest_op(in);
106     assert(dest);
107     assert(aia_operand_is_reg(dest));
108
109     Aia_Instr *ret = NULL;
110
111     Const_String caller = __aia_get_curr_func_name(aia);
112
113     Aia_Operand *src;
114     Aia_Instr *suc = aia_instr_get_sucessor(in);
115     while (suc) {
116         Aia_Operand *tmp = aia_instr_get_dest_op(suc);
117         AIA_INSTR_FOR_EACH_SRC(in, src) {
118             if (tmp && aia_def_to_use_operand_uses(tmp, src))
119                 goto out;
120         }
121         AIA_INSTR_FOR_EACH_OPERAND(suc, tmp) {
122             if (tmp && aia_def_to_use_operand_uses(tmp, dest))
123                 goto out;
124         }
125
126         if (aia_instr_get_type(suc) == AIA_CALL) {
127             Const_String callee;
128             Aia_Operand *lbl_op = aia_instr_get_src_op(suc, 0);
129             if (aia_operand_get_type(lbl_op) == AIA_OPERAND_LABEL)
130                 callee = aia_operand_label_get_name(lbl_op);
131             else
132                 callee = NULL;
133             AIA_INSTR_FOR_EACH_SRC(in, src) {
134                 if (func_kills_operand(callee, caller, src, aia))
135                     goto out;
136                 if (aia_func_uses(callee, caller, src, aia))
137                     goto out;
138             }
139         }
140     }
141     suc = aia_instr_get_sucessor(suc);
142     ret = suc;
143 }
144
145 out:
146     return ret;
147 }
148
149 static Aia_Instr *aia_def_to_use_new_sucessor(Aia_Instr *in,
150     Aia_Operand_Map *map, Aia *aia)
151 {
152     switch (aia_instr_get_type(in)) {
153     case AIA_CASE_INCONCRETE:
154         return NULL;
155     case AIA_CALL:
156         return NULL;
157     default:
158         break;
159     }
160
161     Aia_Operand *dest = aia_instr_get_dest_op(in);
162     if (!dest)
163         return NULL;
164
165     if (aia_operand_get_type(dest) != AIA_OPERAND_REG)
166         return NULL;
167
168     Aia_Instr *suc = aia_def_to_use_get_suc(in, aia);
169     if (!suc)

```

```

170     return NULL;
171
172     replacement_map_insert(map, suc, in);
173
174     return suc;
175 }
176
177 static void aia_def_to_use_block(Aia_Block *b, Aia *aia)
178 {
179     Hash_Map repl_map = HASH_MAP_INIT((Hash_Map_Comparator)
180         replacement_entry_compare);
181
182     Aia_Instr *in;
183     AIA_BLOCK_FOR_EACH_INSTRUCTION_REVERSED(b, in)
184         aia_def_to_use_new_sucessor(in, &repl_map, aia);
185
186     Hash_Map_Slot *slot;
187     HASH_MAP_FOR_EACH(&repl_map, slot) {
188         Replacement_Entry *e = REPLACEMENT_ENTRY_OF(slot);
189         VECTOR_FOR_EACH_ENTRY(&e->instrs, in) {
190             aia_instr_remove(in);
191             aia_instr_insert_before(in, e->new_suc);
192             if (!in->meta_data) {
193                 aia->meta_data = INT_TO_PTR(true);
194                 in->meta_data = INT_TO_PTR(true);
195             }
196         }
197     }
198
199     hash_map_for_each_destroy(&repl_map,
200         replacement_entry_destroy);
201 }
202
203 static inline void aia_def_to_use_dump(Aia *aia UNUSED, Int it_count)
204 {
205     String fname = string_from_format(S("def-to-use-ic-%D"),
206         it_count);
207     FILE *f = file_open(fname, S("w"));
208     if (!f)
209         fatal_error(S("cannot open file " QFY("%S")
210             " for def to use dump [%m]\n"),
211             fname);
212     string_destroy(fname);
213
214     aia_dump(aia, f);
215     file_close(f);
216 }
217
218 static void aia_def_to_use_block_init(Aia_Block *b)
219 {
220     Aia_Instr *in;
221     AIA_BLOCK_FOR_EACH_INSTRUCTION(b, in)
222         in->meta_data = INT_TO_PTR(false);
223 }
224
225 void aia_def_to_use(Aia *aia, Int it_count)
226 {
227     if (!cmdopts.opt_def_to_use)
228         return;
229
230     aia_collect_func_kills(aia);
231     aia_func_access(aia);
232
233     Aia_Block *b;
234     Aia_Section *sec;
235     AIA_FOR_EACH_SECTION(aia, sec) {
236         AIA_SECTION_FOR_EACH_BLOCK(sec, b)
237             aia_def_to_use_block_init(b);
238
239         Aia_Func *func;
240         AIA_SECTION_FOR_EACH_FUNC(sec, func) {
241             AIA_FUNC_FOR_EACH_BLOCK(func, b)
242                 aia_def_to_use_block_init(b);
243         }
244     }

```



```

244     Aia_Func_Trampoline *tramp;
245     AIA_FUNC_FOR_EACH_TRAMPOLINE(func, tramp)
246         aia_def_to_use_block_init(tramp->block);
247     }
248 }
249
250 do {
251     aia->meta_data = INT_TO_PTR(false);
252     AIA_FOR_EACH_SECTION(aia, sec) {
253         __aia_set_curr_func(aia, NULL);
254         AIA_SECTION_FOR_EACH_BLOCK(sec, b)
255             aia_def_to_use_block(b, aia);
256
257         Aia_Func *func;
258         AIA_SECTION_FOR_EACH_FUNC(sec, func) {
259             __aia_set_curr_func(aia, func);
260             AIA_FUNC_FOR_EACH_BLOCK(func, b)
261                 aia_def_to_use_block(b, aia);
262
263             Aia_Func_Trampoline *tramp;
264             AIA_FUNC_FOR_EACH_TRAMPOLINE(func, tramp)
265                 aia_def_to_use_block(tramp->block, aia);
266         }
267     }
268     while (aia->meta_data);
269
270     if (cmdopts.dump_def_to_use_ic)
271         aia_def_to_use_dump(aia, it_count);
272
273     aia_func_access_destroy(aia);
274     aia_destroy_func_kills(aia);
275 }

```

:

A.8.6 src/aia/aia_def_to_use.h

```

1  #ifndef AIA_DEF_TO_USE_H
2  #define AIA_DEF_TO_USE_H
3
4  #include "aia.h"
5
6  void aia_def_to_use(Aia *aia, Int it_count);
7
8  #endif // AIA_DEF_TO_USE_H

```

:

A.8.7 src/aia/aia_func_access.c

```

1  #include "aia_func_access.h"
2  #include "main.h"
3
4  #undef DEBUG_TYPE
5  #define DEBUG_TYPE func-access
6
7  typedef struct Func_Meta_Data {
8      Hash_Map func_uses;
9      bool uses_global;
10     bool any_update;
11 } Func_Meta_Data;
12
13 typedef struct Func_Use_Operand {
14     Const_String used_op_func; /* If NULL used_op is a global var. */
15     Aia_Operand *used_op;
16     Uns __func_name_hash_code;
17     bool __func_name_hash_code_set;

```

```

18 } Func_Use_Operand;
19
20 #define FUNC_USE_OPERAND_INIT(func_name) ((Func_Use_Operand){ \
21     .used_op_func = func_name,          \
22     .used_op = NULL,                    \
23     .__func_name_hash_code = 0,         \
24     .__func_name_hash_code_set = false  \
25 })
26
27 static inline void func_use_operand_set_op(Func_Use_Operand *fuse,
28     Aia_Operand *op)
29 {
30     fuse->used_op = op;
31 }
32
33 typedef struct Func_Use_Opentry {
34     Func_Use_Operand fuse;
35     Hash_Map_Slot hash_slot;
36 } Func_Use_Opentry;
37
38 #define FUNC_USE_OPENTRY_OF(slot) \
39     HASH_MAP_ENTRY(slot, Func_Use_Opentry, hash_slot)
40
41 static bool func_use_contains(Hash_Map *map, Func_Use_Operand *kill_op)
42 {
43     if (!cmdopts.opt_func_access)
44         return true;
45
46     if (!kill_op->__func_name_hash_code_set) {
47         if (kill_op->used_op_func)
48             kill_op->__func_name_hash_code = string_hash_code(
49                 kill_op->used_op_func);
50         kill_op->__func_name_hash_code_set = true;
51     }
52     Uns hash = kill_op->__func_name_hash_code + aia_operand_hash_code(
53         kill_op->used_op);
54     return hash_map_contains(map, kill_op, hash);
55 }
56
57 static void func_use_opentry_destroy(Hash_Map_Slot *slot);
58
59 static bool func_use_insert_label(Aia *aia UNUSED, Hash_Map *map,
60     Aia_Operand *label)
61 {
62     __aia_operand_acquire(label);
63
64     Func_Use_Opentry *e = ALLOC_NEW(Func_Use_Opentry);
65     e->fuse = FUNC_USE_OPERAND_INIT(NULL);
66     e->fuse.used_op_func = NULL;
67     e->fuse.used_op = label;
68     e->fuse.__func_name_hash_code_set = true;
69
70     if (func_use_contains(map, &e->fuse)) {
71         func_use_opentry_destroy(&e->hash_slot);
72         return false;
73     }
74
75     Uns hash = aia_operand_hash_code(label);
76     hash_map_insert(map, &e->hash_slot, hash);
77     return true;
78 }
79
80 static bool func_use_insert_display(Aia *aia, Hash_Map *map,
81     Aia_Operand *display_op)
82 {
83     Const_String func_name = aia_operand_display_ref_get_func_name(display_op);
84     Const_String op_name = aia_operand_display_ref_get_var_name(display_op);
85     // uint8_t var_size = aia_operand_display_ref_get_var_size(display_op);
86     Aia_Operand *kill = aia_operand_local_ref_alloc(aia, op_name
87         /*, var_size */);
88     __aia_operand_acquire(kill);
89
90     Func_Use_Opentry *e = ALLOC_NEW(Func_Use_Opentry);
91     e->fuse = FUNC_USE_OPERAND_INIT(func_name);

```

```

92     e->fuse.used_op = kill;
93     e->fuse.__func_name_hash_code = string_hash_code(func_name);
94     e->fuse.__func_name_hash_code_set = true;
95
96     if (func_use_contains(map, &e->fuse)) {
97         func_use_opentry_destroy(&e->hash_slot);
98         return false;
99     }
100
101     Uns hash = e->fuse.__func_name_hash_code + aia_operand_hash_code(kill);
102     hash_map_insert(map, &e->hash_slot, hash);
103     return true;
104 }
105
106 static bool func_use_insert_use_operand(Hash_Map *map,
107     Func_Use_Operand *kill_op)
108 {
109     Const_String func_name = kill_op->used_op_func;
110     __aia_operand_acquire(kill_op->used_op);
111
112     Func_Use_Opentry *e = ALLOC_NEW(Func_Use_Opentry);
113     e->fuse = FUNC_USE_OPERAND_INIT(func_name);
114     e->fuse.used_op = kill_op->used_op;
115     if (func_name)
116         e->fuse.__func_name_hash_code = string_hash_code(func_name);
117     e->fuse.__func_name_hash_code_set = true;
118
119     if (func_use_contains(map, &e->fuse)) {
120         func_use_opentry_destroy(&e->hash_slot);
121         return false;
122     }
123
124     Uns hash = e->fuse.__func_name_hash_code +
125         aia_operand_hash_code(e->fuse.used_op);
126     hash_map_insert(map, &e->hash_slot, hash);
127     return true;
128 }
129
130 static bool func_use_opentry_compare(Func_Use_Operand *search_op,
131     Hash_Map_Slot *slot)
132 {
133     Func_Use_Opentry *e = FUNC_USE_OPENTRY_OF(slot);
134     if (search_op->used_op_func && e->fuse.used_op_func) {
135         if (string_compare(search_op->used_op_func, e->fuse.used_op_func))
136             return false;
137     } else if (search_op->used_op_func || e->fuse.used_op_func) {
138         return false;
139     }
140     return aia_operands_equal(search_op->used_op, e->fuse.used_op);
141 }
142
143 static void func_use_opentry_destroy(Hash_Map_Slot *slot)
144 {
145     Func_Use_Opentry *e = FUNC_USE_OPENTRY_OF(slot);
146     __aia_operand_release(e->fuse.used_op);
147     free_mem(e);
148 }
149
150 static void func_use_collect_uses(Aia_Operand *op,
151     Aia *aia, Func_Meta_Data *fmeta)
152 {
153     Hash_Map *func_use_map = &fmeta->func_uses;
154     if (!op)
155         return;
156
157     switch (aia_operand_get_type(op)) {
158     case AIA_OPERAND_LABEL:
159         if (func_use_insert_label(aia, func_use_map, op))
160             fmeta->any_update = INT_TO_PTR(true);
161         break;
162
163     case AIA_OPERAND_DISPLAY_REF:
164         if (func_use_insert_display(aia, func_use_map, op))
165             fmeta->any_update = INT_TO_PTR(true);

```

```

166         break;
167
168     default:
169         break;
170     }
171 }
172
173 static void func_use_collect_func_uses(Aia_Instr *in, Aia *aia)
174 {
175     if (aia_instr_get_type(in) != AIA_CALL)
176         return;
177
178     bool uses_global = false;
179
180     Aia_Func *curr_func = __aia_get_curr_func(aia);
181
182     Func_Meta_Data *fmeta = curr_func->func_access_struct;
183
184     Aia_Operand *lbl_op = aia_instr_get_src_op(in, 0);
185     if (aia_operand_get_type(lbl_op) != AIA_OPERAND_LABEL) {
186         uses_global = true;
187         goto out;
188     }
189
190     Const_String callee_name = aia_operand_label_get_name(lbl_op);
191     Aia_Func *callee = aia_func_lookup(aia, callee_name);
192     if (!callee) {
193         uses_global = true;
194         goto out;
195     }
196
197     if (hash_map_contains(&aia->default_funcs, (String)callee_name,
198         string_hash_code(callee_name))) {
199         uses_global = false;
200         goto out;
201     }
202
203     Func_Meta_Data *callee_meta = callee->func_access_struct;
204
205     if (callee_meta->uses_global)
206         uses_global = true;
207
208     Hash_Map *curr_map = &fmeta->func_uses;
209
210     Hash_Map_Slot *slot;
211     HASH_MAP_FOR_EACH(&callee_meta->func_uses, slot) {
212         Func_Use_Opentry *e = FUNC_USE_OPENTRY_OF(slot);
213         if (!func_use_contains(curr_map, &e->fuse)) {
214             DLOG("any update 1\n");
215             fmeta->any_update = true;
216             func_use_insert_use_operand(curr_map, &e->fuse);
217         }
218     }
219
220 out:
221     if (uses_global && !fmeta->uses_global) {
222         DLOG("any update 2\n");
223         fmeta->any_update = true;
224         fmeta->uses_global = true;
225     }
226 }
227
228 static void func_use_collect_block_uses(Aia_Block *b, Aia *aia,
229     Func_Meta_Data *fmeta)
230 {
231     Aia_Instr *in;
232     AIA_BLOCK_FOR_EACH_INSTRUCTION(b, in) {
233
234         switch (aia_instr_get_type(in)) {
235         case AIA_CALL:
236             func_use_collect_func_uses(in, aia);
237             break;
238
239         case __AIA_LABEL:

```

```

240         break;
241     case __AIA_STRING:
242         break;
243     case __AIA_INTEGER:
244         break;
245
246     default:;
247         Aia_Operand *op;
248         AIA_INSTR_FOR_EACH_SRC(in, op)
249             func_use_collect_uses(op, aia, fmeta);
250         break;
251     }
252 }
253 }
254
255 static void aia_collect_func_uses(Aia *aia)
256 {
257     Func_Meta_Data *fmeta;
258     Aia_Func *func;
259     AIA_FOR_EACH_FUNC(aia, func) {
260         fmeta = ALLOC_NEW(Func_Meta_Data);
261         fmeta->func_uses = HASH_MAP_INIT_SIZE(HASH_MAP_SIZE_11,
262             (Hash_Map_Comparator) func_use_opentry_compare);
263         fmeta->uses_global = false;
264         fmeta->any_update = false;
265         func->func_access_struct = fmeta;
266     }
267
268     bool any_update;
269     do {
270         any_update = false;
271
272         AIA_FOR_EACH_FUNC(aia, func) {
273             __aia_set_curr_func(aia, func);
274             fmeta = func->func_access_struct;
275
276             fmeta->any_update = false;
277
278             Double_List_Node *bnode;
279             DOUBLE_LIST_FOR_EACH(&func->blist, bnode) {
280                 DLOG("collect kill fror %S\n", func->func_name);
281                 func_use_collect_block_uses(AIA_BLOCK_OF_DBNODE(bnode), aia,
282                     fmeta);
283             }
284
285             any_update |= fmeta->any_update;
286         }
287     } while (any_update);
288 }
289
290 static void aia_destroy_func_uses(Aia *aia)
291 {
292     if (!cmdopts.opt_func_access)
293         return;
294
295     Aia_Func *func;
296     AIA_FOR_EACH_FUNC(aia, func) {
297         Func_Meta_Data *fmeta = func->func_access_struct;
298         hash_map_for_each_destroy(&fmeta->func_uses,
299             func_use_opentry_destroy);
300         free_mem(fmeta);
301     }
302 }
303
304 void aia_func_access(Aia *aia)
305 {
306     if (!cmdopts.opt_func_access)
307         return;
308
309     aia_collect_func_uses(aia);
310 }
311
312 void aia_func_access_destroy(Aia *aia)

```

```

314 {
315     aia_destroy_func_uses(aia);
316 }
317
318 bool aia_func_uses(Const_String callee_func_name,
319                  Const_String operands_func_name, Aia_Operand *operand, Aia *aia)
320 {
321     Aia_Func *callee_func;
322     if (callee_func_name) {
323         if (hash_map_contains(&aia->default_funcs, (String)callee_func_name,
324                             string_hash_code(callee_func_name)))
325             return false;
326
327         callee_func = aia_func_lookup(aia, callee_func_name);
328     } else {
329         callee_func = NULL;
330     }
331     if (!callee_func ||
332         (!cmdopts.opt_func_access && !aia_func_is_nested(callee_func))) {
333         if (aia_operand_get_type(operand) == AIA_OPERAND_LABEL)
334             return true;
335         return false;
336     }
337
338     Func_Meta_Data *fmeta = callee_func->func_access_struct;
339     if (aia_operand_get_type(operand) == AIA_OPERAND_LABEL &&
340         (!cmdopts.opt_func_access || fmeta->uses_global))
341         return true;
342
343     Func_Use_Operand kill_op = FUNC_USE_OPERAND_INIT(operands_func_name);
344     kill_op.used_op = operand;
345
346     return func_use_contains(&fmeta->func_uses, &kill_op);
347 }

```

:

A.8.8 src/aia/aia_func_access.h

```

1  #ifndef AIA_FUNC_ACCESS_H
2  #define AIA_FUNC_ACCESS_H
3
4  #include "aia.h"
5
6  /* Note that aia_func_access() does not use func->meta_data any more.
7   * It uses func->func_access_struct now, so leave that field untouched. */
8
9  /* Assemble information about which variables, functions use and save it
10   * in func->func_access_struct. After aia_func_access() is called see
11   * aia_func_uses() to test whether a function uses a specific variable. */
12 void aia_func_access(Aia *aia);
13
14 void aia_func_access_destroy(Aia *aia);
15
16 /* Returns true if function 'callee_func_name' accesses the operand
17   * 'operand' inside function 'operands_func_name'.
18   * If 'operands_func_name' == NULL then it is assumed that 'operand'
19   * is a global static variable. And the function still returns true
20   * when 'callee_func' accesses 'operand'.
21   * If 'callee_func_name' == NULL then it is assumed callee_func_name
22   * is a global unknown function. */
23 bool aia_func_uses(Const_String callee_func_name,
24                  Const_String operands_func_name, Aia_Operand *operand, Aia *aia);
25
26 #endif // AIA_FUNC_ACCESS_H

```

:

A.8.9 src/aia/aia_func_kills.c

```

1  #include "aia_func_kills.h"
2  #include <main.h>
3
4  #undef DEBUG_TYPE
5  #define DEBUG_TYPE func-kills
6
7  typedef struct Func_Kill_Opentry {
8      Func_Kill_Operand fkill;
9      Hash_Map_Slot hash_slot;
10 } Func_Kill_Opentry;
11
12 #define FUNC_KILL_OPENTRY_OF(slot) \
13     HASH_MAP_ENTRY(slot, Func_Kill_Opentry, hash_slot)
14
15 bool __func_kill_contains(Hash_Map *map, Func_Kill_Operand *kill_op)
16 {
17     if (!cmdopts.opt_func_access)
18         return true;
19
20     if (!kill_op->__func_name_hash_code_set) {
21         DLOG("func ptr: %p\n", kill_op->killed_op_func);
22         if (kill_op->killed_op_func)
23             kill_op->__func_hame_hash_code = string_hash_code(
24                 kill_op->killed_op_func);
25         kill_op->__func_name_hash_code_set = true;
26     }
27     Uns hash = kill_op->__func_hame_hash_code + aia_operand_hash_code(
28         kill_op->killed_op);
29     return hash_map_contains(map, kill_op, hash);
30 }
31
32 static void func_kill_opentry_destroy(Hash_Map_Slot *slot);
33
34 static bool func_kill_insert_label(Aia *aia UNUSED, Hash_Map *map,
35     Aia_Operand *label)
36 {
37     __aia_operand_acquire(label);
38
39     Func_Kill_Opentry *e = ALLOC_NEW(Func_Kill_Opentry);
40     e->fkill = FUNC_KILL_OPERAND_INIT(NULL);
41     e->fkill.killed_op_func = NULL;
42     e->fkill.killed_op = label;
43     e->fkill.__func_name_hash_code_set = true;
44
45     if (__func_kill_contains(map, &e->fkill)) {
46         func_kill_opentry_destroy(&e->hash_slot);
47         return false;
48     }
49
50     Uns hash = aia_operand_hash_code(label);
51     hash_map_insert(map, &e->hash_slot, hash);
52     return true;
53 }
54
55 static bool func_kill_insert_display(Aia *aia, Hash_Map *map,
56     Aia_Operand *display_op)
57 {
58     Const_String func_name = aia_operand_display_ref_get_func_name(display_op);
59     Const_String op_name = aia_operand_display_ref_get_var_name(display_op);
60     // uint8_t var_size = aia_operand_display_ref_get_var_size(display_op);
61     Aia_Operand *kill = aia_operand_local_ref_alloc(aia, op_name
62         /*, var_size */);
63     __aia_operand_acquire(kill);
64
65     Func_Kill_Opentry *e = ALLOC_NEW(Func_Kill_Opentry);
66     e->fkill = FUNC_KILL_OPERAND_INIT(func_name);
67     e->fkill.killed_op = kill;
68     e->fkill.__func_hame_hash_code = string_hash_code(func_name);
69     e->fkill.__func_name_hash_code_set = true;
70
71     if (__func_kill_contains(map, &e->fkill)) {

```

```

72     func_kill_opentry_destroy(&e->hash_slot);
73     return false;
74 }
75
76 Uns hash = e->fkill.__func_hame_hash_code + aia_operand_hash_code(kill);
77 hash_map_insert(map, &e->hash_slot, hash);
78 return true;
79 }
80
81 static void func_kill_insert_func_kill_operand(Hash_Map *map,
82     Func_Kill_Operand *kill_op)
83 {
84     Const_String func_name = kill_op->killed_op_func;
85     __aia_operand_acquire(kill_op->killed_op);
86
87     Func_Kill_Opentry *e = ALLOC_NEW(Func_Kill_Opentry);
88     e->fkill = FUNC_KILL_OPERAND_INIT(func_name);
89     e->fkill.killed_op = kill_op->killed_op;
90     if (func_name)
91         e->fkill.__func_hame_hash_code = string_hash_code(func_name);
92     e->fkill.__func_name_hash_code_set = true;
93
94     Uns hash = e->fkill.__func_hame_hash_code +
95         aia_operand_hash_code(e->fkill.killed_op);
96     hash_map_insert(map, &e->hash_slot, hash);
97 }
98
99 static bool func_kill_opentry_compare(Func_Kill_Operand *search_op,
100     Hash_Map_Slot *slot)
101 {
102     Func_Kill_Opentry *e = FUNC_KILL_OPENTRY_OF(slot);
103     if (search_op->killed_op_func && e->fkill.killed_op_func) {
104         if (string_compare(search_op->killed_op_func, e->fkill.killed_op_func))
105             return false;
106     } else if (search_op->killed_op_func || e->fkill.killed_op_func) {
107         return false;
108     }
109     return aia_operands_equal(search_op->killed_op, e->fkill.killed_op);
110 }
111
112 static void func_kill_opentry_destroy(Hash_Map_Slot *slot)
113 {
114     Func_Kill_Opentry *e = FUNC_KILL_OPENTRY_OF(slot);
115     __aia_operand_release(e->fkill.killed_op);
116     free_mem(e);
117 }
118
119 static void aia_collect_dest_op_kill(Aia_Operand *op,
120     Aia *aia, Aia_Func_Kill_Meta *fmeta)
121 {
122     if (!op)
123         return;
124
125     Hash_Map *func_kill_map = &fmeta->func_kills;
126
127     switch (aia_operand_get_type(op)) {
128     case AIA_OPERAND_LABEL:
129         if (func_kill_insert_label(aia, func_kill_map, op))
130             fmeta->any_update = INT_TO_PTR(true);
131         break;
132
133     case AIA_OPERAND_DISPLAY_REF:
134         if (func_kill_insert_display(aia, func_kill_map, op))
135             fmeta->any_update = INT_TO_PTR(true);
136         break;
137
138     default:
139         break;
140     }
141 }
142
143 static void aia_collect_func_call_op_kills(Aia_Instr *in, Aia *aia)
144 {
145     if (aia_instr_get_type(in) != AIA_CALL)

```



```

146     return;
147
148     bool kills_global = false;
149
150     Aia_Func *curr_func = __aia_get_curr_func(aia);
151
152     Aia_Func_Kill_Meta *fmeta = curr_func->func_kills_struct;
153
154     Aia_Operand *lbl_op = aia_instr_get_src_op(in, 0);
155     if (aia_operand_get_type(lbl_op) != AIA_OPERAND_LABEL) {
156         kills_global = true;
157         goto out;
158     }
159
160     Const_String callee_name = aia_operand_label_get_name(lbl_op);
161     Aia_Func *callee = aia_func_lookup(aia, callee_name);
162     if (!callee) {
163         kills_global = true;
164         goto out;
165     }
166
167     if (hash_map_contains(&aia->default_funcs, (String)callee_name,
168         string_hash_code(callee_name))) {
169         kills_global = false;
170         goto out;
171     }
172
173     Aia_Func_Kill_Meta *callee_meta = callee->func_kills_struct;
174
175     if (callee_meta->kills_global)
176         kills_global = true;
177
178     Hash_Map *curr_map = &fmeta->func_kills;
179
180     Hash_Map_Slot *slot;
181     HASH_MAP_FOR_EACH(&callee_meta->func_kills, slot) {
182         Func_Kill_Opentry *e = FUNC_KILL_OPENTRY_OF(slot);
183         if (!__func_kill_contains(curr_map, &e->fkill)) {
184             DLOG("any update 1\n");
185             fmeta->any_update = true;
186             func_kill_insert_func_kill_operand(curr_map, &e->fkill);
187         }
188     }
189
190 out:
191     if (kills_global && !fmeta->kills_global) {
192         DLOG("any update 2\n");
193         fmeta->any_update = true;
194         fmeta->kills_global = true;
195     }
196 }
197
198 static void aia_collect_block_kills(Aia_Block *b, Aia *aia,
199     Aia_Func_Kill_Meta *fmeta)
200 {
201     Aia_Instr *in;
202     AIA_BLOCK_FOR_EACH_INSTRUCTION(b, in) {
203         switch (aia_instr_get_type(in)) {
204             case AIA_CALL:
205                 aia_collect_func_call_op_kills(in, aia);
206                 break;
207
208             case __AIA_LABEL:
209                 break;
210             case __AIA_STRING:
211                 break;
212             case __AIA_INTEGER:
213                 break;
214
215             default:;
216                 Aia_Operand *op = aia_instr_get_dest_op(in);
217                 aia_collect_dest_op_kill(op, aia, fmeta);
218         }
219     }

```

```

220 }
221
222 void aia_collect_func_kills(Aia *aia)
223 {
224     if (!cmdopts.opt_func_access)
225         return;
226
227     Aia_Func_Kill_Meta *fmeta;
228     Aia_Func *func;
229     AIA_FOR_EACH_FUNC(aia, func) {
230         fmeta = ALLOC_NEW(Aia_Func_Kill_Meta);
231         fmeta->func_kills = HASH_MAP_INIT_SIZE(HASH_MAP_SIZE_11,
232         (Hash_Map_Comparator)func_kill_opentry_compare);
233         fmeta->kills_global = false;
234         fmeta->any_update = false;
235         func->func_kills_struct = fmeta;
236     }
237
238     bool any_update;
239     do {
240         any_update = false;
241
242         AIA_FOR_EACH_FUNC(aia, func) {
243             __aia_set_curr_func(aia, func);
244             fmeta = func->func_kills_struct;
245
246             fmeta->any_update = false;
247
248             Double_List_Node *bnode;
249             DOUBLE_LIST_FOR_EACH(&func->blist, bnode)
250                 aia_collect_block_kills(AIA_BLOCK_OF_DBNODE(bnode),
251                 aia, fmeta);
252
253             any_update |= fmeta->any_update;
254         }
255     } while (any_update);
256 }
257
258 void aia_destroy_func_kills(Aia *aia)
259 {
260     if (!cmdopts.opt_func_access)
261         return;
262
263     Aia_Func *func;
264     AIA_FOR_EACH_FUNC(aia, func) {
265         Aia_Func_Kill_Meta *fmeta = func->func_kills_struct;
266         hash_map_for_each_destroy(&fmeta->func_kills,
267         func_kill_opentry_destroy);
268         free_mem(fmeta);
269     }
270 }
271
272 bool func_kills_operand(Const_String callee_func_name,
273 Const_String operands_func_name, Aia_Operand *operand, Aia *aia)
274 {
275     Aia_Func *callee_func;
276     if (callee_func_name) {
277         if (hash_map_contains(&aia->default_funcs, (String)callee_func_name,
278         string_hash_code(callee_func_name)))
279             return false;
280
281         callee_func = aia_func_lookup(aia, callee_func_name);
282     } else {
283         callee_func = NULL;
284     }
285     if (!callee_func ||
286         (!cmdopts.opt_func_access && !aia_func_is_nested(callee_func))) {
287         if (aia_operand_get_type(operand) == AIA_OPERAND_LABEL)
288             return true;
289         return false;
290     }
291     Aia_Func_Kill_Meta *fmeta = callee_func->func_kills_struct;

```

```

294     if (aia_operand_get_type(operand) == AIA_OPERAND_LABEL &&
295         (!cmdopts.opt_func_access || fmeta->kills_global))
296         return true;
297
298     Func_Kill_Operand kill_op = FUNC_KILL_OPERAND_INIT(operands_func_name);
299     kill_op.killed_op = operand;
300
301     return __func_kill_contains(&fmeta->func_kills, &kill_op);
302 }

```

:

A.8.10 src/aia/aia_func_kills.h

```

1  #ifndef AIA_FUNC_KILLS_H
2  #define AIA_FUNC_KILLS_H
3
4  #include "aia.h"
5
6  /* Note that aia_collect_func_kills() does not use func->meta_data any more.
7   * It uses func->func_kills_struct instead, so leave that field untouched. */
8
9  typedef struct Aia_Func_Kill_Meta {
10     Hash_Map func_kills;
11     bool kills_global;
12     bool any_update;
13 } Aia_Func_Kill_Meta;
14
15 typedef struct Func_Kill_Operand {
16     Const_String killed_op_func; /* If NULL killed_op is a global var. */
17     Aia_Operand *killed_op;
18     Uns __func_hame_hash_code;
19     bool __func_name_hash_code_set;
20 } Func_Kill_Operand;
21
22 #define FUNC_KILL_OPERAND_INIT(killed_func_name) ((Func_Kill_Operand){ \
23     .killed_op_func = killed_func_name,           \
24     .killed_op = NULL,                           \
25     .__func_hame_hash_code = 0,                   \
26     .__func_name_hash_code_set = false            \
27 })
28
29 static inline void func_kill_operand_set_operand(Func_Kill_Operand *fkill,
30     Aia_Operand *op)
31 {
32     fkill->killed_op = op;
33 }
34
35 bool __func_kill_contains(Hash_Map *map, Func_Kill_Operand *kill_op);
36
37 /* Returns true if 'callee_func' kills the operand 'operand' inside
38  * function 'operands_func_name'.
39  * If 'operands_func_name' == NULL then it is assumed that 'operand'
40  * is a global static variable. And the function still returns true
41  * when 'callee_func' kills 'operand'.
42  * If 'callee_func_name' == NULL then it is assumed callee_func_name
43  * is a global unknown function. */
44 bool func_kills_operand(Const_String callee_func_name,
45     Const_String operands_func_name, Aia_Operand *operand, Aia *aia);
46
47 /* Assemble information about which variables functions kill and save it
48  * in func->func_kills_struct. After aia_collect_func_kills() is called see
49  * func_kill_contains() to test wherther a function
50  * kills a specific variable. */
51 void aia_collect_func_kills(Aia *aia);
52
53 void aia_destroy_func_kills(Aia *aia);
54
55 #endif // AIA_FUNC_KILLS_H

```

:

A.8.11 src/aia/aia_instr_elim.c

```

1  #include "aia_instr_elim.h"
2  #include "aia_instr_live_sets.h"
3  #include "aia_operand_set.h"
4  #include "main.h"
5
6  #undef DEBUG_TYPE
7  #define DEBUG_TYPE instr-elim
8
9  static bool aia_instr_elim_set_global(Aia *aia, Aia_Operand *label_operand,
10 Aia_Operand *op, Aia_Operand_Set *fin_glob_set)
11 {
12     Aia_Instr *lbl_in = aia_label_to_instruction(aia, label_operand);
13     if (!lbl_in)
14         return false;
15
16     if (fin_glob_set) {
17         DEBUG(
18             DLOGT(def, " let ");
19             aia_instr_dump(stderr, lbl_in, aia);
20             DLOGT(def, "\n");
21         );
22         Aia_Operand *lbl_op = aia_instr_get_dest_op(lbl_in);
23         if (!aia_operand_set_insert(fin_glob_set, lbl_op))
24             return false;
25     }
26
27     if (aia_operand_get_type(op) != AIA_OPERAND_CONST_INT)
28         return false;
29
30     Aia_Instr *int_instr = aia_instr_get_sucessor(lbl_in);
31     assert(int_instr);
32     if (aia_instr_get_src_ops_size(int_instr) == AIA_BYTE) {
33         int32_t orig = aia_operand_const_int_get_val(op);
34         int8_t real = orig;
35         File_Location *loc = aia_instr_get_location(lbl_in);
36         if (real != orig) {
37             if (cmdopts.warn_overflow && !is_warning_reported_here(loc))
38                 report_warning_location(loc,
39                     S("constant implicitly truncated to " QFY("%" PRIi8)
40                       " to fit " QFY("char") " variable\n"), real);
41             op = aia_operand_const_int_alloc(aia, real);
42         }
43     }
44     aia_instr_replace_op(int_instr, 0, op);
45
46     return true;
47 }
48
49 static void aia_instr_elim_block(Aia_Block *b, Aia *aia, Aia_Instr *init_in,
50 Aia_Operand_Set *fin_glob_set)
51 {
52     VECTOR(removed_ins);
53
54     Aia_Instr *in;
55     AIA_BLOCK_FOR_EACH_INSTRUCTION(b, in) {
56         Aia_Operand *dest = aia_instr_get_dest_op(in);
57
58         switch (aia_instr_get_type(in)) {
59             case AIA_RET:
60                 continue;
61             case AIA_IDIV:
62                 continue;
63             case AIA_CALL:
64                 continue;
65             case AIA_CASE_INCONCRETE:
66                 continue;
67             default:
68                 break;

```

```

69     }
70
71     if (!dest)
72         continue;
73
74     switch (aia_operand_get_type(dest)) {
75     case AIA_OPERAND_ARG:
76         continue;
77
78     default:
79         break;
80     }
81
82     bool live_out = aia_instr_dest_live_out(in, in);
83     if (!live_out) {
84         if (aia_operand_get_type(dest) == AIA_OPERAND_LABEL) {
85
86             if (init_in && aia_instr_is_movx(in) &&
87                 !aia_instr_live_set_contains(init_in, dest)) {
88
89                 if (aia_instr_elim_set_global(aia, dest,
90                     aia_instr_get_src_op(in, 0), fin_glob_set))
91                     vector_append(&removed_ins, in);
92             }
93
94             } else if (!live_out) {
95                 vector_append(&removed_ins, in);
96                 switch (aia_instr_get_type(in)) {
97                 AIA_CASE_SET:
98                     /* Fall through. */
99                     AIA_CASE_COND_JUMP:;
100                     Aia_Instr *pred = aia_instr_get_predecessor(in);
101                     assert(aia_instr_get_type(pred) == AIA_CMP);
102                     vector_append(&removed_ins, pred);
103                     break;
104
105                 default:
106                     break;
107             }
108         }
109     }
110
111     #if 0
112     if (init_in && aia_operand_get_type(dest) == AIA_OPERAND_LABEL)
113         aia_instr_elim_set_global(aia, dest,
114             aia_instr_get_src_op(in, 0), fin_glob_set);
115     #endif
116     if (init_in && aia_operand_get_type(dest) == AIA_OPERAND_LABEL) {
117         if (fin_glob_set)
118             aia_operand_set_insert(fin_glob_set, dest);
119     }
120
121     }
122
123     VECTOR_FOR_EACH_ENTRY(&removed_ins, in) {
124         aia_instr_live_set_destroy(in);
125         aia_instr_remove_destroy(in);
126     }
127
128     if (vector_size(&removed_ins))
129         aia->meta_data = INT_TO_PTR(1);
130     vector_clear(&removed_ins);
131 }
132
133 static void aia_instr_elim_section(Aia_Section *sec, Aia *aia,
134     Aia_Operand_Set *fin_glob_set)
135 {
136     Aia_Instr *init_in;
137     if (sec->sec_type == AIA_SECTION_INIT)
138         init_in = __aia_block_peek_first_instr(sec->entry_block);
139     else
140         init_in = NULL;
141
142     Aia_Block *b;

```

```

143     AIA_SECTION_FOR_EACH_BLOCK(sec, b)
144         aia_instr_elim_block(b, aia, init_in, fin_glob_set);
145
146     Aia_Func *func;
147     AIA_SECTION_FOR_EACH_FUNC(sec, func) {
148         AIA_FUNC_FOR_EACH_BLOCK_REVERSED(func, b)
149             aia_instr_elim_block(b, aia, NULL, fin_glob_set);
150
151     #if 0
152         Aia_Func_Trampoline *tramp;
153         AIA_FUNC_FOR_EACH_TRAMPOLINE(func, tramp)
154             aia_instr_elim_block(tramp->block, aia, NULL, fin_glob_set);
155     #endif
156     }
157 }
158
159 static inline void aia_instr_elim_dump(Aia *aia, Int it_count)
160 {
161     String fname = string_from_format(S("instr-elim-ic-%D"),
162         it_count);
163     aia_instr_live_sets_dump(aia, fname);
164     string_destroy(fname);
165 }
166
167 static void aia_instr_elim_add_data_section(Aia_Section *dsec,
168     Aia_Instr_Forced_Liveness *f)
169 {
170     if (!dsec)
171         return;
172
173     Aia_Block *b;
174     AIA_SECTION_FOR_EACH_BLOCK(dsec, b) {
175         Aia_Instr *prev = NULL;
176         Aia_Instr *curr;
177         AIA_BLOCK_FOR_EACH_INSTRUCTION(b, curr) {
178             if (aia_instr_get_type(curr) == __AIA_INTEGER) {
179                 assert(aia_instr_get_type(prev) == __AIA_LABEL);
180                 Aia_Operand *op = aia_instr_get_src_op(curr, 0);
181                 if (aia_operand_get_type(op) == AIA_OPERAND_CONST_INT) {
182                     op = aia_instr_get_dest_op(prev);
183                     aia_instr_forced_liveness_add(f, op);
184                 }
185             }
186             prev = curr;
187         }
188     }
189 }
190
191 static inline Aia_Instr *aia_instr_elim_get_last_init_instr(Aia_Section *isec)
192 {
193     if (!isec)
194         return NULL;
195     return __aia_block_peek_last_instr(isec->exit_block);
196 }
197
198 bool aia_instr_elim(Aia *aia, Int it_count, bool final_pass)
199 {
200     size_t ret = 0;
201
202     if (!cmdopts.opt_instr_elim)
203         goto out;
204
205     VECTOR_SIZE(force, 1);
206     Aia_Instr_Forced_Liveness f = AIA_INSTR_FORCED_LIVENESS_INIT();
207
208     Aia_Operand_Set *fin_glob_set;
209     if (!final_pass) {
210         fin_glob_set = NULL;
211         aia_instr_elim_add_data_section(aia->sections[AIA_SECTION_DATA], &f);
212         vector_append(&force, &f);
213     } else {
214         fin_glob_set = aia_operand_set_alloc();
215     }
216

```

```

217     for (;;) {
218         f.instr = aia_instr_elim_get_last_init_instr(
219             aia->sections[AIA_SECTION_INIT]);
220
221         aia_instr_live_sets(aia, &force, false);
222
223         aia->meta_data = NULL;
224
225         Aia_Section *sec;
226         AIA_FOR_EACH_SECTION(aia, sec)
227             aia_instr_elim_section(sec, aia, fin_glob_set);
228
229         if (!aia->meta_data || final_pass) {
230             if (cmdopts.dump_instr_elim_ic)
231                 aia_instr_elim_dump(aia, it_count);
232             aia_instr_live_sets_destroy(aia);
233             break;
234         } else {
235             ret = true;
236         }
237
238         aia_instr_live_sets_destroy(aia);
239     }
240
241     vector_for_each_destroy(&force,
242         (Vector_Destructor)aia_instr_forced_liveness_clear);
243     if (final_pass)
244         aia_operand_set_destroy(fin_glob_set);
245
246 out:
247     return ret;
248 }

```

:

A.8.12 src/aia/aia_instr_elim.h

```

1  #ifndef AIA_INSTR_ELIM_H
2  #define AIA_INSTR_ELIM_H
3
4  #include "aia.h"
5
6  bool aia_instr_elim(Aia *aia, Int it_count, bool final_pass);
7
8  #endif // AIA_INSTR_ELIM_H

```

:

A.8.13 src/aia/aia_instr_live_sets.c

```

1  #include "aia_instr_live_sets.h"
2  #include "string_builder.h"
3  #include "aia_func_access.h"
4  #include "aia_func_kills.h"
5
6  typedef struct Aia_Meta {
7      Hash_Map *op_map;
8      bool funcs_always_kill;
9      bool any_update;
10 } Aia_Meta;
11
12 static bool aia_operand_hash_compare(Aia_Operand *search_op,
13     Hash_Map_Slot *slot)
14 {
15     Aia_Operand_Entry *e = AIA_OPERAND_ENTRY_OF(slot);
16     return aia_operands_equal(e->op, search_op);
17 }

```

```

18
19 static void aia_operand_hash_destroy(Hash_Map_Slot *slot)
20 {
21     free_mem(AIA_OPERAND_ENTRY_OF(slot));
22 }
23
24 static Live_Set_Entry *live_set_entry_alloc(Aia_Operand *op)
25 {
26     Live_Set_Entry *e = ALLOC_NEW(Live_Set_Entry);
27     e->live_operand = op;
28     __aia_operand_acquire(op);
29     return e;
30 }
31
32 static void live_set_entry_destroy(Live_Set_Entry *e)
33 {
34     __aia_operand_release(e->live_operand);
35     free_mem(e);
36 }
37
38 static inline bool live_set_remove(Live_Set *s, Aia_Operand *op)
39 {
40     Hash_Map_Slot *slot = hash_map_remove(s, op, aia_operand_hash_code(op));
41     if (slot) {
42         live_set_entry_destroy(LIVE_SET_ENTRY_OF(slot));
43         return true;
44     }
45     return false;
46 }
47
48 static inline bool live_set_contains(Live_Set *s, Aia_Operand *op)
49 {
50     return hash_map_contains(s, op, aia_operand_hash_code(op));
51 }
52
53 bool aia_instr_live_set_contains(Aia_Instr *in, Aia_Operand *op)
54 {
55     return live_set_contains(in->meta_data, op);
56 }
57
58 static inline void live_set_remove_entry(Live_Set *s, Live_Set_Entry *e)
59 {
60     Uns hash_code = hash_map_slot_get_hash_code(&e->hash_slot);
61     assert(hash_map_contains(s, e->live_operand, hash_code));
62     Hash_Map_Slot *slot = hash_map_remove(s, e->live_operand, hash_code);
63     live_set_entry_destroy(LIVE_SET_ENTRY_OF(slot));
64 }
65
66 static inline bool __live_set_insert(Live_Set *s, Aia_Operand *op,
67     Uns hash_code)
68 {
69     if (hash_map_contains(s, op, hash_code))
70         return false;
71     Live_Set_Entry *e = live_set_entry_alloc(op);
72     hash_map_insert(s, &e->hash_slot, hash_code);
73     return true;
74 }
75
76 static inline bool live_set_insert(Live_Set *s, Aia_Operand *op)
77 {
78     return __live_set_insert(s, op, aia_operand_hash_code(op));
79 }
80
81 static inline bool live_set_insert_entry(Live_Set *s, Live_Set_Entry *e)
82 {
83     return __live_set_insert(s, e->live_operand,
84         hash_map_slot_get_hash_code(&e->hash_slot));
85 }
86
87 static bool live_set_entry_compare(Aia_Operand *search_op,
88     Hash_Map_Slot *live_slot)
89 {
90     Live_Set_Entry *e = LIVE_SET_ENTRY_OF(live_slot);
91     return aia_operands_equal(search_op, e->live_operand);

```



```

92 }
93
94 static inline Live_Set *live_set_alloc()
95 {
96     Live_Set *s = ALLOC_NEW(Live_Set);
97     *s = HASH_MAP_INIT_SIZE(HASH_MAP_SIZE_11,
98                             (Hash_Map_Comparator)live_set_entry_compare);
99     return s;
100 }
101
102 static void live_set_entry_hash_destroy(Hash_Map_Slot *slot)
103 {
104     live_set_entry_destroy(LIVE_SET_ENTRY_OF(slot));
105 }
106
107 static inline void __live_set_destroy(Live_Set *s)
108 {
109     if (s) {
110         hash_map_for_each_destroy(s, live_set_entry_hash_destroy);
111         free_mem(s);
112     }
113 }
114
115 void aia_instr_live_set_destroy(Aia_Instr *in)
116 {
117     __live_set_destroy(in->meta_data);
118     in->meta_data = NULL;
119 }
120
121 static void aia_instr_add_operand_entry(Aia *aia, Aia_Operand *op)
122 {
123     if (!op)
124         return;
125
126     Aia_Meta *ameta = aia->meta_data;
127
128     switch (aia_operand_get_type(op)) {
129     case AIA_OPERAND_REG:
130         /* Fall through. */
131     case AIA_OPERAND_LABEL:
132         /* Fall through. */
133     case AIA_OPERAND_LOCAL_REF:
134         /* Fall through. */
135     case AIA_OPERAND_ARG:;
136         Uns hash = aia_operand_hash_code(op);
137         if (!hash_map_contains(ameta->op_map, op, hash)) {
138             Aia_Operand_Entry *e = ALLOC_NEW(Aia_Operand_Entry);
139             e->op = op;
140             hash_map_insert(ameta->op_map, &e->hash_slot, hash);
141         }
142         break;
143
144     case AIA_OPERAND_DISPLAY_REF:
145         aia_instr_add_operand_entry(aia,
146                                     aia_operand_display_ref_get_display_reg(op));
147         break;
148
149     case AIA_OPERAND_ADDR_REF:
150         aia_instr_add_operand_entry(aia, aia_operand_addr_ref_get_label(op));
151         aia_instr_add_operand_entry(aia, aia_operand_addr_ref_get_base(op));
152         aia_instr_add_operand_entry(aia, aia_operand_addr_ref_get_index(op));
153         break;
154
155     default:
156         break;
157     }
158 }
159
160 static void aia_instr_add_live_op(Aia_Instr *in, Aia_Operand *op, Aia *aia)
161 {
162     if (!op)
163         return;
164
165     switch (aia_operand_get_type(op)) {

```

```

166     case AIA_OPERAND_REG:
167         /* Fall through. */
168     case AIA_OPERAND_LABEL:
169         /* Fall through. */
170     case AIA_OPERAND_LOCAL_REF:
171         /* Fall through. */
172     case AIA_OPERAND_ARG:;
173         aia_instr_add_operand_entry(aia, op);
174         live_set_insert(in->meta_data, op);
175         break;
176
177     case AIA_OPERAND_DISPLAY_REF:
178         aia_instr_add_live_op(in,
179             aia_operand_display_ref_get_display_reg(op), aia);
180         break;
181
182     case AIA_OPERAND_ADDR_REF:
183         aia_instr_add_live_op(in, aia_operand_addr_ref_get_label(op), aia);
184         aia_instr_add_live_op(in, aia_operand_addr_ref_get_base(op), aia);
185         aia_instr_add_live_op(in, aia_operand_addr_ref_get_index(op), aia);
186         break;
187
188     default:
189         break;
190 }
191 }
192
193 static void aia_instr_live_init(Aia_Instr *in, Aia *aia)
194 {
195     in->meta_data = live_set_alloc();
196     Aia_Operand *op;
197     AIA_INSTR_FOR_EACH_SRC(in, op)
198         aia_instr_add_live_op(in, op, aia);
199
200     op = aia_instr_get_dest_op(in);
201     if (!op)
202         return;
203
204     aia_instr_add_operand_entry(aia, op);
205
206     switch (aia_operand_get_type(op)) {
207     case AIA_OPERAND_ADDR_REF:
208         /* Fall through. */
209     case AIA_OPERAND_DISPLAY_REF:
210         aia_instr_add_live_op(in, op, aia);
211         break;
212
213     default:
214         break;
215     }
216 }
217
218 static void aia_instr_live_func_access(Aia_Instr *in, Aia *aia)
219 {
220     if (aia_instr_get_type(in) != AIA_CALL)
221         return;
222
223     Aia_Meta *ameta = aia->meta_data;
224
225     Aia_Operand *lbl_op = aia_instr_get_src_op(in, 0);
226     if (aia_operand_get_type(lbl_op) != AIA_OPERAND_LABEL) {
227         // Add all global
228         Hash_Map *map = ameta->op_map;
229         Hash_Map_Slot *slot;
230         HASH_MAP_FOR_EACH(map, slot) {
231             Aia_Operand_Entry *e = AIA_OPERAND_ENTRY_OF(slot);
232             if (aia_operand_get_type(e->op) == AIA_OPERAND_LABEL)
233                 aia_instr_add_live_op(in, e->op, aia);
234         }
235     }
236     return;
237 }
238
239 Const_String callee_name = aia_operand_label_get_name(lbl_op);

```

```

240 #if 0
241     if (hash_map_contains(&aia->default_funcs, (String)callee_name,
242         string_hash_code(callee_name)))
243         return;
244     Aia_Func *callee = aia_func_lookup(aia, callee_name);
245 #endif
246
247     Const_String curr_func_name = __aia_get_curr_func_name(aia);
248
249     inline void add_operand(Aia_Operand *op)
250     {
251         if (ameta->funcs_always_kill && func_kills_operand(callee_name,
252             curr_func_name, op, aia))
253             return;
254
255         if (aia_func_uses(callee_name, curr_func_name, op, aia))
256             aia_instr_add_live_op(in, op, aia);
257     }
258
259     Hash_Map *map = ameta->op_map;
260     Hash_Map_Slot *slot;
261     HASH_MAP_FOR_EACH(map, slot) {
262         Aia_Operand_Entry *e = AIA_OPERAND_ENTRY_OF(slot);
263         add_operand(e->op);
264     }
265
266     if (!curr_func_name)
267         return;
268
269     Aia_Func *func = __aia_get_curr_func(aia);
270
271     Vector *locals = &func->locals;
272     Const_String loc;
273     VECTOR_FOR_EACH_ENTRY(locals, loc) {
274         Aia_Operand *op = aia_operand_local_ref_alloc(aia, loc);
275         __aia_operand_acquire(op);
276         add_operand(op);
277         __aia_operand_release(op);
278     }
279 }
280
281 static inline void aia_instr_block_live_init(Aia_Block *b, Aia *aia)
282 {
283     Aia_Instr *in;
284     AIA_BLOCK_FOR_EACH_INSTRUCTION(b, in)
285         aia_instr_live_init(in, aia);
286 }
287
288 static inline void aia_instr_block_func_access(Aia_Block *b, Aia *aia)
289 {
290     Aia_Instr *in;
291     AIA_BLOCK_FOR_EACH_INSTRUCTION(b, in)
292         aia_instr_live_func_access(in, aia);
293 }
294
295 static void aia_instr_section_live_init(Aia_Section *sec, Aia *aia)
296 {
297     __aia_set_curr_func(aia, NULL);
298
299     Aia_Meta *ameta = aia->meta_data;
300
301     Aia_Block *b;
302     AIA_SECTION_FOR_EACH_BLOCK_REVERSED(sec, b)
303         aia_instr_block_live_init(b, aia);
304     AIA_SECTION_FOR_EACH_BLOCK_REVERSED(sec, b)
305         aia_instr_block_func_access(b, aia);
306
307     hash_map_for_each_destroy(ameta->op_map, aia_operand_hash_destroy);
308
309     Aia_Func *func;
310     AIA_SECTION_FOR_EACH_FUNC(sec, func) {
311         __aia_set_curr_func(aia, func);
312
313         AIA_FUNC_FOR_EACH_BLOCK_REVERSED(func, b)

```

```

314     aia_instr_block_live_init(b, aia);
315     AIA_FUNC_FOR_EACH_BLOCK_REVERSED(func, b)
316         aia_instr_block_func_access(b, aia);
317
318     hash_map_for_each_destroy(ameta->op_map, aia_operand_hash_destroy);
319
320     Aia_Func_Trampoline *tramp;
321     AIA_FUNC_FOR_EACH_TRAMPOLINE(func, tramp) {
322         aia_instr_block_live_init(tramp->block, aia);
323         aia_instr_block_func_access(tramp->block, aia);
324         Aia_Instr *jmp_in = __aia_block_peek_last_instr(tramp->block);
325         aia_instr_add_live_op(jmp_in, aia->record_self_ptr, aia);
326
327         hash_map_for_each_destroy(ameta->op_map, aia_operand_hash_destroy);
328     }
329 }
330
331
332 static inline void live_sets_init(Aia *aia, Vector *forced_live,
333     bool funcs_always_kill)
334 {
335     HASH_MAP(operand_map, (Hash_Map_Comparator)aia_operand_hash_compare);
336     aia->meta_data = &(Aia_Meta){ &operand_map, funcs_always_kill, false };
337
338     Aia_Section *sec;
339     AIA_FOR_EACH_SECTION(aia, sec)
340         aia_instr_section_live_init(sec, aia);
341
342     if (!forced_live)
343         return;
344
345     Aia_Instr_Forced_Liveness *f;
346     VECTOR_FOR_EACH_ENTRY(f, forced_live, f) {
347         Live_Set *set = f->instr->meta_data;
348
349         Aia_Operand *op;
350         VECTOR_FOR_EACH_ENTRY(op, f->live_operands, op)
351             live_set_insert(set, op);
352     }
353 }
354
355 static inline bool aia_instr_call_defines_op(Aia_Instr *in,
356     Aia_Operand *op, Aia *aia)
357 {
358     if (aia_instr_get_type(in) != AIA_CALL)
359         return false;
360
361     Aia_Meta *ameta = aia->meta_data;
362     if (!ameta->funcs_always_kill)
363         return false;
364
365     Aia_Operand *lbl_op = aia_instr_get_src_op(in, 0);
366     if (aia_operand_get_type(lbl_op) != AIA_OPERAND_LABEL)
367         return false;
368
369     Const_String callee_name = aia_operand_label_get_name(lbl_op);
370     Const_String curr_func_name = __aia_get_curr_func_name(aia);
371
372     return func_kills_operand(callee_name, curr_func_name, op, aia);
373 }
374
375 static bool aia_instr_defines_op(Aia_Instr *in, Aia_Operand *op, Aia *aia)
376 {
377     Aia_Operand *dest = aia_instr_get_dest_op(in);
378     if (!dest)
379         return false;
380
381     if (aia_instr_call_defines_op(in, op, aia))
382         return true;
383
384     switch (aia_operand_get_type(op)) {
385     case AIA_OPERAND_REG:
386         /* Fall through. */
387     case AIA_OPERAND_LABEL:

```

```

388     /* Fall through. */
389     case AIA_OPERAND_LOCAL_REF:
390     /* Fall through. */
391     case AIA_OPERAND_ARG:
392     /* Fall through. */
393     return aia_operands_equal(dest, op);
394
395     case AIA_OPERAND_DISPLAY_REF:
396     /* Fall through. */
397     case AIA_OPERAND_ADDR_REF:
398     return false;
399
400     default:
401     return false;
402 }
403 }
404
405 static void aia_instr_live_add(Aia_Instr *in, Live_Set *vars, Aia *aia)
406 {
407     Aia_Meta *ameta = aia->meta_data;
408     Live_Set_Entry *e;
409     LIVE_SET_FOR_EACH_ENTRY(vars, e) {
410         if (!aia_instr_defines_op(in, e->live_operand, aia)) {
411             if (live_set_insert_entry(in->meta_data, e))
412                 ameta->any_update = true;
413         }
414     }
415 }
416
417 static void aia_instr_live_set_build(Aia_Instr *in, Aia_Block *b, Aia *aia)
418 {
419     Aia_Instr *pred = aia_instr_get_predecessor(in);
420     if (!pred) {
421         Aia_Block *pred_block;
422         AIA_BLOCK_FOR_EACH_PREDECESSOR(b, pred_block) {
423             aia_instr_live_add(__aia_block_peek_last_instr(
424                 pred_block), in->meta_data, aia);
425         }
426     } else {
427         aia_instr_live_add(pred, in->meta_data, aia);
428     }
429 }
430
431
432 static void aia_instr_block_live_sets_build(Aia_Block *b, Aia *aia)
433 {
434     Aia_Instr *in;
435     AIA_BLOCK_FOR_EACH_INSTRUCTION_REVERSED(b, in)
436         aia_instr_live_set_build(in, b, aia);
437 }
438
439
440 static void aia_section_live_sets_build(Aia *aia, Aia_Section *sec)
441 {
442     __aia_set_curr_func(aia, NULL);
443     Aia_Block *b;
444     AIA_SECTION_FOR_EACH_BLOCK_REVERSED(sec, b)
445         aia_instr_block_live_sets_build(b, aia);
446
447     Aia_Func *func;
448     AIA_SECTION_FOR_EACH_FUNC(sec, func) {
449         __aia_set_curr_func(aia, func);
450         AIA_FUNC_FOR_EACH_BLOCK_REVERSED(func, b)
451             aia_instr_block_live_sets_build(b, aia);
452
453         Aia_Func_trampoline *tramp;
454         AIA_FUNC_FOR_EACH_TRAMPOLINE(func, tramp)
455             aia_instr_block_live_sets_build(tramp->block, aia);
456     }
457 }
458
459 static inline void live_sets_build(Aia *aia, bool funcs_always_kill)
460 {
461     Aia_Section *sec;

```

```

462     Aia_Meta ameta = { NULL, funcs_always_kill, false };
463     aia->meta_data = &ameta;
464     do {
465         ameta.any_update = false;
466         AIA_FOR_EACH_SECTION(aia, sec)
467             aia_section_live_sets_build(aia, sec);
468     } while (ameta.any_update);
469 }
470
471 static void aia_instr_live_destroy(Aia_Block *b)
472 {
473     Aia_Instr *in;
474     AIA_BLOCK_FOR_EACH_INSTRUCTION(b, in)
475         aia_instr_live_set_destroy(in);
476 }
477
478 static void aia_instr_section_live_destroy(Aia_Section *sec)
479 {
480     Aia_Block *b;
481     AIA_SECTION_FOR_EACH_BLOCK_REVERSED(sec, b)
482         aia_instr_live_destroy(b);
483
484     Aia_Func *func;
485     AIA_SECTION_FOR_EACH_FUNC(sec, func) {
486         AIA_FUNC_FOR_EACH_BLOCK_REVERSED(func, b)
487             aia_instr_live_destroy(b);
488
489         Aia_Func_Trampoline *tramp;
490         AIA_FUNC_FOR_EACH_TRAMPOLINE(func, tramp)
491             aia_instr_live_destroy(tramp->block);
492     }
493 }
494
495 void aia_instr_live_sets_destroy(Aia *aia)
496 {
497     Aia_Section *sec;
498     AIA_FOR_EACH_SECTION(aia, sec)
499         aia_instr_section_live_destroy(sec);
500 }
501
502 void aia_instr_live_sets(Aia *aia, Vector *forced_live,
503                          bool funcs_always_kill)
504 {
505     if (funcs_always_kill)
506         aia_collect_func_kills(aia);
507
508     aia_func_access(aia);
509     live_sets_init(aia, forced_live, funcs_always_kill);
510     aia_func_access_destroy(aia);
511
512     live_sets_build(aia, funcs_always_kill);
513
514     if (funcs_always_kill)
515         aia_destroy_func_kills(aia);
516 }
517
518 static bool aia_instr_is_live_set_op(Aia_Operand *op)
519 {
520     if (!op)
521         return false;
522
523     switch (aia_operand_get_type(op)) {
524     case AIA_OPERAND_REG:
525         /* Fall through. */
526     case AIA_OPERAND_LABEL:
527         /* Fall through. */
528     case AIA_OPERAND_LOCAL_REF:
529         /* Fall through. */
530     case AIA_OPERAND_ARG:
531         /* Fall through. */
532         return true;
533
534     case AIA_OPERAND_DISPLAY_REF:
535         /* Fall through. */

```

```

536     case AIA_OPERAND_ADDR_REF:
537         /* Fall through. */
538     default:
539         return false;
540     }
541 }
542
543 static bool aia_instr_is_live_set_instr(Aia_Instr *in)
544 {
545     return aia_instr_is_concrete(in);
546 }
547
548 /* Returns true if destination operand of dest_instr is
549  * live going out of in. */
550 bool aia_instr_dest_live_out(Aia_Instr *in, Aia_Instr *dest_instr)
551 {
552     bool ret = true;
553
554     Aia_Operand *op = aia_instr_get_dest_op(dest_instr);
555
556     if (!op)
557         goto out;
558     if (!aia_instr_is_live_set_op(op))
559         goto out;
560     if (!aia_instr_is_live_set_instr(dest_instr))
561         goto out;
562
563     Aia_Instr *suc = aia_instr_get_sucessor(in);
564
565     if (!suc) {
566         Aia_Block *b = aia_instr_get_block(in);
567         Aia_Block *suc_block;
568         AIA_BLOCK_FOR_EACH_SUCESSOR(b, suc_block) {
569             suc = __aia_block_peek_first_instr(suc_block);
570             if (live_set_contains(suc->meta_data, op)) {
571                 ret = true;
572                 goto out;
573             }
574         }
575         ret = false;
576     } else {
577         ret = live_set_contains(suc->meta_data, op);
578     }
579
580 out:
581     return ret;
582 }
583
584 static void aia_instr_live_set_dump(FILE *stream, Aia_Instr *in)
585 {
586     file_print_message(stream, S(" # live { "));
587     Live_Set_Entry *e;
588     Live_Set *s = in->meta_data;
589
590     Uns live_size = hash_map_size(s);
591     if (!live_size)
592         goto out;
593
594     LIVE_SET_FOR_EACH_ENTRY(s, e) {
595         aia_operand_dump(stream, e->live_operand, false);
596         if (--live_size)
597             file_print_message(stream, S(", "));
598     }
599
600 out:
601     file_print_message(stream, S(" }"));
602 }
603
604 void aia_instr_live_sets_dump(Aia *aia, Const_String postfix)
605 {
606     aia_set_instr_dump_callback(aia, aia_instr_live_set_dump);
607
608     String fname = string_from_format(S("%S.vitaly.%S"),
609         aia_get_file_name(aia), postfix);

```

```

610     FILE *f = file_open(fname, S("w"));
611     if (!f)
612         fatal_error(S("unable to open file %S for intermediate "
613             "code dump [%m]\n"), fname);
614
615     string_destroy(fname);
616     aia_dump(aia, f);
617     file_close(f);
618
619     aia_clear_instr_dump_callback(aia);
620 }

```

:

A.8.14 src/aia/aia_instr_live_sets.h

```

1  #ifndef AIA_INSTR_LIVE_SETS_H
2  #define AIA_INSTR_LIVE_SETS_H
3
4  #include "aia.h"
5  #include <hash_map.h>
6
7  /* Uses instr->meta_data and aia->meta_data,
8   * so make sure they are free before calling aia_instr_live_sets().
9   * It is a requirement to let the instr->meta_data alone until
10  * aia_instr_live_sets_destroy() is called. */
11
12  typedef Hash_Map Live_Set;
13
14  typedef struct Live_Set_Entry {
15     Aia_Operand *live_operand;
16     Hash_Map_Slot hash_slot;
17 } Live_Set_Entry;
18
19 #define LIVE_SET_ENTRY_OF(slot) \
20     HASH_MAP_ENTRY(slot, Live_Set_Entry, hash_slot)
21
22 // Warning. Goto to break the loop.
23 #define LIVE_SET_FOR_EACH_ENTRY(live_set, ent) \
24     for (Hash_Map_Slot *__s = INT_TO_PTR(1); __s; __s = NULL) \
25         HASH_MAP_FOR_EACH((Live_Set *)live_set, __s) \
26             if ((ent = LIVE_SET_ENTRY_OF(__s)) || !ent)
27
28 #define AIA_INSTR_LIVE_SET_FOR_EACH_OP(instr, operand) \
29     for (Live_Set_Entry *__e = NULL; !__e; __e = INT_TO_PTR(1)) \
30         LIVE_SET_FOR_EACH_ENTRY((instr->meta_data, __e) \
31             if ((operand = __e->live_operand) || !operand)
32
33  typedef struct Aia_Operand_Entry {
34     Aia_Operand *op;
35     Hash_Map_Slot hash_slot;
36 } Aia_Operand_Entry;
37
38 #define AIA_OPERAND_ENTRY_OF(slot) \
39     HASH_MAP_ENTRY(slot, Aia_Operand_Entry, hash_slot)
40
41 /* Struct to force operands in live_operands live into instruction instr. */
42  typedef struct Aia_Instr_Forced_Liveness {
43     Aia_Instr *instr;
44     Vector live_operands;
45 } Aia_Instr_Forced_Liveness;
46
47 #define AIA_INSTR_FORCED_LIVENESS_INIT() ((Aia_Instr_Forced_Liveness){ \
48     .instr = NULL, \
49     .live_operands = VECTOR_INIT() \
50 })
51
52  static inline void aia_instr_forced_liveness_add(Aia_Instr_Forced_Liveness *f,
53     Aia_Operand *op)
54 {
55     __aia_operand_acquire(op);

```



```

56     vector_append(&f->live_operands, op);
57 }
58
59 static inline void aia_instr_forced_liveness_clear(
60     Aia_Instr_Forced_Liveness *f)
61 {
62     Aia_Operand *op;
63     VECTOR_FOR_EACH_ENTRY(&f->live_operands, op)
64         __aia_operand_release(op);
65     vector_clear(&f->live_operands);
66 }
67
68 /* Saves Live sets in instr->meta_data.
69  * forced_liveness is a vector with Aia_Instr_Forced_Liveness structs
70  * to force liveness into instructions.
71  * forced_liveness == NULL is fine.
72  *
73  * If funcs_always_kill == true then the liveness analysis will
74  * assume that if a function kill some operand in some way
75  * the operand is never live in or out of the function. */
76 void aia_instr_live_sets(Aia *aia, Vector *forced_liveness,
77     bool funcs_always_kill);
78
79 /* Destroy the live sets in instr->meta_data. */
80 void aia_instr_live_sets_destroy(Aia *aia);
81
82 /* destroy live set in in->meta_data.
83  * Call this when an instruction is removed before
84  * calling aia_instr_live_sets_destroy(). */
85 void aia_instr_live_set_destroy(Aia_Instr *in);
86
87 /* Returns true if destination operand of dest_instr is
88  * live going out of in.
89  * Also returns true if dest == NULL. */
90 bool aia_instr_dest_live_out(Aia_Instr *in, Aia_Instr *dest_instr);
91
92 /* Dump IC with live sets and file name: "file.vit.vitaly.postfix". */
93 void aia_instr_live_sets_dump(Aia *aia, Const_String postfix);
94
95 /* Return true if instr has op in live set. */
96 bool aia_instr_live_set_contains(Aia_Instr *in, Aia_Operand *op);
97
98 #endif // AIA_INSTR_LIVE_SETS_H

```

:

A.8.15 src/aia/aia_optimize.c

```

1  #include "aia_optimize.h"
2  #include "aia_const_prop.h"
3  #include "aia_instr_elim.h"
4  #include "aia_block_elim.h"
5  #include "aia_unused_set.h"
6  #include "aia_def_to_use.h"
7  #include "aia_warn_undefined.h"
8  #include <main.h>
9
10 #undef DEBUG_TYPE
11 #define DEBUG_TYPE aia-optimize
12
13 void aia_optimize(Aia *aia)
14 {
15     if (!aia_is_valid(aia))
16         return;
17
18     aia_unused_set_eliminate(aia);
19
20     bool any_update;
21     Int it_count = 0;
22
23     inline void run()

```

```

24     {
25         do {
26             DLOG("optimize %S\n\tpass %#D\n",
27                 aia_get_file_name(aia), it_count + 1);
28             any_update = false;
29             any_update |= aia_const_prop(aia, it_count);
30             any_update |= aia_instr_elim(aia, it_count, false);
31             aia_def_to_use(aia, it_count);
32             ++it_count;
33         } while (any_update);
34     }
35
36     run();
37     aia_instr_elim(aia, it_count, true);
38     run();
39
40     aia_warn_undefined(aia);
41 }

```

:

A.8.16 src/aia/aia_optimize.h

```

1  #ifndef AIA_OPTIMIZE_H
2  #define AIA_OPTIMIZE_H
3
4  #include "aia.h"
5
6  void aia_optimize(Aia *aia);
7
8  #endif // AIA_OPTIMIZE_H

```

:

A.8.17 src/aia/aia_unused_set.c

```

1  #include "aia_unused_set.h"
2  #include "aia_instr_live_sets.h"
3  #include <main.h>
4
5  #ifdef SEQLEN
6  #undef SEQLEN
7  #endif
8
9  #ifdef SEQLAST
10 #undef SEQLAST
11 #endif
12
13 /* length of the instruction sequence: .cmp; .setxy; .cmp; .jxy; */
14 #define SEQLEN 4
15
16 #define SEQLAST (SEQLEN - 1)
17
18 static bool aia_unused_set_sequence(Aia_Instr *seq[SEQLEN],
19     Vector *removed_instr_dest)
20 {
21     if (aia_instr_get_type(seq[0]) != AIA_CMP)
22         return false;
23     if (!aia_instr_is_set_instr(seq[1]))
24         return false;
25     if (aia_instr_get_type(seq[2]) != AIA_CMP)
26         return false;
27     if (!aia_instr_is_cond_jump(seq[3]))
28         return false;
29
30     if (aia_instr_dest_live_out(seq[3], seq[1]))
31         return false;

```

```

32
33 inline bool operand_is_int0(Aia_Operand *op)
34 {
35     if (aia_operand_get_type(op) != AIA_OPERAND_CONST_INT)
36         return false;
37     return !aia_operand_const_int_get_val(op);
38 }
39
40 Aia_Operand *dest = aia_instr_get_dest_op(seq[1]);
41 if (!aia_operand_is_reg(dest))
42     return false;
43
44 Aia_Operand *op1 = aia_instr_get_src_op(seq[2], 0);
45 if (!aia_operands_equal(dest, op1))
46     return false;
47
48 Aia_Operand *op2 = aia_instr_get_src_op(seq[2], 1);
49 if (!operand_is_int0(op2))
50     return false;
51
52 uint16_t new_jump = AIA_JE; // assign something to make gcc shut up...
53
54 /* Assume the destination from the .set instruction is also destination
55  * in the 2nd .cmp instruction. */
56 switch (aia_instr_get_type(seq[1])) {
57 case AIA_SETE:
58     switch (aia_instr_get_type(seq[3])) {
59     case AIA_JNE:
60         new_jump = AIA_JE;
61         break;
62     case AIA_JE:
63         new_jump = AIA_JNE;
64         break;
65     case AIA_JG:
66         new_jump = AIA_JE;
67         break;
68     case AIA_JLE:
69         new_jump = AIA_JNE;
70         break;
71     case AIA_JL:
72         new_jump = AIA_JE;
73         break;
74     case AIA_JGE:
75         new_jump = AIA_JNE;
76         break;
77     default:
78         assert(false);
79     }
80     break;
81
82 case AIA_SETNE:
83     switch (aia_instr_get_type(seq[3])) {
84     case AIA_JNE:
85         new_jump = AIA_JNE;
86         break;
87     case AIA_JE:
88         new_jump = AIA_JE;
89         break;
90     case AIA_JG:
91         new_jump = AIA_JNE;
92         break;
93     case AIA_JLE:
94         new_jump = AIA_JE;
95         break;
96     case AIA_JL:
97         new_jump = AIA_JNE;
98         break;
99     case AIA_JGE:
100         new_jump = AIA_JE;
101         break;
102     default:
103         assert(false);
104     }
105     break;

```

```

106
107     case AIA_SETL:
108         switch (aia_instr_get_type(seq[3])) {
109             case AIA_JNE:
110                 new_jump = AIA_JL;
111                 break;
112             case AIA_JE:
113                 new_jump = AIA_JGE;
114                 break;
115             case AIA_JG:
116                 new_jump = AIA_JL;
117                 break;
118             case AIA_JLE:
119                 new_jump = AIA_JGE;
120                 break;
121             case AIA_JL:
122                 new_jump = AIA_JL;
123                 break;
124             case AIA_JGE:
125                 new_jump = AIA_JGE;
126                 break;
127             default:
128                 assert(false);
129         }
130         break;
131
132     case AIA_SETGE:
133         switch (aia_instr_get_type(seq[3])) {
134             case AIA_JNE:
135                 new_jump = AIA_JGE;
136                 break;
137             case AIA_JE:
138                 new_jump = AIA_JL;
139                 break;
140             case AIA_JG:
141                 new_jump = AIA_JGE;
142                 break;
143             case AIA_JLE:
144                 new_jump = AIA_JL;
145                 break;
146             case AIA_JL:
147                 new_jump = AIA_JGE;
148                 break;
149             case AIA_JGE:
150                 new_jump = AIA_JL;
151                 break;
152             default:
153                 assert(false);
154         }
155         break;
156
157     case AIA_SETG:
158         switch (aia_instr_get_type(seq[3])) {
159             case AIA_JNE:
160                 new_jump = AIA_JG;
161                 break;
162             case AIA_JE:
163                 new_jump = AIA_JLE;
164                 break;
165             case AIA_JG:
166                 new_jump = AIA_JG;
167                 break;
168             case AIA_JLE:
169                 new_jump = AIA_JLE;
170                 break;
171             case AIA_JL:
172                 new_jump = AIA_JG;
173                 break;
174             case AIA_JGE:
175                 new_jump = AIA_JLE;
176                 break;
177             default:
178                 assert(false);
179         }

```

```

180         break;
181
182     case AIA_SETLE:
183         switch (aia_instr_get_type(seq[3])) {
184             case AIA_JNE:
185                 new_jump = AIA_JLE;
186                 break;
187             case AIA_JE:
188                 new_jump = AIA_JG;
189                 break;
190             case AIA_JG:
191                 new_jump = AIA_JLE;
192                 break;
193             case AIA_JLE:
194                 new_jump = AIA_JG;
195                 break;
196             case AIA_JL:
197                 new_jump = AIA_JLE;
198                 break;
199             case AIA_JGE:
200                 new_jump = AIA_JG;
201                 break;
202             default:
203                 assert(false);
204         }
205         break;
206
207     default:
208         assert(false);
209 }
210
211 vector_append(removed_instr_dest, seq[1]);
212 vector_append(removed_instr_dest, seq[2]);
213 seq[3]->type = new_jump;
214
215 return true;
216 }
217
218 static void aia_unused_set_block(Aia_Block *b, Aia *aia)
219 {
220     VECTOR(removed_ins);
221
222     Aia_Instr *prev_ins[SEQLEN];
223     int i = 0;
224
225     Aia_Instr *in;
226     AIA_BLOCK_FOR_EACH_INSTRUCTION(b, in) {
227         if (i < SEQLAST) {
228             prev_ins[i + 1] = in;
229         } else {
230             for (int j = 0; j < SEQLAST; j++) {
231                 prev_ins[j] = prev_ins[j + 1];
232             }
233             prev_ins[SEQLAST] = in;
234             if (aia_unused_set_sequence(prev_ins, &removed_ins))
235                 aia->meta_data = INT_TO_PTR(true);
236         }
237         ++i;
238     }
239
240     VECTOR_FOR_EACH_ENTRY(&removed_ins, in) {
241         aia_instr_live_set_destroy(in);
242         aia_instr_remove_destroy(in);
243     }
244     vector_clear(&removed_ins);
245 }
246
247 static void aia_unused_set_section(Aia_Section *sec, Aia *aia)
248 {
249     Aia_Block *b;
250     AIA_SECTION_FOR_EACH_BLOCK_REVERSED(sec, b)
251         aia_unused_set_block(b, aia);
252
253     Aia_Func *func;

```

```

254     AIA_SECTION_FOR_EACH_FUNC(sec, func) {
255         AIA_FUNC_FOR_EACH_BLOCK_REVERSED(func, b)
256             aia_unused_set_block(b, aia);
257
258         Aia_Func_Trampoline *tramp;
259         AIA_FUNC_FOR_EACH_TRAMPOLINE(func, tramp)
260             aia_unused_set_block(tramp->block, aia);
261     }
262 }
263
264 bool aia_unused_set_eliminate(Aia *aia)
265 {
266     bool ret = false;
267     if (!cmdopts.opt_unused_set)
268         goto out;
269
270     aia_instr_live_sets(aia, NULL, false);
271
272     aia->meta_data = INT_TO_PTR(false);
273
274     Aia_Section *sec;
275     AIA_FOR_EACH_SECTION(aia, sec)
276         aia_unused_set_section(sec, aia);
277
278     ret = PTR_TO_INT(aia->meta_data);
279
280     aia_instr_live_sets_destroy(aia);
281
282 out:
283     return ret;
284 }

```

:

A.8.18 src/aia/aia_unused_set.h

```

1  #ifndef AIA_UNUSED_SET_H
2  #define AIA_UNUSED_SET_H
3
4  #include "aia.h"
5
6  bool aia_unused_set_eliminate(Aia *aia);
7
8  #endif // AIA_UNUSED_SET_H

```

:

A.8.19 src/x86_32/x86_32_regs.c

```

1  #include <aia/aia.h>
2  #include "x86_32_regs.h"
3  #include <hash_map.h>
4
5  CONST_STRING(reg_eax_str, "eax");
6  CONST_STRING(reg_al_str, "al");
7  CONST_STRING(reg_ah_str, "ah");
8
9  CONST_STRING(reg_ebx_str, "ebx");
10 CONST_STRING(reg_bl_str, "bl");
11 CONST_STRING(reg_bh_str, "bh");
12
13 CONST_STRING(reg_ecx_str, "ecx");
14 CONST_STRING(reg_cl_str, "cl");
15 CONST_STRING(reg_ch_str, "ch");
16
17 CONST_STRING(reg_edx_str, "edx");
18 CONST_STRING(reg_dl_str, "dl");

```

```

19  CONST_STRING(reg_dh_str, "dh");
20
21  CONST_STRING(reg_esi_str, "esi");
22  CONST_STRING(reg_edi_str, "edi");
23  CONST_STRING(reg_ebp_str, "ebp");
24  CONST_STRING(reg_esp_str, "esp");
25
26  Aia_Operand *reg_eax;
27  Aia_Operand *reg_al;
28  Aia_Operand *reg_ah;
29
30  Aia_Operand *reg_edx;
31  Aia_Operand *reg_dl;
32  Aia_Operand *reg_dh;
33
34  Aia_Operand *reg_ecx;
35  Aia_Operand *reg_cl;
36  Aia_Operand *reg_ch;
37
38  Aia_Operand *reg_ebx;
39  Aia_Operand *reg_bl;
40  Aia_Operand *reg_bh;
41
42  Aia_Operand *reg_esi;
43  Aia_Operand *reg_edi;
44  Aia_Operand *reg_ebp;
45  Aia_Operand *reg_esp;
46
47  X86_32_Reg_List *reg_list_eax;
48  X86_32_Reg_List *reg_list_al;
49  X86_32_Reg_List *reg_list_ah;
50
51  X86_32_Reg_List *reg_list_ebx;
52  X86_32_Reg_List *reg_list_bl;
53  X86_32_Reg_List *reg_list_bh;
54
55  X86_32_Reg_List *reg_list_ecx;
56  X86_32_Reg_List *reg_list_cl;
57  X86_32_Reg_List *reg_list_ch;
58
59  X86_32_Reg_List *reg_list_dh;
60  X86_32_Reg_List *reg_list_dh;
61  X86_32_Reg_List *reg_list_edx;
62
63  X86_32_Reg_List *reg_list_esi;
64  X86_32_Reg_List *reg_list_edi;
65  X86_32_Reg_List *reg_list_ebp;
66  X86_32_Reg_List *reg_list_esp;
67
68  typedef struct Reg_Idx_Entry {
69      Const_String reg_name;
70      Int reg_idx;
71      Hash_Map_Slot hash_slot;
72  } Reg_Idx_Entry;
73
74  #define REG_IDX_ENTRY_OF(slot) HASH_MAP_ENTRY(slot, Reg_Idx_Entry, hash_slot)
75
76  static bool idx_reg_hash_compare(ssize_t reg_idx, Hash_Map_Slot *slot)
77  {
78      Reg_Idx_Entry *e = REG_IDX_ENTRY_OF(slot);
79      return e->reg_idx == reg_idx;
80  }
81
82  static bool reg_idx_hash_compare(String reg_name, Hash_Map_Slot *slot)
83  {
84      Reg_Idx_Entry *e = REG_IDX_ENTRY_OF(slot);
85      return !string_compare(reg_name, e->reg_name);
86  }
87
88  static void reg_idx_hash_destroy(Hash_Map_Slot *s)
89  {
90      free_mem(REG_IDX_ENTRY_OF(s));
91  }
92

```

```

93 HASH_MAP_SIZE(reg_idx_to_lo8bit_name, HASH_MAP_SIZE_11,
94               (Hash_Map_Comparator) idx_reg_hash_compare);
95
96 HASH_MAP_SIZE(reg_idx_to_32bit_name, HASH_MAP_SIZE_11,
97               (Hash_Map_Comparator) idx_reg_hash_compare);
98
99 HASH_MAP_SIZE(reg_name_to_idx, HASH_MAP_SIZE_23,
100              (Hash_Map_Comparator) reg_idx_hash_compare);
101
102 Int x86_32_reg_name_get_idx(Const_String reg_name)
103 {
104     Uns hash = string_hash_code(reg_name);
105     Hash_Map_Slot *slot = hash_map_get(&reg_name_to_idx, (String)reg_name,
106                                       hash);
107     if (!slot)
108         return -1;
109     return REG_IDX_ENTRY_OF(slot)->reg_idx;
110 }
111
112 Const_String x86_32_reg_idx_get_lo8bit_name(Int idx)
113 {
114     Hash_Map_Slot *slot = hash_map_get(&reg_idx_to_lo8bit_name,
115                                       INT_TO_PTR(idx), idx);
116     if (!slot)
117         return NULL;
118     return REG_IDX_ENTRY_OF(slot)->reg_name;
119 }
120
121 Const_String x86_32_reg_idx_get_32bit_name(Int idx)
122 {
123     Hash_Map_Slot *slot = hash_map_get(&reg_idx_to_32bit_name,
124                                       INT_TO_PTR(idx), idx);
125     if (!slot)
126         return NULL;
127     return REG_IDX_ENTRY_OF(slot)->reg_name;
128 }
129
130 typedef struct Reg_Str_Entry {
131     Const_String reg;
132     Hash_Map_Slot hash_slot;
133 } Reg_Str_Entry;
134
135 #define REG_STR_ENTRY_OF(slot) HASH_MAP_ENTRY(slot, Reg_Str_Entry, hash_slot)
136
137 static bool reg_str_map_compare(String search_reg, Hash_Map_Slot *s)
138 {
139     Reg_Str_Entry *e = REG_STR_ENTRY_OF(s);
140     return !string_compare(e->reg, search_reg);
141 }
142
143 static HASH_MAP(reg_str_map, (Hash_Map_Comparator) reg_str_map_compare);
144
145 static void reg_str_map_hash_destroy(Hash_Map_Slot *s)
146 {
147     free_mem(REG_STR_ENTRY_OF(s));
148 }
149
150 #define X86_32_REG_LIST_OF(slot) \
151     HASH_MAP_ENTRY(slot, X86_32_Reg_List, hash_slot)
152
153 static bool reg_list_map_compare(String search_reg, Hash_Map_Slot *s)
154 {
155     X86_32_Reg_List *list = X86_32_REG_LIST_OF(s);
156     return !string_compare(aia_operand_reg_get_name(list->reg_list[0]),
157                           search_reg);
158 }
159
160 static HASH_MAP(reg_list_map, (Hash_Map_Comparator) reg_list_map_compare);
161
162 static void reg_list_map_hash_destroy(Hash_Map_Slot *s)
163 {
164     free_mem(X86_32_REG_LIST_OF(s));
165 }
166

```



```

167 void x86_32_regs_init(Aia *aia)
168 {
169     Uns hash1, hash2, hash3;
170     Reg_Str_Entry *str_e;
171
172     reg_eax = aia_operand_reg_alloc(aia, reg_eax_str);
173     str_e = ALLOC_NEW(Reg_Str_Entry);
174     str_e->reg = reg_eax_str;
175     hash1 = string_hash_code(reg_eax_str);
176     hash_map_insert(&reg_str_map, &str_e->hash_slot, hash1);
177     __aia_operand_acquire(reg_eax);
178
179     reg_al = aia_operand_reg_alloc(aia, reg_al_str);
180     str_e = ALLOC_NEW(Reg_Str_Entry);
181     str_e->reg = reg_al_str;
182     hash2 = string_hash_code(reg_al_str);
183     hash_map_insert(&reg_str_map, &str_e->hash_slot, hash2);
184     __aia_operand_acquire(reg_al);
185
186     reg_ah = aia_operand_reg_alloc(aia, reg_ah_str);
187     str_e = ALLOC_NEW(Reg_Str_Entry);
188     str_e->reg = reg_ah_str;
189     hash3 = string_hash_code(reg_ah_str);
190     hash_map_insert(&reg_str_map, &str_e->hash_slot, hash3);
191     __aia_operand_acquire(reg_ah);
192
193     reg_list_eax = alloc_mem(sizeof(X86_32_Reg_List) +
194                             sizeof(Aia_Operand *) * 3);
195     reg_list_eax->num_regs = 3;
196     reg_list_eax->reg_list[0] = reg_eax;
197     reg_list_eax->reg_list[1] = reg_al;
198     reg_list_eax->reg_list[2] = reg_ah;
199     hash_map_insert(&reg_list_map, &reg_list_eax->hash_slot, hash1);
200
201     reg_list_al = alloc_mem(sizeof(X86_32_Reg_List) +
202                             sizeof(Aia_Operand *) * 2);
203     reg_list_al->num_regs = 2;
204     reg_list_al->reg_list[0] = reg_al;
205     reg_list_al->reg_list[1] = reg_eax;
206     hash_map_insert(&reg_list_map, &reg_list_al->hash_slot, hash2);
207
208     reg_list_ah = alloc_mem(sizeof(X86_32_Reg_List) +
209                             sizeof(Aia_Operand *) * 2);
210     reg_list_ah->num_regs = 2;
211     reg_list_ah->reg_list[0] = reg_ah;
212     reg_list_ah->reg_list[1] = reg_eax;
213     hash_map_insert(&reg_list_map, &reg_list_ah->hash_slot, hash3);
214
215     reg_edx = aia_operand_reg_alloc(aia, reg_edx_str);
216     str_e = ALLOC_NEW(Reg_Str_Entry);
217     str_e->reg = reg_edx_str;
218     hash1 = string_hash_code(reg_edx_str);
219     hash_map_insert(&reg_str_map, &str_e->hash_slot, hash1);
220     __aia_operand_acquire(reg_edx);
221
222     reg_dl = aia_operand_reg_alloc(aia, reg_dl_str);
223     str_e = ALLOC_NEW(Reg_Str_Entry);
224     str_e->reg = reg_dl_str;
225     hash2 = string_hash_code(reg_dl_str);
226     hash_map_insert(&reg_str_map, &str_e->hash_slot, hash2);
227     __aia_operand_acquire(reg_dl);
228
229     reg_dh = aia_operand_reg_alloc(aia, reg_dh_str);
230     str_e = ALLOC_NEW(Reg_Str_Entry);
231     str_e->reg = reg_dh_str;
232     hash3 = string_hash_code(reg_dh_str);
233     hash_map_insert(&reg_str_map, &str_e->hash_slot, hash3);
234     __aia_operand_acquire(reg_dh);
235
236     reg_list_edx = alloc_mem(sizeof(X86_32_Reg_List) +
237                             sizeof(Aia_Operand *) * 3);
238     reg_list_edx->num_regs = 3;
239     reg_list_edx->reg_list[0] = reg_edx;
240     reg_list_edx->reg_list[1] = reg_dl;

```

```

241     reg_list_edx->reg_list[2] = reg_dh;
242     hash_map_insert(&reg_list_map, &reg_list_edx->hash_slot, hash1);
243
244     reg_list_dl = alloc_mem(sizeof(X86_32_Reg_List) +
245                             sizeof(Aia_Operand *) * 2);
246     reg_list_dl->num_regs = 2;
247     reg_list_dl->reg_list[0] = reg_dl;
248     reg_list_dl->reg_list[1] = reg_edx;
249     hash_map_insert(&reg_list_map, &reg_list_dl->hash_slot, hash2);
250
251     reg_list_dh = alloc_mem(sizeof(X86_32_Reg_List) +
252                             sizeof(Aia_Operand *) * 2);
253     reg_list_dh->num_regs = 2;
254     reg_list_dh->reg_list[0] = reg_dh;
255     reg_list_dh->reg_list[1] = reg_edx;
256     hash_map_insert(&reg_list_map, &reg_list_dh->hash_slot, hash3);
257
258     reg_ecx = aia_operand_reg_alloc(aia, reg_ecx_str);
259     str_e = ALLOC_NEW(Reg_Str_Entry);
260     str_e->reg = reg_ecx_str;
261     hash1 = string_hash_code(reg_ecx_str);
262     hash_map_insert(&reg_str_map, &str_e->hash_slot, hash1);
263     __aia_operand_acquire(reg_ecx);
264
265     reg_cl = aia_operand_reg_alloc(aia, reg_cl_str);
266     str_e = ALLOC_NEW(Reg_Str_Entry);
267     str_e->reg = reg_cl_str;
268     hash2 = string_hash_code(reg_cl_str);
269     hash_map_insert(&reg_str_map, &str_e->hash_slot, hash2);
270     __aia_operand_acquire(reg_cl);
271
272     reg_ch = aia_operand_reg_alloc(aia, reg_ch_str);
273     str_e = ALLOC_NEW(Reg_Str_Entry);
274     str_e->reg = reg_ch_str;
275     hash3 = string_hash_code(reg_ch_str);
276     hash_map_insert(&reg_str_map, &str_e->hash_slot, hash3);
277     __aia_operand_acquire(reg_ch);
278
279     reg_list_ecx = alloc_mem(sizeof(X86_32_Reg_List) +
280                             sizeof(Aia_Operand *) * 3);
281     reg_list_ecx->num_regs = 3;
282     reg_list_ecx->reg_list[0] = reg_ecx;
283     reg_list_ecx->reg_list[1] = reg_cl;
284     reg_list_ecx->reg_list[2] = reg_ch;
285     hash_map_insert(&reg_list_map, &reg_list_ecx->hash_slot, hash1);
286
287     reg_list_cl = alloc_mem(sizeof(X86_32_Reg_List) +
288                             sizeof(Aia_Operand *) * 2);
289     reg_list_cl->num_regs = 2;
290     reg_list_cl->reg_list[0] = reg_cl;
291     reg_list_cl->reg_list[1] = reg_ecx;
292     hash_map_insert(&reg_list_map, &reg_list_cl->hash_slot, hash2);
293
294     reg_list_ch = alloc_mem(sizeof(X86_32_Reg_List) +
295                             sizeof(Aia_Operand *) * 2);
296     reg_list_ch->num_regs = 2;
297     reg_list_ch->reg_list[0] = reg_ch;
298     reg_list_ch->reg_list[1] = reg_ecx;
299     hash_map_insert(&reg_list_map, &reg_list_ch->hash_slot, hash3);
300
301     reg_ebx = aia_operand_reg_alloc(aia, reg_ebx_str);
302     str_e = ALLOC_NEW(Reg_Str_Entry);
303     str_e->reg = reg_ebx_str;
304     hash1 = string_hash_code(reg_ebx_str);
305     hash_map_insert(&reg_str_map, &str_e->hash_slot, hash1);
306     __aia_operand_acquire(reg_ebx);
307
308     reg_bl = aia_operand_reg_alloc(aia, reg_bl_str);
309     str_e = ALLOC_NEW(Reg_Str_Entry);
310     str_e->reg = reg_bl_str;
311     hash2 = string_hash_code(reg_bl_str);
312     hash_map_insert(&reg_str_map, &str_e->hash_slot, hash2);
313     __aia_operand_acquire(reg_bl);
314

```

```

315     reg_bh = aia_operand_reg_alloc(aia, reg_bh_str);
316     str_e = ALLOC_NEW(Reg_Str_Entry);
317     str_e->reg = reg_bh_str;
318     hash3 = string_hash_code(reg_bh_str);
319     hash_map_insert(&reg_str_map, &str_e->hash_slot, hash3);
320     __aia_operand_acquire(reg_bh);
321
322     reg_list_ebx = alloc_mem(sizeof(X86_32_Reg_List) +
323                             sizeof(Aia_Operand *) * 3);
324     reg_list_ebx->num_regs = 3;
325     reg_list_ebx->reg_list[0] = reg_ebx;
326     reg_list_ebx->reg_list[1] = reg_bl;
327     reg_list_ebx->reg_list[2] = reg_bh;
328     hash_map_insert(&reg_list_map, &reg_list_ebx->hash_slot, hash1);
329
330     reg_list_bl = alloc_mem(sizeof(X86_32_Reg_List) +
331                             sizeof(Aia_Operand *) * 2);
332     reg_list_bl->num_regs = 2;
333     reg_list_bl->reg_list[0] = reg_bl;
334     reg_list_bl->reg_list[1] = reg_ebx;
335     hash_map_insert(&reg_list_map, &reg_list_bl->hash_slot, hash2);
336
337     reg_list_bh = alloc_mem(sizeof(X86_32_Reg_List) +
338                             sizeof(Aia_Operand *) * 2);
339     reg_list_bh->num_regs = 2;
340     reg_list_bh->reg_list[0] = reg_bh;
341     reg_list_bh->reg_list[1] = reg_ebx;
342     hash_map_insert(&reg_list_map, &reg_list_bh->hash_slot, hash3);
343
344     reg_esi = aia_operand_reg_alloc(aia, reg_esi_str);
345     str_e = ALLOC_NEW(Reg_Str_Entry);
346     str_e->reg = reg_esi_str;
347     hash1 = string_hash_code(reg_esi_str);
348     hash_map_insert(&reg_str_map, &str_e->hash_slot, hash1);
349     __aia_operand_acquire(reg_esi);
350
351     reg_list_esi = alloc_mem(sizeof(X86_32_Reg_List) + sizeof(Aia_Operand *));
352     reg_list_esi->num_regs = 1;
353     reg_list_esi->reg_list[0] = reg_esi;
354     hash_map_insert(&reg_list_map, &reg_list_esi->hash_slot, hash1);
355
356     reg_edi = aia_operand_reg_alloc(aia, reg_edi_str);
357     str_e = ALLOC_NEW(Reg_Str_Entry);
358     str_e->reg = reg_edi_str;
359     hash1 = string_hash_code(reg_edi_str);
360     hash_map_insert(&reg_str_map, &str_e->hash_slot, hash1);
361     __aia_operand_acquire(reg_edi);
362
363     reg_list_edi = alloc_mem(sizeof(X86_32_Reg_List) + sizeof(Aia_Operand *));
364     reg_list_edi->num_regs = 1;
365     reg_list_edi->reg_list[0] = reg_edi;
366     hash_map_insert(&reg_list_map, &reg_list_edi->hash_slot, hash1);
367
368     reg_ebp = aia_operand_reg_alloc(aia, reg_ebp_str);
369     str_e = ALLOC_NEW(Reg_Str_Entry);
370     str_e->reg = reg_ebp_str;
371     hash1 = string_hash_code(reg_ebp_str);
372     hash_map_insert(&reg_str_map, &str_e->hash_slot, hash1);
373     __aia_operand_acquire(reg_ebp);
374
375     reg_list_ebp = alloc_mem(sizeof(X86_32_Reg_List) + sizeof(Aia_Operand *));
376     reg_list_ebp->num_regs = 1;
377     reg_list_ebp->reg_list[0] = reg_ebp;
378     hash_map_insert(&reg_list_map, &reg_list_ebp->hash_slot, hash1);
379
380     reg_esp = aia_operand_reg_alloc(aia, reg_esp_str);
381     str_e = ALLOC_NEW(Reg_Str_Entry);
382     str_e->reg = reg_esp_str;
383     hash1 = string_hash_code(reg_esp_str);
384     hash_map_insert(&reg_str_map, &str_e->hash_slot, hash1);
385     __aia_operand_acquire(reg_esp);
386
387     reg_list_esp = alloc_mem(sizeof(X86_32_Reg_List) + sizeof(Aia_Operand *));
388     reg_list_esp->num_regs = 1;

```

```

389     reg_list_esp->reg_list[0] = reg_esp;
390     hash_map_insert(&reg_list_map, &reg_list_esp->hash_slot, hash1);
391
392     /* Important. The given register indices are significant. */
393
394     Reg_Idx_Entry *regi_en;
395     Const_String reg_name;
396
397     reg_name = reg_eax_str;
398     regi_en = ALLOC_NEW(Reg_Idx_Entry);
399     regi_en->reg_name = reg_name;
400     regi_en->reg_idx = 0;
401     hash_map_insert(&reg_name_to_idx, &regi_en->hash_slot,
402                     string_hash_code(reg_name));
403
404     reg_name = reg_al_str;
405     regi_en = ALLOC_NEW(Reg_Idx_Entry);
406     regi_en->reg_name = reg_name;
407     regi_en->reg_idx = 0;
408     hash_map_insert(&reg_name_to_idx, &regi_en->hash_slot,
409                     string_hash_code(reg_name));
410
411     reg_name = reg_ah_str;
412     regi_en = ALLOC_NEW(Reg_Idx_Entry);
413     regi_en->reg_name = reg_name;
414     regi_en->reg_idx = 0;
415     hash_map_insert(&reg_name_to_idx, &regi_en->hash_slot,
416                     string_hash_code(reg_name));
417
418     reg_name = reg_edx_str;
419     regi_en = ALLOC_NEW(Reg_Idx_Entry);
420     regi_en->reg_name = reg_name;
421     regi_en->reg_idx = 1;
422     hash_map_insert(&reg_name_to_idx, &regi_en->hash_slot,
423                     string_hash_code(reg_name));
424
425     reg_name = reg_dl_str;
426     regi_en = ALLOC_NEW(Reg_Idx_Entry);
427     regi_en->reg_name = reg_name;
428     regi_en->reg_idx = 1;
429     hash_map_insert(&reg_name_to_idx, &regi_en->hash_slot,
430                     string_hash_code(reg_name));
431
432     reg_name = reg_dh_str;
433     regi_en = ALLOC_NEW(Reg_Idx_Entry);
434     regi_en->reg_name = reg_name;
435     regi_en->reg_idx = 1;
436     hash_map_insert(&reg_name_to_idx, &regi_en->hash_slot,
437                     string_hash_code(reg_name));
438
439     reg_name = reg_ecx_str;
440     regi_en = ALLOC_NEW(Reg_Idx_Entry);
441     regi_en->reg_name = reg_name;
442     regi_en->reg_idx = 2;
443     hash_map_insert(&reg_name_to_idx, &regi_en->hash_slot,
444                     string_hash_code(reg_name));
445
446     reg_name = reg_cl_str;
447     regi_en = ALLOC_NEW(Reg_Idx_Entry);
448     regi_en->reg_name = reg_name;
449     regi_en->reg_idx = 2;
450     hash_map_insert(&reg_name_to_idx, &regi_en->hash_slot,
451                     string_hash_code(reg_name));
452
453     reg_name = reg_ch_str;
454     regi_en = ALLOC_NEW(Reg_Idx_Entry);
455     regi_en->reg_name = reg_name;
456     regi_en->reg_idx = 2;
457     hash_map_insert(&reg_name_to_idx, &regi_en->hash_slot,
458                     string_hash_code(reg_name));
459
460     reg_name = reg_ebx_str;
461     regi_en = ALLOC_NEW(Reg_Idx_Entry);
462     regi_en->reg_name = reg_name;

```

```

463     regi_en->reg_idx = 3;
464     hash_map_insert(&reg_name_to_idx, &regi_en->hash_slot,
465         string_hash_code(reg_name));
466
467     reg_name = reg_bl_str;
468     regi_en = ALLOC_NEW(Reg_Idx_Entry);
469     regi_en->reg_name = reg_name;
470     regi_en->reg_idx = 3;
471     hash_map_insert(&reg_name_to_idx, &regi_en->hash_slot,
472         string_hash_code(reg_name));
473
474     reg_name = reg_bh_str;
475     regi_en = ALLOC_NEW(Reg_Idx_Entry);
476     regi_en->reg_name = reg_name;
477     regi_en->reg_idx = 3;
478     hash_map_insert(&reg_name_to_idx, &regi_en->hash_slot,
479         string_hash_code(reg_name));
480
481     reg_name = reg_esi_str;
482     regi_en = ALLOC_NEW(Reg_Idx_Entry);
483     regi_en->reg_name = reg_name;
484     regi_en->reg_idx = 4;
485     hash_map_insert(&reg_name_to_idx, &regi_en->hash_slot,
486         string_hash_code(reg_name));
487
488     reg_name = reg_edi_str;
489     regi_en = ALLOC_NEW(Reg_Idx_Entry);
490     regi_en->reg_name = reg_name;
491     regi_en->reg_idx = 5;
492     hash_map_insert(&reg_name_to_idx, &regi_en->hash_slot,
493         string_hash_code(reg_name));
494
495     reg_name = reg_ebp_str;
496     regi_en = ALLOC_NEW(Reg_Idx_Entry);
497     regi_en->reg_name = reg_name;
498     regi_en->reg_idx = 6;
499     hash_map_insert(&reg_name_to_idx, &regi_en->hash_slot,
500         string_hash_code(reg_name));
501
502     regi_en = ALLOC_NEW(Reg_Idx_Entry);
503     regi_en->reg_name = reg_eax_str;
504     regi_en->reg_idx = 0;
505     hash_map_insert(&reg_idx_to_32bit_name, &regi_en->hash_slot,
506         (Uns)regi_en->reg_idx);
507
508     regi_en = ALLOC_NEW(Reg_Idx_Entry);
509     regi_en->reg_name = reg_edx_str;
510     regi_en->reg_idx = 1;
511     hash_map_insert(&reg_idx_to_32bit_name, &regi_en->hash_slot,
512         (Uns)regi_en->reg_idx);
513
514     regi_en = ALLOC_NEW(Reg_Idx_Entry);
515     regi_en->reg_name = reg_ecx_str;
516     regi_en->reg_idx = 2;
517     hash_map_insert(&reg_idx_to_32bit_name, &regi_en->hash_slot,
518         (Uns)regi_en->reg_idx);
519
520     regi_en = ALLOC_NEW(Reg_Idx_Entry);
521     regi_en->reg_name = reg_ebx_str;
522     regi_en->reg_idx = 3;
523     hash_map_insert(&reg_idx_to_32bit_name, &regi_en->hash_slot,
524         (Uns)regi_en->reg_idx);
525
526     regi_en = ALLOC_NEW(Reg_Idx_Entry);
527     regi_en->reg_name = reg_esi_str;
528     regi_en->reg_idx = 4;
529     hash_map_insert(&reg_idx_to_32bit_name, &regi_en->hash_slot,
530         (Uns)regi_en->reg_idx);
531
532     regi_en = ALLOC_NEW(Reg_Idx_Entry);
533     regi_en->reg_name = reg_edi_str;
534     regi_en->reg_idx = 5;
535     hash_map_insert(&reg_idx_to_32bit_name, &regi_en->hash_slot,
536         (Uns)regi_en->reg_idx);

```

```

537
538     regi_en = ALLOC_NEW(Reg_Idx_Entry);
539     regi_en->reg_name = reg_ebp_str;
540     regi_en->reg_idx = 6;
541     hash_map_insert(&reg_idx_to_32bit_name, &regi_en->hash_slot,
542         (Uns)regi_en->reg_idx);
543
544     regi_en = ALLOC_NEW(Reg_Idx_Entry);
545     regi_en->reg_name = reg_al_str;
546     regi_en->reg_idx = 0;
547     hash_map_insert(&reg_idx_to_lo8bit_name, &regi_en->hash_slot,
548         (Uns)regi_en->reg_idx);
549
550     regi_en = ALLOC_NEW(Reg_Idx_Entry);
551     regi_en->reg_name = reg_dl_str;
552     regi_en->reg_idx = 1;
553     hash_map_insert(&reg_idx_to_lo8bit_name, &regi_en->hash_slot,
554         (Uns)regi_en->reg_idx);
555
556     regi_en = ALLOC_NEW(Reg_Idx_Entry);
557     regi_en->reg_name = reg_cl_str;
558     regi_en->reg_idx = 2;
559     hash_map_insert(&reg_idx_to_lo8bit_name, &regi_en->hash_slot,
560         (Uns)regi_en->reg_idx);
561
562     regi_en = ALLOC_NEW(Reg_Idx_Entry);
563     regi_en->reg_name = reg_bl_str;
564     regi_en->reg_idx = 3;
565     hash_map_insert(&reg_idx_to_lo8bit_name, &regi_en->hash_slot,
566         (Uns)regi_en->reg_idx);
567 }
568
569 void x86_32_regs_release()
570 {
571     __aia_operand_release(reg_eax);
572     __aia_operand_release(reg_al);
573     __aia_operand_release(reg_ah);
574
575     __aia_operand_release(reg_edx);
576     __aia_operand_release(reg_dl);
577     __aia_operand_release(reg_dh);
578
579     __aia_operand_release(reg_ecx);
580     __aia_operand_release(reg_cl);
581     __aia_operand_release(reg_ch);
582
583     __aia_operand_release(reg_ebx);
584     __aia_operand_release(reg_bx);
585     __aia_operand_release(reg_bh);
586
587     __aia_operand_release(reg_esi);
588     __aia_operand_release(reg_edi);
589     __aia_operand_release(reg_ebp);
590     __aia_operand_release(reg_esp);
591
592     hash_map_for_each_destroy(&reg_str_map, reg_str_map_hash_destroy);
593     hash_map_for_each_destroy(&reg_list_map, reg_list_map_hash_destroy);
594
595     hash_map_for_each_destroy(&reg_name_to_idx, reg_idx_hash_destroy);
596     hash_map_for_each_destroy(&reg_idx_to_32bit_name, reg_idx_hash_destroy);
597     hash_map_for_each_destroy(&reg_idx_to_lo8bit_name, reg_idx_hash_destroy);
598 }
599
600 bool x86_32_is_concrete_reg_str(Const_String str)
601 {
602     return hash_map_contains(&reg_str_map, (String)str, string_hash_code(str));
603 }
604
605 bool x86_32_is_concrete_reg(Aia_Operand *op)
606 {
607     if (!aia_operand_get_type(op) == AIA_OPERAND_REG)
608         return false;
609
610     return x86_32_is_concrete_reg_str(aia_operand_reg_get_name(op));

```

```

611 }
612
613 Aia_Operand *x86_32_get_reg_operand(Const_String reg_str)
614 {
615     X86_32_Reg_List *list = x86_32_get_reg_list(reg_str);
616     if (!list)
617         return NULL;
618     return list->reg_list[0];
619 }
620
621 X86_32_Reg_List *x86_32_get_reg_list(Const_String reg)
622 {
623     Hash_Map_Slot *slot = hash_map_get(&reg_list_map, (String)reg,
624                                         string_hash_code(reg));
625     if (slot)
626         return X86_32_REG_LIST_OF(slot);
627
628     return NULL;
629 }
630
631 bool x86_32_is_callee_save_reg_str(Const_String str)
632 {
633     Int idx = x86_32_reg_name_get_idx(str);
634     if (idx <= 2)
635         return false;
636     return true;
637 }

```

:

A.8.20 src/x86_32/x86_32_regs.h

```

1  #ifndef X86_32_REGS_H
2  #define X86_32_REGS_H
3
4  #include <aia/aia.h>
5
6  #define X86_32_REG_COUNT 7
7  #define X86_32_REG_COUNT_8BIT (X86_32_REG_COUNT - 3)
8
9  typedef struct X86_32_Reg_List {
10     Uns num_regs;
11     Hash_Map_Slot hash_slot;
12     Aia_Operand *reg_list[0];
13 } X86_32_Reg_List;
14
15 extern Const_String reg_eax_str;
16 extern Const_String reg_al_str;
17 extern Const_String reg_ah_str;
18
19 extern Const_String reg_ebx_str;
20 extern Const_String reg_bl_str;
21 extern Const_String reg_bh_str;
22
23 extern Const_String reg_ecx_str;
24 extern Const_String reg_cl_str;
25 extern Const_String reg_ch_str;
26
27 extern Const_String reg_edx_str;
28 extern Const_String reg_dl_str;
29 extern Const_String reg_dh_str;
30
31 extern Const_String reg_esi_str;
32 extern Const_String reg_edi_str;
33 extern Const_String reg_ebp_str;
34 extern Const_String reg_esp_str;
35
36 extern Aia_Operand *reg_eax;
37 extern Aia_Operand *reg_al;
38 extern Aia_Operand *reg_ah;
39

```

```

40 extern Aia_Operand *reg_ebx;
41 extern Aia_Operand *reg_bl;
42 extern Aia_Operand *reg_bh;
43
44 extern Aia_Operand *reg_ecx;
45 extern Aia_Operand *reg_cl;
46 extern Aia_Operand *reg_ch;
47
48 extern Aia_Operand *reg_edx;
49 extern Aia_Operand *reg_dl;
50 extern Aia_Operand *reg_dh;
51
52 extern Aia_Operand *reg_esi;
53 extern Aia_Operand *reg_edi;
54 extern Aia_Operand *reg_ebp;
55 extern Aia_Operand *reg_esp;
56
57 extern X86_32_Reg_List *reg_list_eax;
58 extern X86_32_Reg_List *reg_list_al;
59 extern X86_32_Reg_List *reg_list_ah;
60
61 extern X86_32_Reg_List *reg_list_ebx;
62 extern X86_32_Reg_List *reg_list_bl;
63 extern X86_32_Reg_List *reg_list_bh;
64
65 extern X86_32_Reg_List *reg_list_ecx;
66 extern X86_32_Reg_List *reg_list_cl;
67 extern X86_32_Reg_List *reg_list_ch;
68
69 extern X86_32_Reg_List *reg_list_dli;
70 extern X86_32_Reg_List *reg_list_dh;
71 extern X86_32_Reg_List *reg_list_edx;
72
73 extern X86_32_Reg_List *reg_list_esi;
74 extern X86_32_Reg_List *reg_list_edi;
75 extern X86_32_Reg_List *reg_list_ebp;
76 extern X86_32_Reg_List *reg_list_esp;
77
78 void x86_32_regs_init(Aia *aia);
79
80 void x86_32_regs_release();
81
82 bool x86_32_is_callee_save_reg_str(Const_String str);
83
84 static inline bool x86_32_is_callee_save_reg(Aia_Operand *reg)
85 {
86     return x86_32_is_callee_save_reg_str(aia_operand_reg_get_name(reg));
87 }
88
89 Int x86_32_reg_name_get_idx(Const_String reg_name);
90
91 static inline Int x86_32_reg_get_idx(Aia_Operand *reg)
92 {
93     return x86_32_reg_name_get_idx(aia_operand_reg_get_name(reg));
94 }
95
96 Const_String x86_32_reg_idx_get_lo8bit_name(Int idx);
97
98 Const_String x86_32_reg_idx_get_32bit_name(Int idx);
99
100 bool x86_32_is_concrete_reg_str(Const_String str);
101
102 bool x86_32_is_concrete_reg(Aia_Operand *op);
103
104 Aia_Operand *x86_32_get_reg_operand(Const_String reg_str);
105
106 X86_32_Reg_List *x86_32_get_reg_list(Const_String reg);
107
108 #endif // X86_32_REGS_H

```

:

A.8.21 src/x86_32/x86_32_reg_alloc.h

```

1  #ifndef X86_32_REG_ALLOC_H
2  #define X86_32_REG_ALLOC_H
3
4  #include "x86_32_reg_alloc_color.h"
5
6  /* Does register allocation.
7   * repl_map is mapping temporary registers to memory locations
8   * which should be used as spill location. */
9  static inline void x86_32_reg_alloc(Aia *aia, Aia_Operand_Map *repl_map)
10 {
11     x86_32_reg_alloc_color(aia, repl_map);
12 }
13
14 #endif // X86_32_REG_ALLOC_H

```

:

A.8.22 src/x86_32/x86_32_reg_alloc_color.c

```

1  #include "x86_32_regs.h"
2  #include "x86_32_normalize.h"
3  #include <hash_map.h>
4  #include <string_builder.h>
5  #include <aia/aia_operand_map.h>
6  #include <aia/aia_normalize_addr.h>
7  #include <main.h>
8
9  #undef DEBUG_TYPE
10 #define DEBUG_TYPE liveness-alloc
11
12 typedef struct Live_Node {
13     Aia_Operand *init_operand;
14     Const_String init_var_name;
15     Const_String assigned_var_name;
16     Vector interferences;
17     Hash_Map interference_map;
18     Double_List_Node dbnode;
19     Hash_Map_Slot graph_slot;
20     Hash_Map_Slot regraph_slot;
21     Int reg_idx;
22     uint8_t var_size;
23 } Live_Node;
24
25 typedef struct Live_Node_Inter_Entry {
26     Live_Node *node;
27     Hash_Map_Slot hash_slot;
28 } Live_Node_Inter_Entry;
29
30 #define LIVE_NODE_INTER_ENTRY_OF(slot) \
31     HASH_MAP_ENTRY(slot, Live_Node_Inter_Entry, hash_slot)
32
33 static inline Live_Node_Inter_Entry *live_node_inter_entry_alloc(Live_Node *n)
34 {
35     Live_Node_Inter_Entry *e = ALLOC_NEW(Live_Node_Inter_Entry);
36     e->node = n;
37     return e;
38 }
39
40 static void live_node_inter_hash_destroy(Hash_Map_Slot *s)
41 {
42     free_mem(LIVE_NODE_INTER_ENTRY_OF(s));
43 }
44
45 static inline void live_node_insert_interference(Live_Node *n,
46     Live_Node *inter)
47 {
48     Live_Node_Inter_Entry *e = live_node_inter_entry_alloc(inter);
49     hash_map_insert(&n->interference_map, &e->hash_slot,

```

```

50         hash_map_aligned_ptr_hash(inter));
51     }
52
53     static inline bool live_node_remove_interference(Live_Node *n,
54         Live_Node *inter)
55     {
56         Hash_Map_Slot *slot = hash_map_remove(&n->interference_map, inter,
57             hash_map_aligned_ptr_hash(inter));
58         if (!slot)
59             return false;
60
61         Live_Node_Inter_Entry *e = LIVE_NODE_INTER_ENTRY_OF(slot);
62         free_mem(e);
63         return true;
64     }
65
66     static inline bool live_node_has_interference(Live_Node *n, Live_Node *inter)
67     {
68         return hash_map_contains(&n->interference_map, inter,
69             hash_map_aligned_ptr_hash(inter));
70     }
71
72     static bool live_node_inter_entry_hash_compare(Live_Node *search_node,
73         Hash_Map_Slot *map_slot)
74     {
75         Live_Node_Inter_Entry *e = LIVE_NODE_INTER_ENTRY_OF(map_slot);
76         return search_node == e->node;
77     }
78
79     #define LIVE_NODE_FOR_EACH_INTERFERENCE(n, inter) \
80         for (Hash_Map_Slot *___s = INT_TO_PTR(1); ___s; ___s = NULL) \
81             HASH_MAP_FOR_EACH(&(n)->interference_map, ___s) \
82                 if ((inter = LIVE_NODE_INTER_ENTRY_OF(___s)->node) || !inter)
83
84     #define LIVE_NODE_OF_GRAPH_SLOT(slot) \
85         HASH_MAP_ENTRY(slot, Live_Node, graph_slot)
86
87     #define LIVE_NODE_OF_REGRAPH_SLOT(slot) \
88         HASH_MAP_ENTRY(slot, Live_Node, regraph_slot)
89
90     #define LIVE_NODE_OF_DBNODE(node) HASH_MAP_ENTRY(node, Live_Node, dbnode)
91
92     typedef struct Live_Graph {
93         Hash_Map live_node_map;
94         Hash_Map regraph_map;
95         Live_Node global_ebp;
96         Live_Node global_esi;
97         Live_Node global_edi;
98         Double_List assign_queue;
99         Double_List wait_queue;
100         Vector live_node_stack;
101         Vector spill_list;
102         bool free_regs[X86_32_REG_COUNT];
103     } Live_Graph;
104
105     static UNUSED void live_graph_print(Live_Graph *g)
106     {
107         DLOG("\nLIVE GRAPH:\n");
108         Hash_Map_Slot *slot01;
109         HASH_MAP_FOR_EACH(&g->live_node_map, slot01) {
110             Live_Node *ln = LIVE_NODE_OF_GRAPH_SLOT(slot01);
111             DLOG("%S => ", ln->init_var_name);
112             Live_Node *lni;
113             LIVE_NODE_FOR_EACH_INTERFERENCE(ln, lni)
114                 DLOG("%S, ", lni->init_var_name);
115             DLOG("\n");
116         }
117         DLOG("\n\n");
118     }
119
120     static bool live_node_regraph_compare(Live_Node *search_node,
121         Hash_Map_Slot *slot)
122     {
123         return search_node == LIVE_NODE_OF_REGRAPH_SLOT(slot);

```

```

124 }
125
126 static bool live_node_graph_compare(String search_var,
127     Hash_Map_Slot *map_slot)
128 {
129     Live_Node *n = LIVE_NODE_OF_GRAPH_SLOT(map_slot);
130     return !string_compare(n->init_var_name, search_var);
131 }
132
133 static inline Live_Node *live_graph_lookup_var(Live_Graph *g,
134     Const_String var_name)
135 {
136     Uns hash = string_hash_code(var_name);
137     Hash_Map_Slot *nslot = hash_map_get(&g->live_node_map,
138     (String)var_name, hash);
139     if (nslot)
140         return LIVE_NODE_OF_GRAPH_SLOT(nslot);
141     return NULL;
142 }
143
144 static inline Live_Node *live_graph_lookup(Live_Graph *g,
145     Aia_Operand *op)
146 {
147     assert(aia_operand_is_reg(op));
148     return live_graph_lookup_var(g, aia_operand_reg_get_name(op));
149 }
150
151 static inline bool live_node_add_interference(Live_Node *lhs, Live_Node *rhs);
152
153 static void live_node_set_init_var_name(Live_Node *n,
154     Const_String var_name)
155 {
156     n->init_var_name = var_name;
157     n->assigned_var_name = var_name;
158     n->reg_idx = x86_32_reg_name_get_idx(var_name);
159 }
160
161 static void live_node_set_assigned_var_name(Live_Node *n,
162     Int idx)
163 {
164     Const_String var_name;
165     if (n->var_size == AIA_BYTE)
166         var_name = x86_32_reg_idx_get_lo8bit_name(idx);
167     else
168         var_name = x86_32_reg_idx_get_32bit_name(idx);
169     assert(var_name);
170     n->reg_idx = idx;
171     n->assigned_var_name = var_name;
172 }
173
174 #if 0
175 static void live_node_default_8bit_interferences(Live_Node *n, Live_Graph *g)
176 {
177     live_node_add_interference(n, &g->global_ebp);
178     live_node_add_interference(n, &g->global_esi);
179     live_node_add_interference(n, &g->global_edi);
180 }
181 #endif
182
183 static void live_node_init(Live_Graph *g UNUSED, Live_Node *n,
184     Aia_Operand *op, uint8_t var_size)
185 {
186     assert(aia_operand_is_reg(op));
187     __aia_operand_acquire(op);
188     n->interferences = VECTOR_INIT();
189     n->interference_map = HASH_MAP_INIT_SIZE(HASH_MAP_SIZE_11,
190     (Hash_Map_Comparator)live_node_inter_entry_hash_compare);
191     n->var_size = var_size;
192     n->init_operand = op;
193     live_node_set_init_var_name(n, aia_operand_reg_get_name(op));

```

```

198
199 #if 0
200     if (var_size == AIA_BYTE)
201         live_node_default_8bit_interferences(n, g);
202 #endif
203 }
204
205 static inline Live_Node *__live_graph_get(Live_Graph *g, Aia_Operand *op,
206     uint8_t var_size)
207 {
208     Live_Node *n;
209
210     assert(aia_operand_is_reg(op));
211     Const_String var_name = aia_operand_reg_get_name(op);
212
213     Uns hash = string_hash_code(var_name);
214
215     Hash_Map_Slot *nslot = hash_map_get(&g->live_node_map,
216         (String)var_name, hash);
217     if (nslot)
218         return LIVE_NODE_OF_GRAPH_SLOT(nslot);
219
220     n = ALLOC_NEW(Live_Node);
221     live_node_init(g, n, op, var_size);
222
223     hash_map_insert(&g->live_node_map, &n->graph_slot, hash);
224
225     return n;
226 }
227
228 static inline void live_node_clear(Live_Node *n)
229 {
230     __aia_operand_release(n->init_operand);
231     hash_map_for_each_destroy(&n->interference_map,
232         live_node_inter_hash_destroy);
233     vector_clear(&n->interferences);
234 }
235
236 static inline void live_node_destroy(Live_Node *n)
237 {
238     live_node_clear(n);
239     free_mem(n);
240 }
241
242 static void live_node_graph_hash_destroy(Hash_Map_Slot *s)
243 {
244     live_node_destroy(LIVE_NODE_OF_GRAPH_SLOT(s));
245 }
246
247 static inline bool live_node_link(Live_Node *lhs, Live_Node *rhs)
248 {
249     if (lhs == rhs)
250         return false;
251
252     if (live_node_has_interference(lhs, rhs))
253         return false;
254
255     DLOG("add link %S and %S\n", lhs->init_var_name,
256         rhs->init_var_name);
257
258     Int lhs_i = x86_32_reg_name_get_idx(lhs->init_var_name);
259     if (lhs_i != -1) {
260         Int rhs_i = x86_32_reg_name_get_idx(rhs->init_var_name);
261         if (rhs_i == lhs_i)
262             return false;
263     }
264
265     live_node_insert_interference(lhs, rhs);
266     live_node_insert_interference(rhs, lhs);
267
268     return true;
269 }
270
271 static inline bool live_node_add_interference(Live_Node *lhs, Live_Node *rhs)

```

```

272 {
273     if (live_node_link(lhs, rhs)) {
274         vector_append(&lhs->interferences, rhs);
275         vector_append(&rhs->interferences, lhs);
276         return true;
277     }
278     return false;
279 }
280
281 static Live_Graph *live_graph_alloc()
282 {
283     Live_Graph *g = ALLOC_NEW(Live_Graph);
284     g->live_node_map = HASH_MAP_INIT(
285         (Hash_Map_Comparator)live_node_graph_compare);
286     g->assign_queue = DOUBLE_LIST_INIT(g->assign_queue);
287     g->wait_queue = DOUBLE_LIST_INIT(g->wait_queue);
288     g->live_node_stack = VECTOR_INIT_SIZE(100);
289     g->spill_list = VECTOR_INIT();
290     g->regraph_map = HASH_MAP_INIT(
291         (Hash_Map_Comparator)live_node_regraph_compare);
292
293     for (Uns i = 0; i < X86_32_REG_COUNT; i++)
294         g->free_regs[i] = true;
295
296     live_node_init(g, &g->global_ebp, reg_ebp, AIA_LONG);
297     live_node_init(g, &g->global_esi, reg_esi, AIA_LONG);
298     live_node_init(g, &g->global_edi, reg_edi, AIA_LONG);
299
300     return g;
301 }
302
303 static inline void live_graph_destroy(Live_Graph *g)
304 {
305     #if 0
306     hash_map_for_each_destroy(&g->reg_name_to_idx, reg_idx_hash_destroy);
307     hash_map_for_each_destroy(&g->reg_idx_to_lo8bit_name,
308         reg_idx_hash_destroy);
309     hash_map_for_each_destroy(&g->reg_idx_to_32bit_name, reg_idx_hash_destroy);
310     #endif
311     hash_map_for_each_destroy(&g->live_node_map, live_node_graph_hash_destroy);
312     hash_map_clear(&g->regraph_map);
313     vector_clear(&g->live_node_stack);
314     vector_clear(&g->spill_list);
315     live_node_clear(&g->global_esi);
316     live_node_clear(&g->global_edi);
317     live_node_clear(&g->global_ebp);
318     free_mem(g);
319 }
320
321 typedef Hash_Map Live_Set;
322
323 typedef struct Live_Set_Entry {
324     Const_String var_name;
325     Hash_Map_Slot hash_slot;
326 } Live_Set_Entry;
327
328 #define LIVE_SET_ENTRY_OF(slot) \
329     HASH_MAP_ENTRY(slot, Live_Set_Entry, hash_slot)
330
331 // Warning. Goto to break the loop.
332 #define LIVE_SET_FOR_EACH_VAR(live_set, vname) \
333     for (Hash_Map_Slot *___s = INT_TO_PTR(1); ___s; ___s = NULL) \
334     HASH_MAP_FOR_EACH(live_set, ___s) \
335     if ((vname = LIVE_SET_ENTRY_OF(___s)->var_name) || !vname)
336
337 // Warning. Goto to break the loop.
338 #define LIVE_SET_FOR_EACH_ENTRY(live_set, ent) \
339     for (Hash_Map_Slot *___s = INT_TO_PTR(1); ___s; ___s = NULL) \
340     HASH_MAP_FOR_EACH(live_set, ___s) \
341     if ((ent = LIVE_SET_ENTRY_OF(___s)) || !ent)
342
343 static inline bool live_set_remove_var(Live_Set *s, Const_String var)
344 {
345     Hash_Map_Slot *slot = hash_map_remove(s, (String)var,

```

```

346     string_hash_code(var));
347     if (slot) {
348         free_mem(LIVE_SET_ENTRY_OF(slot));
349         return true;
350     }
351     return false;
352 }
353
354 static inline bool live_set_remove_op(Live_Set *s, Aia_Operand *op)
355 {
356     if (!aia_operand_is_reg(op))
357         return false;
358     return live_set_remove_var(s, aia_operand_reg_get_name(op));
359 }
360
361 static inline bool live_set_contains(Live_Set *s, Const_String var)
362 {
363     return hash_map_contains(s, (String)var, string_hash_code(var));
364 }
365
366 static inline bool live_set_contains_op(Live_Set *s, Aia_Operand *op)
367 {
368     if (aia_operand_is_reg(op))
369         return live_set_contains(s, aia_operand_reg_get_name(op));
370     return false;
371 }
372
373 static inline bool is_operand_live_out(Aia_Instr *in, Aia_Operand *op)
374 {
375     if (!op)
376         return false;
377
378     Aia_Instr *suc;
379
380     switch (aia_operand_get_type(op)) {
381     case AIA_OPERAND_REG:
382         suc = aia_instr_get_sucessor(in);
383         if (suc) {
384             return live_set_contains(suc->meta_data,
385                                     aia_operand_reg_get_name(op));
386         } else {
387             Aia_Block *b = aia_instr_get_block(in);
388             Aia_Block *suc_b;
389             AIA_BLOCK_FOR_EACH_SUCESSOR(b, suc_b) {
390                 suc = __aia_block_peek_first_instr(suc_b);
391                 if (live_set_contains(suc->meta_data,
392                                     aia_operand_reg_get_name(op)))
393                     return true;
394             }
395             return false;
396         }
397     case AIA_OPERAND_ADDR_REF:
398         return true;
399     case AIA_OPERAND_DISPLAY_REF:
400         return true;
401     default:
402         return true;
403     }
404 }
405
406 static inline void live_set_remove_entry(Live_Set *s, Live_Set_Entry *e)
407 {
408     Uns hash_code = hash_map_slot_get_hash_code(&e->hash_slot);
409     assert(hash_map_contains(s, (String)e->var_name, hash_code));
410     Hash_Map_Slot *slot = hash_map_remove(s, (String)e->var_name, hash_code);
411     free_mem(LIVE_SET_ENTRY_OF(slot));
412 }
413
414 static inline bool __live_set_insert_var(Live_Set *s, Const_String var,
415                                         Uns hash_code)

```

```

420 {
421     if (hash_map_contains(s, (String)var, hash_code))
422         return false;
423     Live_Set_Entry *e = ALLOC_NEW(Live_Set_Entry);
424     e->var_name = var;
425     hash_map_insert(s, &e->hash_slot, hash_code);
426     return true;
427 }
428
429 static inline bool live_set_insert_var(Live_Set *s, Const_String var)
430 {
431     return __live_set_insert_var(s, var, string_hash_code(var));
432 }
433
434 static inline bool live_set_insert_op(Live_Set *s, Aia_Operand *op)
435 {
436     if (!aia_operand_is_reg(op))
437         return false;
438     return live_set_insert_var(s, aia_operand_reg_get_name(op));
439 }
440
441 static inline bool live_set_insert_entry(Live_Set *s, Live_Set_Entry *e)
442 {
443     return __live_set_insert_var(s, e->var_name,
444         hash_map_slot_get_hash_code(&e->hash_slot));
445 }
446
447 static bool live_set_entry_compare(String search_str, Hash_Map_Slot *live_slot)
448 {
449     Live_Set_Entry *e = LIVE_SET_ENTRY_OF(live_slot);
450     return !string_compare(search_str, e->var_name);
451 }
452
453 static inline Live_Set *live_set_alloc()
454 {
455     Live_Set *s = ALLOC_NEW(Live_Set);
456     *s = HASH_MAP_INIT_SIZE(HASH_MAP_SIZE_11,
457         (Hash_Map_Comparator) live_set_entry_compare);
458     return s;
459 }
460
461 static void live_set_entry_hash_destroy(Hash_Map_Slot *slot)
462 {
463     free_mem(LIVE_SET_ENTRY_OF(slot));
464 }
465
466 static inline void live_set_destroy(Live_Set *s)
467 {
468     if (s) {
469         hash_map_for_each_destroy(s, live_set_entry_hash_destroy);
470         free_mem(s);
471     }
472 }
473
474 static bool x86_32_instr_defines_op(Aia_Instr *in, Const_String op_name)
475 {
476     Aia_Operand *dest = aia_instr_get_dest_op(in);
477     if (!dest)
478         return false;
479
480     switch (aia_operand_get_type(dest)) {
481     case AIA_OPERAND_REG:
482         return !string_compare(op_name, aia_operand_reg_get_name(dest));
483
484     case AIA_OPERAND_LABEL:
485         return false;
486
487     case AIA_OPERAND_LOCAL_REF:
488         return false;
489
490     case AIA_OPERAND_ARG:
491         return false;
492
493     case AIA_OPERAND_DISPLAY_REF:

```

```

494     /* Fall through. */
495     case AIA_OPERAND_ADDR_REF:
496         return false;
497
498     default:
499         fatal_error(S("unexpected dest instruction operand type. "
500                     "Aborting...\n"));
501     }
502 }
503
504 static void x86_32_instr_add_live_vars(Aia_Instr *in, Live_Set *vars, Aia *aia)
505 {
506     Live_Set_Entry *e;
507     LIVE_SET_FOR_EACH_ENTRY(vars, e) {
508         if (!x86_32_instr_defines_op(in, e->var_name)) {
509             if (live_set_insert_entry(in->meta_data, e))
510                 aia->meta_data = INT_TO_PTR(1);
511         }
512     }
513 }
514
515 static void x86_32_instr_reg_alloc(Aia_Instr *in, Aia_Block *b, Aia *aia)
516 {
517     Aia_Instr *pred = aia_instr_get_predecessor(in);
518     if (!pred) {
519         Aia_Block *pred_block;
520         AIA_BLOCK_FOR_EACH_PREDECESSOR(b, pred_block)
521             x86_32_instr_add_live_vars(__aia_block_peek_last_instr(
522                 pred_block), in->meta_data, aia);
523     } else {
524         x86_32_instr_add_live_vars(pred, in->meta_data, aia);
525     }
526 }
527
528 static void x86_32_block_reg_alloc(Aia_Block *b, Aia *aia)
529 {
530     Aia_Instr *in;
531     AIA_BLOCK_FOR_EACH_INSTRUCTION_REVERSED(b, in)
532         x86_32_instr_reg_alloc(in, b, aia);
533 }
534
535 static void x86_32_section_reg_alloc(Aia *aia, Aia_Section *sec)
536 {
537     Aia_Block *b;
538     AIA_SECTION_FOR_EACH_BLOCK_REVERSED(sec, b)
539         x86_32_block_reg_alloc(b, aia);
540
541     Aia_Func *func;
542     AIA_SECTION_FOR_EACH_FUNC(sec, func) {
543         AIA_FUNC_FOR_EACH_BLOCK_REVERSED(func, b)
544             x86_32_block_reg_alloc(b, aia);
545
546         Aia_Func_Trampoline *tramp;
547         AIA_FUNC_FOR_EACH_TRAMPOLINE(func, tramp)
548             x86_32_block_reg_alloc(tramp->block, aia);
549     }
550 }
551
552 static void x86_32_instr_add_live_op(Aia_Instr *in, Aia_Operand *op)
553 {
554     switch (aia_operand_get_type(op)) {
555     case AIA_OPERAND_REG:
556         live_set_insert_var(in->meta_data, aia_operand_reg_get_name(op));
557         break;
558
559     case AIA_OPERAND_LABEL:
560         break;
561
562     case AIA_OPERAND_LOCAL_REF:
563         break;
564
565     case AIA_OPERAND_ARG:
566         break;
567 }

```



```

568     case AIA_OPERAND_DISPLAY_REF:
569         assert(aia_operand_is_reg(
570             aia_operand_display_ref_get_display_reg(op)));
571         x86_32_instr_add_live_op(in,
572             aia_operand_display_ref_get_display_reg(op));
573         break;
574
575     case AIA_OPERAND_ADDR_REF:
576         if (aia_operand_addr_ref_get_base(op)) {
577             assert(aia_operand_is_reg(aia_operand_addr_ref_get_base(op)));
578             x86_32_instr_add_live_op(in,
579                 aia_operand_addr_ref_get_base(op));
580         }
581         if (aia_operand_addr_ref_get_index(op)) {
582             assert(aia_operand_is_reg(aia_operand_addr_ref_get_index(op)));
583             x86_32_instr_add_live_op(in,
584                 aia_operand_addr_ref_get_index(op));
585         }
586         break;
587
588     default:
589         fatal_error(S("unexpected instruction operand type. Aborting...\n"));
590 }
591
592 }
593
594 static inline void add_default_interferences(Aia_Instr *in)
595 {
596     switch (aia_instr_get_type(in)) {
597 #if 0
598     case AIA_CALL:
599         live_set_insert_var(in->meta_data, reg_ecx_str);
600         /* Fall through. */
601 #endif
602     case AIA_IDIV:
603         live_set_insert_var(in->meta_data, reg_eax_str);
604         live_set_insert_var(in->meta_data, reg_edx_str);
605
606     default:
607         break;
608     }
609 }
610
611 static void x86_32_instr_live_init(Aia_Instr *in)
612 {
613     in->meta_data = live_set_alloc();
614     Aia_Operand *op;
615     AIA_INSTR_FOR_EACH_SRC(in, op) {
616         if (aia_operand_is_dest(op))
617             x86_32_instr_add_live_op(in, op);
618     }
619
620     op = aia_instr_get_dest_op(in);
621     if (!op)
622         return;
623
624     switch (aia_operand_get_type(op)) {
625     case AIA_OPERAND_ADDR_REF:
626         /* Fall through. */
627     case AIA_OPERAND_DISPLAY_REF:
628         x86_32_instr_add_live_op(in, op);
629         break;
630
631     default:
632         break;
633     }
634
635     add_default_interferences(in);
636 }
637
638 static void x86_32_block_live_init(Aia_Block *b, void *data UNUSED)
639 {
640     Aia_Instr *in;
641     AIA_BLOCK_FOR_EACH_INSTRUCTION(b, in)

```

```

642     x86_32_instr_live_init(in);
643 }
644
645 static void x86_32_section_live_init(Aia_Section *sec)
646 {
647     Aia_Block *b;
648     AIA_SECTION_FOR_EACH_BLOCK_REVERSED(sec, b)
649         x86_32_block_live_init(b, NULL);
650
651     Aia_Func *func;
652     AIA_SECTION_FOR_EACH_FUNC(sec, func) {
653         AIA_FUNC_FOR_EACH_BLOCK_REVERSED(func, b)
654             x86_32_block_live_init(b, NULL);
655
656         Aia_Func_trampoline *tramp;
657         AIA_FUNC_FOR_EACH_TRAMPOLINE(func, tramp)
658             x86_32_block_live_init(tramp->block, NULL);
659     }
660 }
661
662 static void x86_32_block_live_destroy(Aia_Block *b, void *data UNUSED)
663 {
664     Aia_Instr *in;
665     AIA_BLOCK_FOR_EACH_INSTRUCTION(b, in)
666         live_set_destroy(in->meta_data);
667 }
668
669 static void x86_32_section_live_destroy(Aia_Section *sec)
670 {
671     Aia_Block *b;
672     AIA_SECTION_FOR_EACH_BLOCK_REVERSED(sec, b)
673         x86_32_block_live_destroy(b, NULL);
674
675     Aia_Func *func;
676     AIA_SECTION_FOR_EACH_FUNC(sec, func) {
677         AIA_FUNC_FOR_EACH_BLOCK_REVERSED(func, b)
678             x86_32_block_live_destroy(b, NULL);
679
680         Aia_Func_trampoline *tramp;
681         AIA_FUNC_FOR_EACH_TRAMPOLINE(func, tramp)
682             x86_32_block_live_destroy(tramp->block, NULL);
683     }
684 }
685
686 static inline void live_sets_init(Aia *aia)
687 {
688     Aia_Section *sec;
689     AIA_FOR_EACH_SECTION(aia, sec)
690         x86_32_section_live_init(sec);
691 }
692
693 static inline void live_sets_destroy(Aia *aia)
694 {
695     Aia_Section *sec;
696     AIA_FOR_EACH_SECTION(aia, sec)
697         x86_32_section_live_destroy(sec);
698 }
699
700 static inline void live_sets_build(Aia *aia)
701 {
702     Aia_Section *sec;
703     do {
704         aia->meta_data = NULL;
705         AIA_FOR_EACH_SECTION(aia, sec)
706             x86_32_section_reg_alloc(aia, sec);
707     } while (aia->meta_data);
708 }
709
710 static void x86_32_nop_mov_remove_block(Aia_Block *b, void *arg UNUSED)
711 {
712     VECTOR(removed_ins);
713
714     Aia_Instr *in;
715     AIA_BLOCK_FOR_EACH_INSTRUCTION(b, in) {

```

```

716     if (aia_instr_get_type(in) == AIA_MOV) {
717         Aia_Operand *src = aia_instr_get_src_op(in, 0);
718         Aia_Operand *dest = aia_instr_get_dest_op(in);
719         if (aia_operands_equal(src, dest))
720             vector_append(&removed_ins, in);
721     }
722 }
723
724 vector_for_each_destroy(&removed_ins,
725     (Vector_Destructor)aia_instr_remove_destroy);
726 }
727
728 static inline void x86_32_nop_mov_remove(Aia *aia)
729 {
730     Aia_Section *sec;
731     AIA_FOR_EACH_SECTION(aia, sec) {
732         aia_section_for_each_block_depth(sec,
733             (Aia_Block_Callback)x86_32_nop_mov_remove_block, NULL);
734     }
735     Aia_Func *func;
736     AIA_SECTION_FOR_EACH_FUNC(sec, func) {
737         aia_func_for_each_block_depth(func,
738             (Aia_Block_Callback)x86_32_nop_mov_remove_block, NULL);
739     }
740     Aia_Func_Trampoline *tramp;
741     AIA_FUNC_FOR_EACH_TRAMPOLINE(func, tramp)
742         x86_32_nop_mov_remove_block(tramp->block, NULL);
743 }
744 }
745 }
746
747 static Aia_Operand *x86_32_get_assigned_reg_operand(Aia_Operand *op,
748     Live_Graph *g, uint8_t reg_size)
749 {
750     Const_String op_name = aia_operand_reg_get_name(op);
751     Uns hash = string_hash_code(op_name);
752     DLOG("instr assign %S to reg\n", op_name);
753
754     Hash_Map_Slot *s = hash_map_get(&g->live_node_map, (String)op_name, hash);
755     if (!s)
756         return NULL;
757
758     Live_Node *n = LIVE_NODE_OF_GRAPH_SLOT(s);
759
760     assert(x86_32_is_concrete_reg_str(n->assigned_var_name));
761     Aia_Operand *reg = x86_32_get_reg_operand(n->assigned_var_name);
762     assert(reg);
763
764     DLOG("found assigned reg: %S\n", aia_operand_reg_get_name(reg));
765
766     Int reg_idx = x86_32_reg_get_idx(reg);
767     assert(reg_idx > -1);
768     if (reg_size == AIA_LONG) {
769         Const_String name = x86_32_reg_idx_get_32bit_name(reg_idx);
770         reg = x86_32_get_reg_operand(name);
771     } else {
772         Const_String name = x86_32_reg_idx_get_lo8bit_name(reg_idx);
773         reg = x86_32_get_reg_operand(name);
774     }
775
776     return reg;
777 }
778
779 static inline bool x86_32_instr_assign_reg(Aia_Instr *in,
780     Aia_Operand *op, Int idx, Live_Graph *g)
781 {
782     Aia_Operand *reg;
783     uint8_t reg_size;
784     if (idx == -1)
785         reg_size = aia_instr_get_dest_op_size(in);
786     else
787         reg_size = aia_instr_get_src_ops_size(in);
788
789     Aia_Block *b = aia_instr_get_block(in);

```

```

790     Aia *aia = b->section->aia;
791
792     switch (aia_operand_get_type(op)) {
793     case AIA_OPERAND_REG:
794         reg = x86_32_get_assigned_reg_operand(op, g, reg_size);
795         if (!reg)
796             return false;
797         aia_instr_replace_op(in, idx, reg);
798         break;
799
800     case AIA_OPERAND_DISPLAY_REF:
801         reg = aia_operand_display_ref_get_display_reg(op);
802         reg = x86_32_get_assigned_reg_operand(reg, g, AIA_LONG);
803         assert(reg);
804         op = aia_operand_display_ref_with_replaced_reg(aia, op, reg);
805         aia_instr_replace_op(in, idx, op);
806         break;
807
808     case AIA_OPERAND_ADDR_REF:
809         reg = aia_operand_addr_ref_get_base(op);
810         if (reg) {
811             reg = x86_32_get_assigned_reg_operand(reg, g, AIA_LONG);
812             assert(reg);
813             op = aia_operand_addr_ref_with_replaced_base(aia, op, reg);
814             aia_instr_replace_op(in, idx, op);
815         }
816         reg = aia_operand_addr_ref_get_index(op);
817         if (reg) {
818             reg = x86_32_get_assigned_reg_operand(reg, g, AIA_LONG);
819             assert(reg);
820             op = aia_operand_addr_ref_with_replaced_index(aia, op, reg);
821             aia_instr_replace_op(in, idx, op);
822         }
823         break;
824
825     default:
826         break;
827     }
828
829     return true;
830 }
831
832 static void x86_32_instr_assign_regs(Aia_Instr *in, Live_Graph *g)
833 {
834     Int idx = -1;
835     Aia_Operand *op;
836
837     AIA_INSTR_FOR_EACH_OPERAND(in, op) {
838         if (op && !x86_32_instr_assign_reg(in, op, idx, g)) {
839             Aia_Block *b = aia_instr_get_block(in);
840             if (!b->meta_data)
841                 b->meta_data = vector_alloc();
842             if (aia_instr_get_type(in) == AIA_CALL) {
843                 assert(idx == -1);
844                 __aia_operand_release(op);
845                 in->dest_op = NULL;
846             } else {
847                 vector_append(b->meta_data, in);
848                 break;
849             }
850         }
851         ++idx;
852     }
853 }
854
855 static void x86_32_block_assign_regs(Aia_Block *b, Live_Graph *g)
856 {
857     b->meta_data = NULL;
858     Aia_Instr *in;
859     AIA_BLOCK_FOR_EACH_INSTRUCTION_REVERSED(b, in)
860         x86_32_instr_assign_regs(in, g);
861
862     if (b->meta_data) {
863         Aia_Instr *in;

```

```

864     Vector *invec = b->meta_data;
865     VECTOR_FOR_EACH_ENTRY(invec, in) {
866         live_set_destroy(in->meta_data);
867         assert(aia_instr_get_type(in) != AIA_IDIV);
868         assert(aia_instr_get_type(in) != AIA_CALL);
869         aia_instr_remove_destroy(in);
870     }
871     vector_destroy(invec, NULL);
872 }
873 }
874
875 static void x86_32_section_assign_regs(Aia *aia, Aia_Section *sec)
876 {
877     Aia_Block *b;
878     AIA_SECTION_FOR_EACH_BLOCK_REVERSED(sec, b)
879         x86_32_block_assign_regs(b, aia->meta_data);
880
881     Aia_Func *func;
882     AIA_SECTION_FOR_EACH_FUNC(sec, func) {
883         AIA_FUNC_FOR_EACH_BLOCK_REVERSED(func, b)
884             x86_32_block_assign_regs(b, aia->meta_data);
885
886         Aia_Func_Trampoline *tramp;
887         AIA_FUNC_FOR_EACH_TRAMPOLINE(func, tramp)
888             x86_32_block_assign_regs(tramp->block, aia->meta_data);
889     }
890 }
891
892 static inline void x86_32_assign_regs(Aia *aia, Live_Graph *g)
893 {
894     Aia_Section *sec;
895     aia->meta_data = g;
896     AIA_FOR_EACH_SECTION(aia, sec)
897         x86_32_section_assign_regs(aia, sec);
898 }
899
900 typedef struct X86_32_Spill_Meta_Data {
901     Live_Graph *graph;
902     Aia_Section *curr_sec;
903     Vector tmp_var_names;
904     Vector inserted_tmp_var_labels;
905     Vector norm_instrs;
906     Aia_Operand_Map *repl_map;
907 } X86_32_Spill_Meta_Data;
908
909 static inline bool x86_32_reg_alloc_replace_reg(Aia_Instr *in, Int idx,
910     Aia_Operand *from, Aia_Operand *repl)
911 {
912     Aia_Operand *op = aia_instr_get_op(in, idx);
913     if (!aia_operands_equal(op, from))
914         return false;
915
916     assert(op);
917     DEBUG(
918         DLOGT(def, "replace: ");
919         aia_operand_dump(stderr, op, false);
920         DLOGT(def, " with: ");
921         aia_operand_dump(stderr, repl, false);
922         DLOGT(def, "\n");
923     );
924
925     aia_instr_replace_op(in, idx, repl);
926     return true;
927 }
928
929 static bool x86_32_reg_alloc_replace_addr_ref(Aia_Instr *in, Int idx,
930     Aia_Operand *from, Aia_Operand *repl)
931 {
932     Aia_Operand *op = aia_instr_get_op(in, idx);
933     assert(op);
934
935     bool ret = false;
936
937     Aia_Block *b = aia_instr_get_block(in);

```

```

938     Aia *aia = b->section->aia;
939
940     Aia_Operand *base = aia_operand_addr_ref_get_base(op);
941     if (base && aia_operands_equal(base, from)) {
942         op = aia_operand_addr_ref_with_replaced_base(aia, op, repl);
943         aia_instr_replace_op(in, idx, op);
944         ret = true;
945     }
946
947     Aia_Operand *index = aia_operand_addr_ref_get_index(op);
948     if (index && aia_operands_equal(index, from)) {
949         op = aia_operand_addr_ref_with_replaced_index(aia, op, repl);
950         aia_instr_replace_op(in, idx, op);
951         ret = true;
952     }
953
954     DEBUG(
955         DLOGT(def, "new addr ref: ");
956         aia_operand_dump(stderr, op, false);
957         DLOGT(def, "\n");
958     );
959
960     return ret;
961 }
962
963 static bool x86_32_reg_alloc_replace_display_ref(Aia_Instr *in, Int idx,
964     Aia_Operand *from, Aia_Operand *repl)
965 {
966     Aia_Operand *op = aia_instr_get_op(in, idx);
967     assert(op);
968
969     Aia_Block *b = aia_instr_get_block(in);
970     Aia *aia = b->section->aia;
971
972     Aia_Operand *reg = aia_operand_display_ref_get_display_reg(op);
973     assert(reg);
974
975     if (!aia_operands_equal(from, reg))
976         return false;
977
978     op = aia_operand_display_ref_with_replaced_reg(aia, op, repl);
979     aia_instr_replace_op(in, idx, op);
980
981     DEBUG(
982         DLOGT(def, "new display ref: ");
983         aia_operand_dump(stderr, op, false);
984         DLOGT(def, "\n");
985     );
986
987     return true;
988 }
989
990 static inline void x86_32_instr_spill(Aia_Instr *in, Aia_Operand *op,
991     Int in_idx, Int md_idx, Aia *aia)
992 {
993     X86_32_Spill_Meta_Data *md = aia->meta_data;
994
995     Live_Node *live_node = vector_get(&md->graph->spill_list, md_idx);
996     Const_String spill_var = live_node->init_var_name;
997     Const_String new_var_name = vector_get(&md->tmp_var_names, md_idx);
998
999     Aia_Operand *repl1 = NULL;
1000     Aia_Operand *repl2 = NULL;
1001     Const_String op_name;
1002     switch (aia_operand_get_type(op)) {
1003     case AIA_OPERAND_REG:
1004         repl1 = op;
1005         op_name = aia_operand_reg_get_name(op);
1006         if (string_compare(op_name, spill_var))
1007             return;
1008         assert(!x86_32_is_concrete_reg_str(op_name));
1009         break;
1010     case AIA_OPERAND_ADDR_REF:;

```

```

1012     repl2 = aia_operand_addr_ref_get_index(op);
1013     Const_String tn;
1014     if (repl2)
1015         tn = aia_operand_reg_get_name(repl2);
1016     else
1017         tn = S(" ");
1018
1019     repl1 = aia_operand_addr_ref_get_base(op);
1020     if (repl1)
1021         op_name = aia_operand_reg_get_name(repl1);
1022     else
1023         op_name = S("");
1024
1025     bool repl2_res = string_compare(tn, spill_var);
1026     bool repl1_res = string_compare(op_name, spill_var);
1027     if (repl1_res && repl2_res)
1028         return;
1029
1030     if (repl1_res)
1031         repl1 = NULL;
1032     if (repl2_res)
1033         repl2 = NULL;
1034
1035     assert(!x86_32_is_concrete_reg_str(tn));
1036     assert(!x86_32_is_concrete_reg_str(op_name));
1037     break;
1038
1039 case AIA_OPERAND_DISPLAY_REF:
1040     repl1 = aia_operand_display_ref_get_display_reg(op);
1041     assert(repl1);
1042     op_name = aia_operand_reg_get_name(repl1);
1043     if (string_compare(op_name, spill_var))
1044         return;
1045     assert(!x86_32_is_concrete_reg_str(op_name));
1046     break;
1047
1048 default:
1049     return;
1050 }
1051
1052 Aia_Operand *new_op;
1053 uint8_t size;
1054
1055 if (in_idx < 0)
1056     size = aia_instr_get_dest_op_size(in);
1057 else
1058     size = aia_instr_get_src_ops_size(in);
1059
1060 Aia_Operand_Map *repl_map = md->repl_map;
1061 if (repl1)
1062     new_op = aia_operand_map_get_value(repl_map, repl1);
1063 else
1064     new_op = aia_operand_map_get_value(repl_map, repl2);
1065
1066 if (!new_op) {
1067     if (___aia_get_curr_func(aia)) {
1068         DLOG("spill local\n");
1069         new_op = aia_operand_local_ref_alloc(aia, new_var_name
1070             /* , size */);
1071         if (!vector_get(&md->inserted_tmp_var_labels, md_idx)) {
1072             vector_set(&md->inserted_tmp_var_labels, md_idx,
1073                 INT_TO_PTR(1));
1074             ___aia_func_append_local(aia, new_var_name);
1075         }
1076     } else {
1077         if (!vector_get(&md->inserted_tmp_var_labels, md_idx)) {
1078             vector_set(&md->inserted_tmp_var_labels, md_idx,
1079                 INT_TO_PTR(1));
1080             DLOG("insert label: %S\n", new_var_name);
1081             ___aia_switch_section(aia, AIA_SECTION_DATA);
1082             ___aia_insert_label_instr(aia,
1083                 new_var_name,
1084                 0,
1085                 size == AIA_LONG ? 4 : 0,

```

```

1086             AIA_LINKAGE_PRIVATE,
1087             AIA_LABEL_TYPE_OBJ,
1088             size == AIA_LONG ? 4 : 1,
1089             aia_get_null_location(aia));
1090     Aia_Operand *val = aia_operand_const_int_alloc(aia, 0);
1091     __aia_insert_const_val_instr(aia, val, size,
1092     aia_get_null_location(aia));
1093 }
1094
1095     new_op = aia_operand_label_alloc(aia, new_var_name, 0);
1096 }
1097 }
1098
1099     switch (aia_operand_get_type(op)) {
1100     case AIA_OPERAND_REG:
1101         x86_32_reg_alloc_replace_reg(in, in_idx, repl1, new_op);
1102         break;
1103
1104     case AIA_OPERAND_ADDR_REF:
1105         if (repl1)
1106             x86_32_reg_alloc_replace_addr_ref(in, in_idx, repl1, new_op);
1107         if (repl2)
1108             x86_32_reg_alloc_replace_addr_ref(in, in_idx, repl2, new_op);
1109         break;
1110
1111     case AIA_OPERAND_DISPLAY_REF:
1112         x86_32_reg_alloc_replace_display_ref(in, in_idx, repl1, new_op);
1113         break;
1114
1115     default:
1116         assert(false);
1117     }
1118     vector_append(&md->norm_instrs, in);
1119 }
1120
1121 static void x86_32_instr_spill_regs(Aia_Instr *in, Aia *aia)
1122 {
1123     Int idx = -1;
1124     Aia_Operand *op;
1125
1126     X86_32_Spill_Meta_Data *md = aia->meta_data;
1127
1128     AIA_INSTR_FOR_EACH_OPERAND(in, op) {
1129         if (op) {
1130             __aia_operand_acquire(op);
1131             for (Uns i = 0; i < vector_size(&md->graph->spill_list); i++)
1132                 x86_32_instr_spill(in, op, idx, i, aia);
1133             __aia_operand_release(op);
1134         }
1135         ++idx;
1136     }
1137 }
1138
1139 static void x86_32_block_spill_regs(Aia_Block *b, Aia *aia)
1140 {
1141     X86_32_Spill_Meta_Data *md = aia->meta_data;
1142     Aia_Instr *in;
1143     AIA_BLOCK_FOR_EACH_INSTRUCTION_REVERSED(b, in) {
1144         x86_32_instr_spill_regs(in, aia);
1145         //aia_normalize_addr_instructions(aia, &md->norm_instrs);
1146         //x86_32_normalize_instructions(aia, &md->norm_instrs);
1147         vector_clear(&md->norm_instrs);
1148     }
1149 }
1150
1151 static void x86_32_section_spill_regs(Aia *aia, Aia_Section *sec)
1152 {
1153     X86_32_Spill_Meta_Data *md = aia->meta_data;
1154     md->curr_sec = sec;
1155     __aia_set_curr_func(aia, NULL);
1156
1157     Aia_Block *b;
1158     AIA_SECTION_FOR_EACH_BLOCK_REVERSED(sec, b)
1159         x86_32_block_spill_regs(b, aia);

```



```

1160
1161     Aia_Func *func;
1162     AIA_SECTION_FOR_EACH_FUNC(sec, func) {
1163         __aia_set_curr_func(aia, func);
1164         AIA_FUNC_FOR_EACH_BLOCK_REVERSED(func, b)
1165             x86_32_block_spill_regs(b, aia);
1166     }
1167 }
1168
1169 static inline void x86_32_spill_regs(Aia *aia, Live_Graph *g,
1170     Aia_Operand_Map *repl_map)
1171 {
1172     aia->meta_data = ALLOC_NEW(X86_32_Spill_Meta_Data);
1173     X86_32_Spill_Meta_Data *md = aia->meta_data;
1174     md->graph = g;
1175     md->tmp_var_names = VECTOR_INIT();
1176     md->inserted_tmp_var_labels = VECTOR_INIT();
1177     md->norm_instrs = VECTOR_INIT();
1178     md->repl_map = repl_map;
1179
1180     Live_Node *n;
1181     VECTOR_FOR_EACH_ENTRY(&g->spill_list, n) {
1182         vector_append(&md->tmp_var_names, aia_tmp_name_gen(aia,
1183             n->init_var_name));
1184         vector_append(&md->inserted_tmp_var_labels, NULL);
1185     }
1186
1187     Aia_Section *sec;
1188     AIA_FOR_EACH_SECTION(aia, sec)
1189         x86_32_section_spill_regs(aia, sec);
1190
1191     //aia_normalize_addr_instructions(aia, &md->norm_instrs);
1192     //x86_32_normalize_instructions(aia, &md->norm_instrs);
1193
1194     assert(vector_is_empty(&md->norm_instrs));
1195     vector_clear(&md->inserted_tmp_var_labels);
1196     vector_for_each_destroy(&md->tmp_var_names,
1197         (Vector_Destructor)string_destroy);
1198     free_mem(aia->meta_data);
1199
1200     __aia_set_curr_func(aia, NULL);
1201 }
1202
1203 static void __x86_32_instr_force_interferences(Aia_Instr *in, Live_Graph *g,
1204     Live_Node *interfear_node)
1205 {
1206     Live_Set *live_set;
1207     Live_Node *oth;
1208     Const_String var;
1209
1210     live_set = in->meta_data;
1211     LIVE_SET_FOR_EACH_VAR(live_set, var) {
1212         oth = live_graph_lookup_var(g, var);
1213         assert(oth);
1214         DLOG("interference: %S and %S\n",
1215             interfear_node->init_var_name, oth->init_var_name);
1216         live_node_add_interference(interfear_node, oth);
1217     }
1218 }
1219
1220 static void x86_32_instr_add_interferences(Aia_Instr *in, Live_Graph *g,
1221     Live_Node *interfear_node)
1222 {
1223     if (!live_set_contains(in->meta_data, interfear_node->init_var_name))
1224         return;
1225
1226     __x86_32_instr_force_interferences(in, g, interfear_node);
1227 }
1228
1229 static void live_graph_build_queues(Live_Graph *g)
1230 {
1231     Hash_Map_Slot *slot;
1232     HASH_MAP_FOR_EACH(&g->live_node_map, slot) {
1233         Live_Node *n = LIVE_NODE_OF_GRAPH_SLOT(slot);

```

```

1234     if (x86_32_is_concrete_reg_str(n->init_var_name))
1235         continue;
1236     if (n->var_size == AIA_LONG) {
1237         if (hash_map_size(&n->interference_map) < X86_32_REG_COUNT)
1238             double_list_append(&g->assign_queue, &n->dbnode);
1239         else
1240             double_list_append(&g->wait_queue, &n->dbnode);
1241     } else {
1242         if (hash_map_size(&n->interference_map) < X86_32_REG_COUNT_8BIT)
1243             double_list_append(&g->assign_queue, &n->dbnode);
1244         else
1245             double_list_append(&g->wait_queue, &n->dbnode);
1246     }
1247 }
1248 }
1249
1250 static void live_graph_push_node(Live_Graph *g, Live_Node *n)
1251 {
1252     Live_Node *inode;
1253
1254     assert(!x86_32_is_concrete_reg_str(n->init_var_name));
1255
1256     DLOG("iterate: %S\n", n->init_var_name);
1257
1258     LIVE_NODE_FOR_EACH_INTERFERENCE(n, inode) {
1259
1260         if (x86_32_is_concrete_reg_str(inode->init_var_name))
1261             continue;
1262
1263         DLOG("test %S interferes with %S\n", inode->init_var_name,
1264             n->init_var_name);
1265
1266         assert(live_node_has_interference(inode, n));
1267         DLOG("remove interference from %S to %S\n", inode->init_var_name,
1268             n->init_var_name);
1269         live_node_remove_interference(inode, n);
1270
1271         if (inode->var_size == AIA_LONG) {
1272             if (hash_map_size(&inode->interference_map) ==
1273                 X86_32_REG_COUNT - 1) {
1274                 double_list_remove(&inode->dbnode);
1275                 double_list_append(&g->assign_queue, &inode->dbnode);
1276             }
1277         } else {
1278             assert(inode->var_size == AIA_BYTE);
1279             if (hash_map_size(&inode->interference_map) ==
1280                 X86_32_REG_COUNT_8BIT - 1) {
1281                 double_list_remove(&inode->dbnode);
1282                 double_list_append(&g->assign_queue, &inode->dbnode);
1283             }
1284         }
1285     }
1286
1287     VECTOR_FOR_EACH_ENTRY(&n->interferences, inode) {
1288         if (x86_32_is_concrete_reg_str(inode->init_var_name))
1289             continue;
1290         live_node_remove_interference(n, inode);
1291     }
1292
1293     vector_append(&g->live_node_stack, n);
1294 }
1295
1296 static bool live_graph_build_stack(Live_Graph *g)
1297 {
1298     ALWAYS_INLINE Live_Node *get_spill()
1299     {
1300         Live_Node *spill;
1301         Double_List_Node *node;
1302         Double_List *list = &g->wait_queue;
1303
1304         node = double_list_peek_first(list);
1305         spill = LIVE_NODE_OF_DBNODE(node);
1306         Uns spill_size = hash_map_size(&spill->interference_map);
1307

```

```

1308     DOUBLE_LIST_FOR_EACH_AFTER(list, node, node) {
1309         Live_Node *curr = LIVE_NODE_OF_DBNODE(node);
1310         Uns curr_size = hash_map_size(&curr->interference_map);
1311         if (curr_size > spill_size) {
1312             spill = curr;
1313             spill_size = curr_size;
1314         }
1315     }
1316
1317     return spill;
1318 }
1319
1320 Double_List_Node *dbnode;
1321 Live_Node *n;
1322 bool ret = true;
1323
1324 for (;;) {
1325     while (!double_list_is_empty(&g->assign_queue)) {
1326         dbnode = double_list_pop_first(&g->assign_queue);
1327         live_graph_push_node(g, LIVE_NODE_OF_DBNODE(dbnode));
1328     }
1329
1330     if (double_list_is_empty(&g->wait_queue))
1331         break;
1332
1333     ret = false;
1334
1335     n = get_spill();
1336     double_list_remove(&n->dbnode);
1337     live_graph_push_node(g, n);
1338     vector_append(&g->spill_list, n);
1339 }
1340
1341 return ret;
1342 }
1343
1344 static void live_node_assign_reg(Live_Node *n, Live_Graph *g)
1345 {
1346     assert(n->reg_idx < 0);
1347
1348     Live_Node *inter;
1349     VECTOR_FOR_EACH_ENTRY(&n->interferences, inter) {
1350         if (hash_map_contains(&g->regraph_map, inter,
1351             hash_map_aligned_ptr_hash(inter))) {
1352             bool link_ret = live_node_link(n, inter);
1353             (void)link_ret;
1354             assert(link_ret);
1355         }
1356     }
1357
1358     LIVE_NODE_FOR_EACH_INTERFERENCE(n, inter) {
1359         assert(inter->reg_idx >= 0);
1360         DLOG("found interference: %S\n",
1361             x86_32_reg_idx_get_32bit_name(inter->reg_idx));
1362         g->free_regs[inter->reg_idx] = false;
1363     }
1364
1365     hash_map_insert(&g->regraph_map, &n->regraph_slot,
1366         hash_map_aligned_ptr_hash(n));
1367
1368     for (Uns i = 0; i < X86_32_REG_COUNT; i++) {
1369         if (g->free_regs[i]) {
1370             live_node_set_assigned_var_name(n, i);
1371             break;
1372         }
1373     }
1374
1375     for (Uns i = 0; i < X86_32_REG_COUNT; i++)
1376         g->free_regs[i] = true;
1377
1378     assert(n->reg_idx >= 0);
1379 }
1380
1381 static void live_graph_assign_regs(Live_Graph *g)

```

```

1382 {
1383     Vector *vec = &g->live_node_stack;
1384
1385     while (!vector_is_empty(vec)) {
1386         Live_Node *next = vector_pop_last(vec);
1387         live_node_assign_reg(next, g);
1388     }
1389
1390     hash_map_clear(&g->regraph_map);
1391 }
1392
1393 static void __x86_32_instr_add_interferences(Aia_Instr *in,
1394     Aia_Instr *suc, Live_Graph *g)
1395 {
1396     Live_Node *n;
1397     switch (aia_instr_get_type(in)) {
1398     case AIA_IDIV:
1399         n = live_graph_lookup(g, reg_eax);
1400         assert(n);
1401         x86_32_instr_add_interferences(suc, g, n);
1402         n = live_graph_lookup(g, reg_edx);
1403         assert(n);
1404         x86_32_instr_add_interferences(suc, g, n);
1405         return;
1406
1407     default:
1408         break;
1409     }
1410
1411     Aia_Operand *dest = aia_instr_get_dest_op(in);
1412     if (!dest)
1413         return;
1414
1415     switch (aia_operand_get_type(dest)) {
1416     case AIA_OPERAND_REG:
1417         n = live_graph_lookup(g, dest);
1418         assert(n);
1419         x86_32_instr_add_interferences(suc, g, n);
1420         break;
1421
1422     case AIA_OPERAND_DISPLAY_REF:
1423         n = live_graph_lookup(g,
1424             aia_operand_display_ref_get_display_reg(dest));
1425         assert(n);
1426         x86_32_instr_add_interferences(suc, g, n);
1427         break;
1428
1429     case AIA_OPERAND_ADDR_REF:
1430         Aia_Operand *tmp = aia_operand_addr_ref_get_base(dest);
1431         if (tmp) {
1432             n = live_graph_lookup(g, tmp);
1433             assert(n);
1434             x86_32_instr_add_interferences(suc, g, n);
1435         }
1436         tmp = aia_operand_addr_ref_get_index(dest);
1437         if (tmp) {
1438             n = live_graph_lookup(g, tmp);
1439             assert(n);
1440             x86_32_instr_add_interferences(suc, g, n);
1441         }
1442         break;
1443
1444     case AIA_OPERAND_LABEL:
1445         /* Fall through. */
1446     case AIA_OPERAND_LOCAL_REF:
1447         /* Fall through. */
1448     case AIA_OPERAND_ARG:
1449         /* Fall through. */
1450         break;
1451
1452     default:
1453         fatal_error(S("unexpected instruction dest-operand type. "
1454             "Aborting...\n"));
1455     }

```

```

1456 }
1457
1458 static void x86_32_instr_live_graph_build(Aia_Instr *in, Live_Graph *g)
1459 {
1460     Aia_Block *b = aia_instr_get_block(in);
1461     Aia_Instr *suc = aia_instr_get_sucessor(in);
1462     if (!suc) {
1463         Aia_Block *suc_block;
1464         AIA_BLOCK_FOR_EACH_SUCESSOR(b, suc_block) {
1465             __x86_32_instr_add_interferences(in,
1466             __aia_block_peek_first_instr(suc_block), g);
1467         }
1468     } else {
1469         __x86_32_instr_add_interferences(in, suc, g);
1470     }
1471
1472 #if 0
1473     void add_default_8bit_interferences(Aia_Operand *src)
1474     {
1475         DEBUG(
1476             if (aia_operand_is_reg(src))
1477                 DLOG("try add default interference to: %S\n",
1478                     aia_operand_reg_get_name(src));
1479             );
1480
1481         if (aia_operand_is_reg(src) &&
1482             aia_instr_get_src_ops_size(in) == AIA_BYTE) {
1483             DLOG("DO add default interference to: %S\n",
1484                 aia_operand_reg_get_name(src));
1485             Live_Node *live_node = live_graph_lookup(g, src);
1486             assert(live_node);
1487             live_node_default_8bit_interferences(live_node, g);
1488         }
1489     }
1490
1491     Aia_Operand *src;
1492     AIA_INSTR_FOR_EACH_SRC(in, src)
1493         add_default_8bit_interferences(src);
1494 #endif
1495 }
1496
1497 static void x86_32_block_live_graph_build(Aia_Block *b, Live_Graph *g)
1498 {
1499     Aia_Instr *in;
1500     AIA_BLOCK_FOR_EACH_INSTRUCTION(b, in)
1501         x86_32_instr_live_graph_build(in, g);
1502 }
1503
1504 static void x86_32_section_live_graph_build(Aia_Section *sec, Live_Graph *g)
1505 {
1506     Aia_Block *b;
1507     AIA_SECTION_FOR_EACH_BLOCK_REVERSED(sec, b)
1508         x86_32_block_live_graph_build(b, g);
1509
1510     Aia_Func *func;
1511     AIA_SECTION_FOR_EACH_FUNC(sec, func) {
1512         AIA_FUNC_FOR_EACH_BLOCK_REVERSED(func, b)
1513             x86_32_block_live_graph_build(b, g);
1514
1515         Aia_Func_trampoline *tramp;
1516         AIA_FUNC_FOR_EACH_TRAMPOLINE(func, tramp)
1517             x86_32_block_live_graph_build(tramp->block, g);
1518     }
1519 }
1520
1521 static inline void live_graph_build(Aia *aia, Live_Graph *g)
1522 {
1523     Aia_Section *sec;
1524     AIA_FOR_EACH_SECTION(aia, sec)
1525         x86_32_section_live_graph_build(sec, g);
1526 }
1527
1528 static void live_graph_insert(Live_Graph *g, Aia_Operand *op, uint8_t var_size)
1529 {

```

```

1530     if (!op)
1531         return;
1532
1533     switch (aia_operand_get_type(op)) {
1534     case AIA_OPERAND_REG:
1535         break;
1536     case AIA_OPERAND_DISPLAY_REF:
1537         live_graph_insert(g,
1538             aia_operand_display_ref_get_display_reg(op), var_size);
1539         return;
1540     case AIA_OPERAND_ADDR_REF:
1541         live_graph_insert(g, aia_operand_addr_ref_get_base(op), var_size);
1542         live_graph_insert(g, aia_operand_addr_ref_get_index(op), var_size);
1543         return;
1544     default:
1545         return;
1546     }
1547
1548     Live_Node *n = __live_graph_get(g, op, var_size);
1549     if (var_size != AIA_LONG) {
1550         assert(var_size == AIA_BYTE);
1551         n->var_size = var_size;
1552     }
1553 }
1554
1555 static void live_graph_block_init(Aia_Block *b, Live_Graph *g, Aia *aia UNUSED)
1556 {
1557     Aia_Operand *op;
1558
1559     Aia_Instr *in;
1560     AIA_BLOCK_FOR_EACH_INSTRUCTION(b, in) {
1561         AIA_INSTR_FOR_EACH_SRC(in, op)
1562             live_graph_insert(g, op, aia_instr_get_src_ops_size(in));
1563
1564         op = aia_instr_get_dest_op(in);
1565         live_graph_insert(g, op, aia_instr_get_dest_op_size(in));
1566     }
1567 }
1568
1569 static void live_graph_insert_x86_32_regs(Live_Graph *g)
1570 {
1571     live_graph_insert(g, reg_eax, AIA_LONG);
1572     live_graph_insert(g, reg_edx, AIA_LONG);
1573     live_graph_insert(g, reg_ecx, AIA_LONG);
1574     live_graph_insert(g, reg_ebx, AIA_LONG);
1575     live_graph_insert(g, reg_esi, AIA_LONG);
1576     live_graph_insert(g, reg_edi, AIA_LONG);
1577     live_graph_insert(g, reg_ebp, AIA_LONG);
1578     live_graph_insert(g, reg_esp, AIA_LONG);
1579     live_graph_insert(g, reg_al, AIA_BYTE);
1580     live_graph_insert(g, reg_dl, AIA_BYTE);
1581     live_graph_insert(g, reg_cl, AIA_BYTE);
1582     live_graph_insert(g, reg_bl, AIA_BYTE);
1583     live_graph_insert(g, reg_ah, AIA_BYTE);
1584     live_graph_insert(g, reg_dh, AIA_BYTE);
1585     live_graph_insert(g, reg_ch, AIA_BYTE);
1586     live_graph_insert(g, reg_bh, AIA_BYTE);
1587 }
1588
1589 static void live_graph_init(Aia *aia, Live_Graph *g)
1590 {
1591     live_graph_insert_x86_32_regs(g);
1592
1593     Aia_Section *sec;
1594     AIA_FOR_EACH_SECTION(aia, sec) {
1595         Aia_Block *b;
1596         AIA_SECTION_FOR_EACH_BLOCK_REVERSED(sec, b)
1597             live_graph_block_init(b, g, aia);
1598     }
1599
1600     Aia_Func *func;
1601     AIA_SECTION_FOR_EACH_FUNC(sec, func) {
1602         AIA_FUNC_FOR_EACH_BLOCK_REVERSED(func, b)
1603             live_graph_block_init(b, g, aia);

```

```

1604
1605     Aia_Func_Trampoline *tramp;
1606     AIA_FUNC_FOR_EACH_TRAMPOLINE(func, tramp)
1607         live_graph_block_init(tramp->block, g, aia);
1608 }
1609 }
1610 }
1611
1612 static void live_graph_instr_call_add_interference(Aia_Instr *in,
1613 Live_Graph *g)
1614 {
1615     if (aia_instr_get_type(in) != AIA_CALL)
1616         return;
1617
1618     DLOG("add call interferences\n");
1619
1620     Live_Set *vars = in->meta_data;
1621     Live_Set_Entry *e;
1622     LIVE_SET_FOR_EACH_ENTRY(vars, e) {
1623         __x86_32_instr_force_interferences(in, g,
1624         live_graph_lookup(g, reg_cl));
1625         __x86_32_instr_force_interferences(in, g,
1626         live_graph_lookup(g, reg_al));
1627         __x86_32_instr_force_interferences(in, g,
1628         live_graph_lookup(g, reg_dl));
1629     }
1630 }
1631
1632 static void live_graph_block_default_interferences(Aia_Block *b,
1633 Live_Graph *g)
1634 {
1635     Aia_Instr *in;
1636     AIA_BLOCK_FOR_EACH_INSTRUCTION(b, in)
1637         live_graph_instr_call_add_interference(in, g);
1638 }
1639
1640 static inline void live_graph_default_interferences(Aia *aia, Live_Graph *g)
1641 {
1642     Aia_Block *b;
1643     Aia_Section *sec;
1644     AIA_FOR_EACH_SECTION(aia, sec) {
1645         AIA_SECTION_FOR_EACH_BLOCK_REVERSED(sec, b)
1646             live_graph_block_default_interferences(b, g);
1647
1648         Aia_Func *func;
1649         AIA_SECTION_FOR_EACH_FUNC(sec, func) {
1650             AIA_FUNC_FOR_EACH_BLOCK_REVERSED(func, b)
1651                 live_graph_block_default_interferences(b, g);
1652
1653             Aia_Func_Trampoline *tramp;
1654             AIA_FUNC_FOR_EACH_TRAMPOLINE(func, tramp)
1655                 live_graph_block_default_interferences(tramp->block, g);
1656         }
1657     }
1658 }
1659
1660 static void x86_32_reg_alloc_do_replace(Aia_Instr *in,
1661 Aia_Operand *from, Aia_Operand *to)
1662 {
1663     if (!aia_operand_is_reg(from) || !aia_operand_is_reg(to))
1664         return;
1665     if (x86_32_is_concrete_reg(from))
1666         return;
1667     if (aia_operands_equal(from, to))
1668         return;
1669
1670     __aia_operand_acquire(from);
1671
1672     DEBUG(
1673         DLOGT(def, "try to replace ");
1674         aia_operand_dump(stderr, from, false);
1675         DLOGT(def, " with ");
1676         aia_operand_dump(stderr, to, false);
1677         DLOGT(def, "\n");

```

```

1678 );
1679
1680 Aia_Instr *suc = aia_instr_get_sucessor(in);
1681 for (; suc; suc = aia_instr_get_sucessor(suc)) {
1682     if (!live_set_contains_op(suc->meta_data, from))
1683         break;
1684
1685     if (!aia_instr_is_concrete(suc))
1686         continue;
1687     if (aia_instr_get_type(suc) == AIA_CALL) {
1688         if (x86_32_is_concrete_reg(to) &&
1689             !x86_32_is_callee_save_reg(to))
1690             goto out;
1691     }
1692     Aia_Operand *dest = aia_instr_get_dest_op(suc);
1693     if (dest) {
1694
1695         if (aia_operands_equal(dest, from) &&
1696             live_set_contains_op(suc->meta_data, to))
1697             goto out;
1698
1699         if (aia_operands_equal(dest, to)) {
1700             switch (aia_instr_get_type(suc)) {
1701                 case AIA_MOV:;
1702                 Aia_Operand *src = aia_instr_get_src_op(suc, 0);
1703                 if (aia_operands_equal(from, src) ||
1704                     aia_operands_equal(to, src))
1705                     continue;
1706
1707                 break;
1708
1709                 default:
1710                     break;
1711             }
1712             goto out;
1713         }
1714     }
1715 }
1716
1717 if (!suc)
1718     goto out;
1719
1720 suc = in;
1721 do {
1722     Int idx = -1;
1723     Aia_Operand *op;
1724     bool did_replace = false;
1725     AIA_INSTR_FOR_EACH_OPERAND(suc, op) {
1726         if (op) {
1727             switch (aia_operand_get_type(op)) {
1728                 case AIA_OPERAND_REG:
1729                     did_replace |= x86_32_reg_alloc_replace_reg(suc, idx,
1730                         from, to);
1731                     break;
1732                 case AIA_OPERAND_ADDR_REF:
1733                     did_replace |= x86_32_reg_alloc_replace_addr_ref(suc,
1734                         idx, from, to);
1735                     break;
1736                 case AIA_OPERAND_DISPLAY_REF:
1737                     did_replace |= x86_32_reg_alloc_replace_display_ref(suc,
1738                         idx, from, to);
1739                     break;
1740
1741                 default:
1742                     break;
1743             }
1744         }
1745         ++idx;
1746     }
1747
1748     if (live_set_remove_op(suc->meta_data, from))
1749         live_set_insert_op(suc->meta_data, to);
1750
1751     suc = aia_instr_get_sucessor(suc);

```



```

1752     } while (live_set_contains_op(suc->meta_data, from));
1753
1754 out:
1755     __aia_operand_release(from);
1756 }
1757
1758 static void x86_32_reg_alloc_replace_regs_block(Aia_Block *b)
1759 {
1760     Aia_Instr *in;
1761     Aia_Operand *src;
1762     Aia_Operand *dest;
1763     AIA_BLOCK_FOR_EACH_INSTRUCTION(b, in) {
1764         switch (aia_instr_get_type(in)) {
1765             case AIA_MOV:
1766                 src = aia_instr_get_src_op(in, 0);
1767                 dest = aia_instr_get_dest_op(in);
1768                 x86_32_reg_alloc_do_replace(in, dest, src);
1769                 break;
1770             default:
1771                 break;
1772         }
1773     }
1774 }
1775
1776 static void x86_32_reg_alloc_replace_regs(Aia *aia)
1777 {
1778     Aia_Block *b;
1779     Aia_Section *sec;
1780     AIA_FOR_EACH_SECTION(aia, sec) {
1781         AIA_SECTION_FOR_EACH_BLOCK_REVERSED(sec, b)
1782             x86_32_reg_alloc_replace_regs_block(b);
1783     }
1784
1785     Aia_Func *func;
1786     AIA_SECTION_FOR_EACH_FUNC(sec, func) {
1787         AIA_FUNC_FOR_EACH_BLOCK_REVERSED(func, b)
1788             x86_32_reg_alloc_replace_regs_block(b);
1789     }
1790
1791     Aia_Func_trampoline *tramp;
1792     AIA_FUNC_FOR_EACH_TRAMPOLINE(func, tramp)
1793         x86_32_reg_alloc_replace_regs_block(tramp->block);
1794 }
1795
1796 static void x86_32_instr_live_set_dump(FILE *stream, Aia_Instr *in)
1797 {
1798     String_Builder sb = STRING_BUILDER_INIT();
1799     string_builder_assign(&sb, S(" # live { "));
1800     Const_String v;
1801     Live_Set *s = in->meta_data;
1802     LIVE_SET_FOR_EACH_VAR(s, v) {
1803         string_builder_append(&sb, v);
1804         string_builder_append_char(&sb, ' ');
1805     }
1806     string_builder_append_char(&sb, '}');
1807     file_print_message(stream, string_builder_const_str(&sb));
1808     string_builder_clear(&sb);
1809 }
1810
1811 static void x86_32_reg_liveness_dump(Aia *aia, Const_String postfix)
1812 {
1813     aia_set_instr_dump_callback(aia, x86_32_instr_live_set_dump);
1814
1815     String fname = string_from_format(S("%S.vitaly.%S"),
1816         aia_get_file_name(aia), postfix);
1817     FILE *f = file_open(fname, S("w"));
1818     if (!f)
1819         fatal_error(S("unable to open file %S for intermediate "
1820             "code dump [%m]\n"), fname);
1821     string_destroy(fname);
1822     aia_dump(aia, f);
1823     file_close(f);
1824 }
1825

```

```

1826     aia_clear_instr_dump_callback(aia);
1827 }
1828
1829 static void x86_32_reg_alloc_dump(Aia *aia)
1830 {
1831     String fname = string_from_format(S("%S.vitaly.reg-alloc-x86-32-ic"),
1832     aia_get_file_name(aia));
1833     FILE *f = file_open(fname, S("w"));
1834     if (!f)
1835         fatal_error(S("unable to open file %S for intermediate code dump\n"),
1836         fname);
1837     string_destroy(fname);
1838     aia_dump(aia, f);
1839     file_close(f);
1840 }
1841
1842 void x86_32_reg_alloc_color(Aia *aia, Aia_Operand_Map *repl_map)
1843 {
1844     bool init_liveness_dumped = false;
1845
1846     DEBUG(
1847         Aia_Operand_Map_Entry *e;
1848         AIA_OPERAND_MAP_FOR_EACH_ENTRY(repl_map, e) {
1849             aia_operand_dump(stderr, e->operand, false);
1850             DLOGT(def, " -> ");
1851             aia_operand_dump(stderr, e->value, false);
1852             DLOGT(def, "\n");
1853         }
1854     );
1855
1856     DLOG("reg alloc file: %S\n", aia_get_file_name(aia));
1857     if (cmdopts.opt_unused_mov) {
1858         live_sets_init(aia);
1859         live_sets_build(aia);
1860
1861         if (cmdopts.dump_init_liveness_x86_32_ic) {
1862             x86_32_reg_liveness_dump(aia, S("init-liveness-x86-32-ic"));
1863             init_liveness_dumped = true;
1864         }
1865
1866         x86_32_reg_alloc_replace_regs(aia);
1867
1868         if (cmdopts.dump_unused_mov_ic)
1869             x86_32_reg_liveness_dump(aia, S("unused-mov-ic"));
1870
1871         live_sets_destroy(aia);
1872     }
1873
1874     x86_32_nop_mov_remove(aia);
1875
1876     for (;;) {
1877         live_sets_init(aia);
1878         live_sets_build(aia);
1879
1880         if (cmdopts.dump_init_liveness_x86_32_ic && !init_liveness_dumped) {
1881             x86_32_reg_liveness_dump(aia, S("init-liveness-x86-32-ic"));
1882             init_liveness_dumped = true;
1883         }
1884     }
1885
1886     Live_Graph *g = live_graph_alloc();
1887     live_graph_init(aia, g);
1888     live_graph_build(aia, g);
1889     live_graph_default_interferences(aia, g);
1890
1891     live_graph_build_queues(g);
1892
1893     if (live_graph_build_stack(g)) {
1894         if (cmdopts.dump_liveness_x86_32_ic)
1895             x86_32_reg_liveness_dump(aia, S("liveness-x86-32-ic"));
1896
1897         live_graph_assign_regs(g);
1898         x86_32_assign_regs(aia, g);
1899     }

```

```

1900         if (cmdopts.dump_reg_alloc_x86_32_ic)
1901             x86_32_reg_alloc_dump(aia);
1902
1903         live_graph_destroy(g);
1904         live_sets_destroy(aia);
1905         break;
1906
1907     }
1908     x86_32_spill_regs(aia, g, repl_map);
1909
1910     live_graph_destroy(g);
1911     live_sets_destroy(aia);
1912
1913     x86_32_nop_mov_remove(aia);
1914
1915     X86_32_Spill_Meta_Data *md = aia->meta_data;
1916     x86_32_normalize(aia);
1917     aia->meta_data = md;
1918
1919     aia_normalize_addr(aia);
1920 }
1921
1922 x86_32_nop_mov_remove(aia);
1923 }

```

:

A.8.23 src/x86_32/x86_32_reg_alloc_color.h

```

1  #ifndef X86_32_REG_ALLOC_COLOR_H
2  #define X86_32_REG_ALLOC_COLOR_H
3
4  #include <aia/aia_operand_map.h>
5
6  void x86_32_reg_alloc_color(Aia *aia, Aia_Operand_Map *repl_map);
7
8  #endif // X86_32_REG_ALLOC_COLOR_H

```

:

A.8.24 src/x86_32/x86_32_reg_vars.c

```

1  #include "x86_32_reg_vars.h"
2  #include <aia/aia_func_access.h>
3  #include <aia/aia_func_kills.h>
4  #include <aia/aia_operand_set.h>
5  #include <aia/aia_operand_map.h>
6  #include <main.h>
7
8  typedef struct Instr_Meta {
9      Aia_Operand_Set use_set;
10     Aia_Operand_Set def_set;
11     bool is_inserted;
12 } Instr_Meta;
13
14 typedef struct Aia_Meta {
15     Aia_Operand_Map subst_map;
16     Aia_Operand_Map *repl_map;
17 } Aia_Meta;
18
19 static inline bool x86_32_reg_vars_operand_significant(Aia_Operand *op)
20 {
21     switch (aia_operand_get_type(op)) {
22     case AIA_OPERAND_LABEL:
23         /* Fall through. */
24     case AIA_OPERAND_LOCAL_REF:
25         return true;

```

```

26     default:
27         return false;
28     }
29 }
30
31 static void x86_32_reg_vars_init_instr(Aia_Instr *in, bool is_inserted)
32 {
33     Instr_Meta *imeta = ALLOC_NEW(Instr_Meta);
34     imeta->use_set = AIA_OPERAND_SET_INIT();
35     imeta->def_set = AIA_OPERAND_SET_INIT();
36     imeta->is_inserted = is_inserted;
37     in->meta_data = imeta;
38 }
39
40 static void x86_32_reg_vars_init_block(Aia_Block *b)
41 {
42     Aia_Instr *in;
43     AIA_BLOCK_FOR_EACH_INSTRUCTION(b, in)
44         x86_32_reg_vars_init_instr(in, false);
45 }
46
47 static void x86_32_reg_vars_ins_glob_uses_defs(Aia_Block *b,
48     Aia_Operand_Set *glob_use_set,
49     Aia_Operand_Set *all_defs,
50     Aia_Operand_Set *all_uses,
51     Aia *aia UNUSED)
52 {
53     Aia_Instr *in;
54     AIA_BLOCK_FOR_EACH_INSTRUCTION(b, in) {
55
56         switch (aia_instr_get_type(in)) {
57             AIA_CASE_INCONCRETE:
58                 continue;
59             case AIA_CALL:
60                 continue;
61             default:
62                 break;
63         }
64
65         Aia_Operand *op = aia_instr_get_dest_op(in);
66         if (op) {
67             if (aia_operand_get_type(op) == AIA_OPERAND_LABEL)
68                 aia_operand_set_insert(glob_use_set, op);
69             if (x86_32_reg_vars_operand_significant(op))
70                 aia_operand_set_insert(all_defs, op);
71         }
72
73         AIA_INSTR_FOR_EACH_SRC(in, op) {
74             if (x86_32_reg_vars_operand_significant(op))
75                 aia_operand_set_insert(all_uses, op);
76         }
77     }
78 }
79
80 static void x86_32_reg_vars_insert_call_ops(Aia_Block *b,
81     Aia_Operand_Set *all_defs, Aia *aia)
82 {
83     Aia_Instr *in;
84     AIA_BLOCK_FOR_EACH_INSTRUCTION(b, in) {
85         if (aia_instr_get_type(in) != AIA_CALL)
86             continue;
87
88         Instr_Meta *imeta = in->meta_data;
89         Aia_Operand_Set *use_set = &imeta->use_set;
90         Aia_Operand_Set *def_set = &imeta->def_set;
91
92         Aia_Operand *call_lbl = aia_instr_get_src_op(in, 0);
93         Const_String callee_name;
94         if (aia_operand_get_type(call_lbl) == AIA_OPERAND_LABEL)
95             callee_name = aia_operand_label_get_name(call_lbl);
96         else
97             callee_name = NULL;
98         Const_String curr_func = __aia_get_curr_func_name(aia);
99     }

```

```

100     Aia_Operand *op;
101     AIA_OPERAND_SET_FOR_EACH_OPERAND(all_defs, op) {
102         if (aia_func_uses(callee_name, curr_func, op, aia))
103             aia_operand_set_insert(use_set, op);
104         if (func_kills_operand(callee_name, curr_func, op, aia)) {
105             aia_operand_set_insert(def_set, op);
106             aia_operand_set_insert(use_set, op);
107         }
108     }
109 }
110 }
111
112 static void x86_32_reg_vars_ins_entry_defs(Aia_Operand_Set *def_set,
113     Aia_Operand_Set *all_uses)
114 {
115     Aia_Operand *op;
116     AIA_OPERAND_SET_FOR_EACH_OPERAND(all_uses, op)
117         aia_operand_set_insert(def_set, op);
118 }
119
120 static void x86_32_reg_vars_init(Aia *aia)
121 {
122     inline Instr_Meta *get_last_instr_meta(Aia_Block *b)
123     {
124         Aia_Instr *in = __aia_block_peek_last_instr(b);
125         return in->meta_data;
126     }
127
128     inline Instr_Meta *get_first_instr_meta(Aia_Block *b)
129     {
130         Aia_Instr *in = __aia_block_peek_first_instr(b);
131         return in->meta_data;
132     }
133
134     Aia_Block *b;
135     Instr_Meta *imeta;
136     Aia_Operand_Set all_defs = AIA_OPERAND_SET_INIT();
137     Aia_Operand_Set all_uses = AIA_OPERAND_SET_INIT();
138
139     Aia_Section *sec;
140     AIA_FOR_EACH_SECTION(aia, sec) {
141         __aia_set_curr_func(aia, NULL);
142         AIA_SECTION_FOR_EACH_BLOCK(sec, b)
143             x86_32_reg_vars_init_block(b);
144
145         imeta = get_last_instr_meta(sec->exit_block);
146         AIA_SECTION_FOR_EACH_BLOCK(sec, b)
147             x86_32_reg_vars_ins_glob_uses_defs(b, &imeta->use_set,
148                 &all_defs, &all_uses, aia);
149
150         imeta = get_first_instr_meta(sec->entry_block);
151         x86_32_reg_vars_ins_entry_defs(&imeta->def_set, &all_uses);
152         aia_operand_set_for_each_destroy(&all_uses);
153
154         AIA_SECTION_FOR_EACH_BLOCK(sec, b)
155             x86_32_reg_vars_insert_call_ops(b, &all_defs, aia);
156
157         aia_operand_set_for_each_destroy(&all_defs);
158
159         Aia_Func *func;
160         AIA_SECTION_FOR_EACH_FUNC(sec, func) {
161             __aia_set_curr_func(aia, func);
162             AIA_FUNC_FOR_EACH_BLOCK(func, b)
163                 x86_32_reg_vars_init_block(b);
164
165             imeta = get_last_instr_meta(func->exit_block);
166             AIA_FUNC_FOR_EACH_BLOCK(func, b)
167                 x86_32_reg_vars_ins_glob_uses_defs(b, &imeta->use_set,
168                     &all_defs, &all_uses, aia);
169
170             imeta = get_first_instr_meta(func->entry_block);
171             x86_32_reg_vars_ins_entry_defs(&imeta->def_set,
172                 &all_uses);
173             aia_operand_set_for_each_destroy(&all_uses);

```

```

174
175     AIA_FUNC_FOR_EACH_BLOCK(func, b)
176         x86_32_reg_vars_insert_call_ops(b, &all_defs, aia);
177
178     aia_operand_set_for_each_destroy(&all_defs);
179
180 #if 0
181     Aia_Func_Trampoline *tramp;
182     AIA_FUNC_FOR_EACH_TRAMPOLINE(func, tramp) {
183         x86_32_reg_vars_init_block(tramp->block);
184         imeta = get_last_instr_meta(tramp->block);
185
186         x86_32_reg_vars_ins_glob_uses_defs(tramp->block,
187             &imeta->use_set, &all_defs, &all_uses, aia);
188
189         imeta = get_first_instr_meta(tramp->block);
190         x86_32_reg_vars_ins_entry_defs(&imeta->def_set,
191             &all_uses);
192         aia_operand_set_for_each_destroy(&all_uses);
193
194         x86_32_reg_vars_insert_call_ops(b, &all_defs, aia);
195
196         aia_operand_set_for_each_destroy(&all_defs);
197     }
198 #endif
199 }
200 }
201 }
202
203 static inline void x86_32_reg_vars_instr_destroy(Aia_Instr *in)
204 {
205     Instr_Meta *imeta = in->meta_data;
206     if (imeta) {
207         aia_operand_set_for_each_destroy(&imeta->use_set);
208         aia_operand_set_for_each_destroy(&imeta->def_set);
209         free_mem(imeta);
210     }
211 }
212
213 static void x86_32_reg_vars_destroy(Aia *aia)
214 {
215     Aia_Instr *in;
216     Aia_Block *b;
217
218     Aia_Section *sec;
219     AIA_FOR_EACH_SECTION(aia, sec) {
220         AIA_SECTION_FOR_EACH_BLOCK(sec, b) {
221             AIA_BLOCK_FOR_EACH_INSTRUCTION(b, in)
222                 x86_32_reg_vars_instr_destroy(in);
223         }
224
225         Aia_Func *func;
226         AIA_SECTION_FOR_EACH_FUNC(sec, func) {
227             AIA_FUNC_FOR_EACH_BLOCK(func, b) {
228                 AIA_BLOCK_FOR_EACH_INSTRUCTION(b, in)
229                     x86_32_reg_vars_instr_destroy(in);
230             }
231         }
232     #if 0
233         Aia_Func_Trampoline *tramp;
234         AIA_FUNC_FOR_EACH_TRAMPOLINE(func, tramp) {
235             b = tramp->block;
236             AIA_BLOCK_FOR_EACH_INSTRUCTION(b, in)
237                 x86_32_reg_vars_instr_destroy(in);
238         }
239     #endif
240 }
241 }
242 }
243
244 static void x86_32_reg_vars_insert_uses_instr(Aia_Instr *pred,
245     Aia_Instr *in, Aia *aia)
246 {
247     Instr_Meta *imeta = in->meta_data;

```

```

248     Aia_Operand_Set *set = &imeta->use_set;
249     Instr_Meta *pred_imeta = pred->meta_data;
250     Aia_Operand_Set *pred_set = &pred_imeta->use_set;
251
252     Aia_Operand *dest = aia_instr_get_dest_op(pred);
253     Aia_Operand *op;
254     AIA_OPERAND_SET_FOR_EACH_OPERAND(set, op) {
255         if (!dest || !aia_operands_equal(dest, op)) {
256             /* Don't change aia->meta_data when it's != NULL,
257              * It might be pointing to Aia_Meta. */
258             if (aia_operand_set_insert(pred_set, op) && !aia->meta_data)
259                 aia->meta_data = INT_TO_PTR(true);
260         }
261     }
262
263     if (!aia_instr_is_concrete(in))
264         return;
265
266     Aia_Operand_Set *def_set = &imeta->def_set;
267     AIA_INSTR_FOR_EACH_SRC(in, op) {
268         if (aia_operand_set_contains(def_set, op))
269             /* Don't change aia->meta_data when it's != NULL,
270              * It might be pointing to Aia_Meta. */
271             if (aia_operand_set_insert(set, op) && !aia->meta_data)
272                 aia->meta_data = INT_TO_PTR(true);
273     }
274 }
275
276 static void x86_32_reg_vars_insert_defs_mov(Aia_Instr *suc,
277     Aia_Instr *in, Aia *aia)
278 {
279     Instr_Meta *imeta = in->meta_data;
280     Aia_Operand_Set *set = &imeta->def_set;
281     Instr_Meta *suc_imeta = suc->meta_data;
282     Aia_Operand_Set *suc_set = &suc_imeta->def_set;
283
284     Aia_Operand *d = aia_instr_get_dest_op(suc);
285     Aia_Operand *s = aia_instr_get_src_op(suc, 0);
286
287     Aia_Operand *op;
288     AIA_OPERAND_SET_FOR_EACH_OPERAND(set, op) {
289         bool src_same = aia_operands_equal(s, op);
290         if (!d || (!aia_operands_equal(d, op) && !src_same)) {
291             /* Don't change aia->meta_data when it's != NULL,
292              * It might be pointing to Aia_Meta. */
293             if (aia_operand_set_insert(suc_set, op) && !aia->meta_data)
294                 aia->meta_data = INT_TO_PTR(true);
295         } else if (src_same) {
296             aia_operand_set_insert(&imeta->use_set, op);
297         }
298     }
299 }
300
301 static void x86_32_reg_vars_insert_defs_instr(Aia_Instr *suc,
302     Aia_Instr *in, Aia *aia)
303 {
304     switch (aia_instr_get_type(suc)) {
305     case AIA_MOV:
306     case AIA_MOVS:
307         x86_32_reg_vars_insert_defs_mov(suc, in, aia);
308         return;
309
310     default:
311         break;
312     }
313
314     Instr_Meta *imeta = in->meta_data;
315     Aia_Operand_Set *set = &imeta->def_set;
316     Instr_Meta *suc_imeta = suc->meta_data;
317     Aia_Operand_Set *suc_set = &suc_imeta->def_set;
318
319     Aia_Operand *dest = aia_instr_get_dest_op(suc);
320     Aia_Operand *op;
321     AIA_OPERAND_SET_FOR_EACH_OPERAND(set, op) {

```

```

322         if (!dest || !aia_operands_equal(dest, op)) {
323             if (aia_operand_set_insert(suc_set, op) && !aia->meta_data)
324                 aia->meta_data = INT_TO_PTR(true);
325         }
326     }
327 }
328
329 static void x86_32_reg_vars_build_uses_instr(Aia_Instr *in, Aia *aia)
330 {
331     Aia_Instr *pred = aia_instr_get_predecessor(in);
332     if (!pred) {
333         Aia_Block *b = aia_instr_get_block(in);
334         Aia_Block *pred_block;
335         AIA_BLOCK_FOR_EACH_PREDECESSOR(b, pred_block) {
336             x86_32_reg_vars_insert_uses_instr(
337                 __aia_block_peek_last_instr(pred_block), in, aia);
338         }
339     } else {
340         x86_32_reg_vars_insert_uses_instr(pred, in, aia);
341     }
342 }
343
344 static void x86_32_reg_vars_build_defs_instr(Aia_Instr *in, Aia *aia)
345 {
346     Aia_Instr *suc = aia_instr_get_sucessor(in);
347     if (!suc) {
348         Aia_Block *b = aia_instr_get_block(in);
349         Aia_Block *suc_block;
350         AIA_BLOCK_FOR_EACH_SUCESSOR(b, suc_block) {
351             x86_32_reg_vars_insert_defs_instr(
352                 __aia_block_peek_first_instr(suc_block), in, aia);
353         }
354     } else {
355         x86_32_reg_vars_insert_defs_instr(suc, in, aia);
356     }
357 }
358
359 static inline void x86_32_reg_vars_build_uses_block(Aia_Block *b, Aia *aia)
360 {
361     Aia_Instr *in;
362     AIA_BLOCK_FOR_EACH_INSTRUCTION_REVERSED(b, in)
363         x86_32_reg_vars_build_uses_instr(in, aia);
364 }
365
366 static inline void x86_32_reg_vars_build_defs_block(Aia_Block *b, Aia *aia)
367 {
368     Aia_Instr *in;
369     AIA_BLOCK_FOR_EACH_INSTRUCTION(b, in)
370         x86_32_reg_vars_build_defs_instr(in, aia);
371 }
372
373 static void x86_32_reg_vars_build_uses(Aia *aia)
374 {
375     do {
376         aia->meta_data = INT_TO_PTR(false);
377
378         Aia_Block *b;
379         Aia_Section *sec;
380         AIA_FOR_EACH_SECTION(aia, sec) {
381             __aia_set_curr_func(aia, NULL);
382             AIA_SECTION_FOR_EACH_BLOCK_REVERSED(sec, b)
383                 x86_32_reg_vars_build_uses_block(b, aia);
384
385             Aia_Func *func;
386             AIA_SECTION_FOR_EACH_FUNC(sec, func) {
387                 __aia_set_curr_func(aia, func);
388                 AIA_FUNC_FOR_EACH_BLOCK_REVERSED(func, b)
389                     x86_32_reg_vars_build_uses_block(b, aia);
390
391 #if 0
392                 Aia_Func_Trampoline *tramp;
393                 AIA_FUNC_FOR_EACH_TRAMPOLINE(func, tramp)
394                     x86_32_reg_vars_build_uses_block(tramp->block, aia);
395 #endif

```



```

396     }
397 }
398 } while (aia->meta_data);
399 }
400
401 static void x86_32_reg_vars_build_defs(Aia *aia)
402 {
403     do {
404         aia->meta_data = INT_TO_PTR(false);
405
406         Aia_Block *b;
407         Aia_Section *sec;
408         AIA_FOR_EACH_SECTION(aia, sec) {
409             __aia_set_curr_func(aia, NULL);
410             AIA_SECTION_FOR_EACH_BLOCK(sec, b)
411                 x86_32_reg_vars_build_defs_block(b, aia);
412
413             Aia_Func *func;
414             AIA_SECTION_FOR_EACH_FUNC(sec, func) {
415                 __aia_set_curr_func(aia, func);
416                 AIA_FUNC_FOR_EACH_BLOCK(func, b)
417                     x86_32_reg_vars_build_defs_block(b, aia);
418
419 #if 0
420                 Aia_Func_Trampoline *tramp;
421                 AIA_FUNC_FOR_EACH_TRAMPOLINE(func, tramp)
422                     x86_32_reg_vars_build_defs_block(tramp->block, aia);
423 #endif
424             }
425         }
426     } while (aia->meta_data);
427 }
428
429 static bool x86_32_reg_vars_use_starts(Aia_Instr *in, Aia_Operand *op)
430 {
431     Instr_Meta *imeta = in->meta_data;
432
433     Aia_Instr *suc = aia_instr_get_sucessor(in);
434     if (!suc)
435         return false;
436
437     Instr_Meta *smeta = suc->meta_data;
438     if (!aia_operand_set_contains(&imeta->use_set, op) &&
439         aia_operand_set_contains(&smeta->use_set, op))
440         return true;
441
442     return false;
443 }
444
445 static bool x86_32_reg_vars_def_ends(Aia_Instr *in, Aia_Operand *op)
446 {
447     Instr_Meta *imeta = in->meta_data;
448     if (imeta->is_inserted)
449         return false;
450
451     Aia_Instr *pred = aia_instr_get_predecessor(in);
452     if (!pred)
453         return false;
454
455     Instr_Meta *pmeta = pred->meta_data;
456     if (!aia_operand_set_contains(&imeta->def_set, op) &&
457         aia_operand_set_contains(&pmeta->def_set, op))
458         return true;
459
460     return false;
461 }
462
463 static inline Aia_Operand *x86_32_reg_vars_get_repl(Aia *aia, Aia_Operand *op)
464 {
465     Aia_Meta *ameta = aia->meta_data;
466     Aia_Operand *val = aia_operand_map_get_value(&ameta->subst_map, op);
467     if (!val) {
468         val = aia_operand_tmp_reg_alloc(aia);
469         aia_operand_map_insert(&ameta->subst_map, op, val);

```

```

470     }
471     return val;
472 }
473
474 static Aia_Operand *x86_32_reg_vars_replace_op(Aia_Instr *in, Int idx,
475     Aia_Operand *op, Aia *aia)
476 {
477     if (!x86_32_reg_vars_operand_significant(op))
478         return NULL;
479
480     Aia_Operand *val = x86_32_reg_vars_get_repl(aia, op);
481
482     aia_instr_replace_op(in, idx, val);
483     return val;
484 }
485
486 static void x86_32_reg_vars_insert_sucessors_use_set(Aia_Instr *in, Aia *aia)
487 {
488     Aia_Instr *suc = aia_instr_get_sucessor(in);
489     assert(suc);
490     x86_32_reg_vars_insert_uses_instr(in, suc, aia);
491 }
492
493 static void x86_32_reg_vars_insert_predecessors_def_set(Aia_Instr *in,
494     Aia *aia)
495 {
496     Aia_Instr *pred = aia_instr_get_predecessor(in);
497     assert(pred);
498     x86_32_reg_vars_insert_defs_instr(in, pred, aia);
499 }
500
501 static void x86_32_reg_vars_replace_instr(Aia_Instr *in, Aia *aia)
502 {
503     if (!aia_instr_is_concrete(in))
504         return;
505
506     Instr_Meta *imeta = in->meta_data;
507     if (imeta->is_inserted)
508         return;
509
510     Aia_Operand *op;
511     Aia_Operand *val;
512     Aia_Operand_Set *def_set = &imeta->def_set;
513
514     switch (aia_instr_get_type(in)) {
515     case AIA_MOV:
516     case AIA_MOVS:
517         op = aia_instr_get_src_op(in, 0);
518         if (x86_32_reg_vars_def_ends(in, op)) {
519             val = x86_32_reg_vars_get_repl(aia, op);
520             Aia_Instr *new_mov = aia_mov_before(op, val, in,
521                 aia_instr_get_dest_op_size(in));
522             x86_32_reg_vars_init_instr(new_mov, true);
523             x86_32_reg_vars_insert_sucessors_use_set(new_mov, aia);
524             x86_32_reg_vars_insert_predecessors_def_set(new_mov, aia);
525         }
526         break;
527
528     default:
529         break;
530     }
531
532     if (aia_instr_get_type(in) != AIA_CALL) {
533         Int idx = 0;
534         AIA_INSTR_FOR_EACH_SRC(in, op) {
535             if (!aia_operand_set_contains(def_set, op) &&
536                 !x86_32_reg_vars_def_ends(in, op))
537                 x86_32_reg_vars_replace_op(in, idx, op, aia);
538             ++idx;
539         }
540     }
541
542     op = aia_instr_get_dest_op(in);
543     if (!op)

```

```

544     return;
545
546     __aia_operand_acquire(op);
547     val = x86_32_reg_vars_replace_op(in, -1, op, aia);
548     if (val && x86_32_reg_vars_use_starts(in, op)) {
549         Aia_Instr *new_mov = aia_mov_after(val, op, in,
550             aia_instr_get_dest_op_size(in));
551         x86_32_reg_vars_init_instr(new_mov, true);
552         x86_32_reg_vars_insert_sucessors_use_set(new_mov, aia);
553         x86_32_reg_vars_insert_predecessors_def_set(new_mov, aia);
554     }
555     __aia_operand_release(op);
556 }
557
558 static inline void x86_32_reg_vars_replace_block(Aia_Block *b, Aia *aia)
559 {
560     Aia_Instr *in;
561     AIA_BLOCK_FOR_EACH_INSTRUCTION(b, in)
562         x86_32_reg_vars_replace_instr(in, aia);
563 }
564
565 static inline void x86_32_reg_vars_init_maps(Aia *aia)
566 {
567     Aia_Meta *ameta = aia->meta_data;
568     ameta->subst_map = AIA_OPERAND_MAP_INIT();
569 }
570
571 static inline void x86_32_reg_vars_finalize_subst_map(Aia *aia)
572 {
573     Aia_Meta *ameta = aia->meta_data;
574     Aia_Operand_Map *smap = &ameta->subst_map;
575     Aia_Operand_Map *rmap = ameta->repl_map;
576
577     Aia_Operand_Map_Entry *e;
578     AIA_OPERAND_MAP_FOR_EACH_ENTRY(smap, e)
579         aia_operand_map_insert(rmap, e->value, e->operand);
580
581     aia_operand_map_for_each_destroy(smap);
582 }
583
584 static inline void x86_32_reg_vars_init_ameta(Aia *aia,
585     Aia_Operand_Map *repl_map)
586 {
587     Aia_Meta *ameta = ALLOC_NEW(Aia_Meta);
588     aia->meta_data = ameta;
589     ameta->repl_map = repl_map;
590 }
591
592 static inline void x86_32_reg_vars_finalize_ameta(Aia *aia)
593 {
594     Aia_Meta *ameta = aia->meta_data;
595     free_mem(ameta);
596 }
597
598 static void x86_32_reg_vars_replace(Aia *aia, Aia_Operand_Map *repl_map)
599 {
600     x86_32_reg_vars_init_ameta(aia, repl_map);
601
602     Aia_Block *b;
603     Aia_Section *sec;
604     AIA_FOR_EACH_SECTION(aia, sec) {
605         x86_32_reg_vars_init_maps(aia);
606         AIA_SECTION_FOR_EACH_BLOCK(sec, b)
607             x86_32_reg_vars_replace_block(b, aia);
608         x86_32_reg_vars_finalize_subst_map(aia);
609
610         Aia_Func *func;
611         AIA_SECTION_FOR_EACH_FUNC(sec, func) {
612             x86_32_reg_vars_init_maps(aia);
613             AIA_FUNC_FOR_EACH_BLOCK(func, b)
614                 x86_32_reg_vars_replace_block(b, aia);
615             x86_32_reg_vars_finalize_subst_map(aia);
616         }
617     }

```

```

618         x86_32_reg_vars_init_maps(aia);
619         Aia_Func_Trampoline *tramp;
620         AIA_FUNC_FOR_EACH_TRAMPOLINE(func, tramp)
621             x86_32_reg_vars_replace_block(tramp->block, aia);
622         x86_32_reg_vars_finalize_subst_map(aia);
623     #endif
624     }
625 }
626
627     x86_32_reg_vars_finalize_ameta(aia);
628 }
629
630 static void x86_32_reg_vars_dump_sets(FILE *stream, Aia_Instr *in)
631 {
632     Instr_Meta *imeta = in->meta_data;
633     file_print_message(stream, S("\t# uses { "));
634     Aia_Operand_Set *use_set = &imeta->use_set;
635
636     Uns size = hash_map_size(use_set);
637     if (!size)
638         goto def_set;
639
640     Aia_Operand *op;
641     AIA_OPERAND_SET_FOR_EACH_OPERAND(use_set, op) {
642         aia_operand_dump(stream, op, false);
643         if (--size)
644             file_print_message(stream, S(", "));
645     }
646
647     def_set:
648     file_print_message(stream, S(" } defs { "));
649     Aia_Operand_Set *def_set = &imeta->def_set;
650
651     size = hash_map_size(def_set);
652     if (!size)
653         goto out;
654
655     AIA_OPERAND_SET_FOR_EACH_OPERAND(def_set, op) {
656         aia_operand_dump(stream, op, false);
657         if (--size)
658             file_print_message(stream, S(", "));
659     }
660
661     out:
662     file_print_message(stream, S(" }"));
663
664     if (imeta->is_inserted)
665         file_print_message(stream, S(" inserted"));
666 }
667
668 void x86_32_reg_vars_dump(Aia *aia, Const_String postfix)
669 {
670     aia_set_instr_dump_callback(aia, x86_32_reg_vars_dump_sets);
671
672     String fname = string_from_format(S("%S.vitaly.%S"),
673         aia_get_file_name(aia), postfix);
674     FILE *f = file_open(fname, S("w"));
675     if (!f)
676         fatal_error(S("unable to open file %S for intermediate "
677             "code dump [%m]\n"), fname);
678
679     string_destroy(fname);
680     aia_dump(aia, f);
681     file_close(f);
682
683     aia_clear_instr_dump_callback(aia);
684 }
685
686 Aia_Operand_Map *x86_32_reg_vars(Aia *aia)
687 {
688     Aia_Operand_Map *repl_map = aia_operand_map_alloc();
689     if (!cmdopts.opt_reg_vars)
690         goto out;
691

```

```

692
693     aia_func_access(aia);
694     aia_collect_func_kills(aia);
695
696     x86_32_reg_vars_init(aia);
697
698     aia_func_access_destroy(aia);
699     aia_destroy_func_kills(aia);
700
701     x86_32_reg_vars_build_defs(aia);
702     x86_32_reg_vars_build_uses(aia);
703
704     if (cmdopts.dump_reg_vars_liveness_ic)
705         x86_32_reg_vars_dump(aia, S("reg-vars-liveness-ic"));
706
707     x86_32_reg_vars_replace(aia, repl_map);
708
709     if (cmdopts.dump_reg_vars_ic)
710         x86_32_reg_vars_dump(aia, S("reg-vars-ic"));
711
712     x86_32_reg_vars_destroy(aia);
713
714 out:
715     return repl_map;
716 }

```

:

A.8.25 src/x86_32/x86_32_reg_vars.h

```

1  #ifndef X86_32_REG_VARS_H
2  #define X86_32_REG_VARS_H
3
4  #include <aia/aia.h>
5  #include <aia/aia_operand_map.h>
6
7  /* Modifies AIA such that variables are kept in
8   * registers as longer.
9   *
10  * Returns mapping from register to memory location register
11  * allocator can use for choosing spill locations. */
12 Aia_Operand_Map *x86_32_reg_vars(Aia *aia);
13
14 #endif // X86_32_REG_VARS_H

```

A.9 Code Generation

:

A.9.1 src/vit/end.s

```

1  .section .init
2      popl %ebp
3      popl %edi
4      popl %esi
5      popl %ebx
6
7  .section .fini
8      popl %ebp
9      popl %edi
10     popl %esi
11     popl %ebx
12
13 .section .Vit_libinit, "xa"
14     ret

```

:

A.9.2 src/vit/ini.s

```

1  .section .init
2      pushl %ebx
3      pushl %esi
4      pushl %edi
5      pushl %ebp
6      call  _Vit_libinit
7
8  .section .fini
9      pushl %ebx
10     pushl %esi
11     pushl %edi
12     pushl %ebp
13
14     .section .Vit_libinit, "xa"
15     .type _Vit_libinit, @function
16     .align 4
17     _Vit_libinit:

```

:

A.9.3 src/vit/lib.c

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <inttypes.h>
4  #include <errno.h>
5
6  void _Vit_writelnn()
7  {
8      puts("null");
9  }
10
11 void _Vit_writelni(int32_t i)
12 {
13     printf("%i PRId32\n", i);
14 }
15
16 void _Vit_writelnr(void *rec)
17 {
18     if (!rec)
19         _Vit_writelnn();
20     else
21         printf("%p\n", rec);
22 }
23
24 void _Vit_writelna(void *ary)
25 {
26     _Vit_writelnr(ary);
27 }
28
29 void _Vit_writelnc(char c)
30 {
31     printf("%c\n", c);
32 }
33
34 void _Vit_writelns(const char *s)
35 {
36     if (s)
37         puts(s);
38     else
39         _Vit_writelnn();
40 }
41
42 void _Vit_writelnb(int8_t b)

```

```

43 {
44     if (b)
45         puts("true");
46     else
47         puts("false");
48 }
49
50 void *_Vit_allocab(int32_t size)
51 {
52     if (size < 0) {
53         errno = EINVAL;
54         return NULL;
55     }
56     int32_t *ret = malloc(size + sizeof(int32_t));
57     if (ret)
58         *ret = size;
59     return ret + 1;
60 }
61
62 void *_Vit_alloca(int32_t size)
63 {
64     if (size < 0) {
65         errno = EINVAL;
66         return NULL;
67     }
68     if (size > (INT32_MAX >> 2) - (int32_t)sizeof(int32_t)) {
69         errno = ERANGE;
70         return NULL;
71     }
72     int32_t *ret = malloc((size << 2) + sizeof(int32_t));
73     if (ret)
74         *ret = size;
75     return ret + 1;
76 }
77
78 void *_Vit_alloc(int32_t size)
79 {
80     if (size < 0) {
81         errno = EINVAL;
82         return NULL;
83     }
84     if (!size)
85         size = 1;
86     return malloc(size);
87 }
88
89 void _Vit_deletea(int32_t *ary)
90 {
91     if (ary)
92         free(ary - 1);
93 }
94
95 void _Vit_delete(void *p)
96 {
97     free(p);
98 }

```

:

A.9.4 src/vit/lib.s

```

1  .section .rodata
2  writelni_fmt:
3      .string "%d\n"
4  writelnc_fmt:
5      .string "%c\n"
6  writelnb_true:
7      .string "true"
8  writelnb_false:
9      .string "false"
10 writelnr_fmt:

```

```

11     .string "%p\n"
12 writelnn_str:
13     .string "null"
14
15     .align 4
16     .type _Vit_writelnn, @function
17     .globl _Vit_writelnn
18     _Vit_writelnn:
19     .cfi_startproc
20         pushl $writelnn_str
21         call puts
22         addl $4, %esp
23         ret
24     .cfi_endproc
25     .size _Vit_writelnn, . - _Vit_writelnn
26
27     .align 4
28     .type _Vit_writelni, @function
29     .globl _Vit_writelni
30     _Vit_writelni:
31     .cfi_startproc
32         pushl 4(%esp)
33         pushl $writelni_fmt
34         call printf
35         addl $8, %esp
36         ret
37     .cfi_endproc
38     .size _Vit_writelni, . - _Vit_writelni
39
40     .align 4
41     .type null_check, @function
42     null_check:
43     .cfi_startproc
44         cmpl $0, 8(%esp)
45         jne 1f
46         pushl $writelnn_str
47         call puts
48         addl $8, %esp
49     1:
50         ret
51     .cfi_endproc
52     .size null_check, . - null_check
53
54     .align 4
55     .type _Vit_writelnr, @function
56     .globl _Vit_writelnr
57     _Vit_writelnr:
58     .cfi_startproc
59         call null_check
60         pushl 4(%esp)
61         pushl $writelnr_fmt
62         call printf
63         addl $8, %esp
64         ret
65     .cfi_endproc
66     .size _Vit_writelnr, . - _Vit_writelnr
67
68     .align 4
69     .type _Vit_writelna, @function
70     .globl _Vit_writelna
71     _Vit_writelna:
72     .cfi_startproc
73         jmp _Vit_writelnr
74     .cfi_endproc
75     .size _Vit_writelna, . - _Vit_writelna
76
77     .align 4
78     .type _Vit_writelnc, @function
79     .globl _Vit_writelnc
80     _Vit_writelnc:
81     .cfi_startproc
82         pushl 4(%esp)
83         pushl $writelnc_fmt
84         call printf

```



```

85     addl $8, %esp
86     ret
87 .cfi_endproc
88 .size _Vit_writeln, . - _Vit_writeln
89
90 .align 4
91 .type _Vit_writeln, @function
92 .globl _Vit_writeln
93 _Vit_writeln:
94 .cfi_startproc
95     call null_check
96     pushl 4(%esp)
97     call puts
98     addl $4, %esp
99     ret
100 .cfi_endproc
101 .size _Vit_writeln, . - _Vit_writeln
102
103 .align 4
104 .type _Vit_writelnb, @function
105 .globl _Vit_writelnb
106 _Vit_writelnb:
107 .cfi_startproc
108     cmpb $0, 4(%esp)
109     jne 1f
110     pushl $writelnb_false
111     jmp 2f
112 1:
113     pushl $writelnb_true
114 2:
115     call puts
116     addl $4, %esp
117     ret
118 .cfi_endproc
119 .size _Vit_writelnb, . - _Vit_writelnb
120
121 .align 4
122 .type _Vit_allocab, @function
123 .globl _Vit_allocab
124 _Vit_allocab:
125 .cfi_startproc
126     cmpl $0, 4(%esp)
127     jle 2f
128     addl $4, 4(%esp)
129     pushl 4(%esp)
130     call malloc
131     testl %eax, %eax # Return if nothing was allocated (error probably).
132     jnz 1f
133     popl %ecx # expose return address.
134     ret
135 1:
136     addl $4, %esp
137     movl 4(%esp), %edx
138     subl $4, %edx
139     movl %edx, (%eax)
140     addl $4, %eax
141     ret
142 2:
143     xorl %eax, %eax
144     ret
145 .cfi_endproc
146 .size _Vit_allocab, . - _Vit_allocab
147
148 .align 4
149 .type _Vit_allocl, @function
150 .globl _Vit_allocl
151 _Vit_allocl:
152 .cfi_startproc
153     cmpl $0, 4(%esp)
154     jl 2f
155     shll $2, 4(%esp)
156     addl $4, 4(%esp)
157     pushl 4(%esp)
158     call malloc

```

```

159     testl %eax, %eax # Return if nothing was allocated (error probably).
160     jnz 1f
161     popl %ecx      # expose return address.
162     ret
163 1:
164     addl $4, %esp
165     movl 4(%esp), %edx
166     shr1 $2, %edx
167     sub $1, %edx
168     movl %edx, (%eax)
169     addl $4, %eax
170     ret
171 2:
172     xorl %eax, %eax
173     ret
174 .cfi_endproc
175 .size _Vit_allocal, . - _Vit_allocal
176
177 .align 4
178 .type _Vit_alloc, @function
179 .globl _Vit_alloc
180 _Vit_alloc:
181 .cfi_startproc
182     cmpl $0, 4(%esp)
183     jle 1f
184     jmp malloc
185 1:
186     xorl %eax, %eax
187     ret
188 .cfi_endproc
189 .size _Vit_alloc, . - _Vit_alloc
190
191 .align 4
192 .type _Vit_deletea, @function
193 .globl _Vit_deletea
194 _Vit_deletea:
195 .cfi_startproc
196     cmpl $0, 4(%esp)
197     je 1f
198     subl $4, 4(%esp)
199     jmp free
200 1:
201     ret
202 .cfi_endproc
203 .size _Vit_deletea, . - _Vit_deletea
204
205 .align 4
206 .type _Vit_delete, @function
207 .globl _Vit_delete
208 _Vit_delete:
209 .cfi_startproc
210     jmp free
211 .cfi_endproc
212 .size _Vit_delete, . - _Vit_delete

```

:

A.9.5 src/vit/retmain.s

```

1 .section .text
2 .align 4
3 .type main, @function
4 .globl main
5 main:
6     xorl %eax, %eax
7     ret
8 .size main, . - main

```

:

A.9.6 src/vit/vitmain.s

```

1  .section .text
2  .align 4
3  .type main, @function
4  .globl main
5  main:
6  .cfi_startproc
7  /* Hack alert!
8   * Apparently glibc uses a pointer to the command line arguments the
9   * kernel pushed on the stack just before calling _start().
10   * Thus argc is also located at *((int *) (argv - 1)) implying
11   * argv is a vitally array of string. Just pass that to _Vit_main()! */
12   movl 8(%esp), %eax
13   movl %eax, 4(%esp)
14   jmp  _Vit_main
15 .cfi_endproc
16 .size main, . - main
17
18 /*
19 .section .text
20 .align 4
21 .type main, @function
22 .globl main
23 main:
24 .cfi_startproc
25   pushl %ebp
26   movl %esp, %ebp
27
28   movl 8(%ebp), %ecx # argc
29   leal -4(, %ecx, 4), %edx
30   addl 12(%ebp), %edx # argv, 4*(argc-1)
31
32   pushl (%edx)
33   cmpl %edx, 12(%ebp)
34   je   enter_vit
35 1:
36   subl $4, %edx
37   pushl (%edx)
38   cmpl %edx, 12(%ebp)
39   jne  1b
40
41 enter_vit:
42   pushl 8(%ebp)
43   leal 4(%esp), %eax
44   pushl %eax
45   call _Vit_main
46   leave
47   ret
48 .cfi_endproc
49 .size main, . - main
50 */

```

:

A.9.7 src/x86_32/x86_32.c

```

1  #include "x86_32.h"
2  #include "x86_32_regs.h"
3  #include "x86_32_normalize.h"
4  #include "x86_32_reg_alloc.h"
5  #include "x86_32_func_normalize.h"
6  #include "x86_32_reg_vars.h"
7  #include "x86_32_emit.h"
8  #include <main.h>
9
10 String x86_32_gen(Aia *aia)
11 {
12     String asm_fname = NULL;
13

```

```

14     if (!aia_is_valid(aia))
15         goto out;
16
17     x86_32_regs_init(aia);
18
19     x86_32_normalize(aia);
20
21     Aia_Operand_Map *repl_map = x86_32_reg_vars(aia);
22
23     x86_32_reg_alloc(aia, repl_map);
24     aia_operand_map_destroy(repl_map);
25
26     x86_32_func_normalize(aia);
27
28     asm_fname = x86_32_emit(aia);
29
30     x86_32_regs_release();
31
32 out:
33     return asm_fname;
34 }

```

:

A.9.8 src/x86_32/x86_32.h

```

1  #ifndef X86_32_H
2  #define X86_32_H
3
4  #include <aia/aia.h>
5
6  /* Returns name of generated asm file.
7   * NULL if the AIA is invalid. */
8  String x86_32_gen(Aia *aia);
9
10 #endif // X86_32_H

```

:

A.9.9 src/x86_32/x86_32_emit.c

```

1  #include "x86_32_emit.h"
2  #include "x86_32_regs.h"
3  #include <main.h>
4
5  #undef DEBUG_TYPE
6  #define DEBUG_TYPE x86-32-emit
7
8  #define VITMAIN_CSTR "_Vit_main"
9  #define VITMAIN_STR S(VITMAIN_CSTR)
10 #define MAIN_CSTR "main"
11 #define MAIN_STR S(MAIN_CSTR)
12
13 typedef struct Aia_Meta_Data {
14     Uns cfi_offset;
15     bool inside_func;
16 } Aia_Meta_Data;
17
18 static Uns x86_32_emit_prev_line;
19
20 static void __x86_32_emit_line_number(FILE *asm_file, Aia_Instr *in)
21 {
22     File_Location *loc = aia_instr_get_location(in);
23     if ((cmdopts.dump_asm || cmdopts.assemble_only) && loc->line &&
24         loc->line != x86_32_emit_prev_line) {
25         x86_32_emit_prev_line = loc->line;
26         file_print_message(asm_file, S("# line %U\n"), loc->line);

```

```

27     }
28 }
29
30 static void x86_32_emit_line_number(FILE *asm_file, Aia_Instr *in)
31 {
32     switch (aia_instr_get_type(in)) {
33         AIA_CASE_COND_JUMP:
34         case AIA_JMP:
35             break;
36         default:
37             __x86_32_emit_line_number(asm_file, in);
38     }
39 }
40
41 static void x86_32_operand_dump(FILE *asm_file, Aia_Operand *op,
42     bool print_int_dollar)
43 {
44     switch (op->op_type) {
45         case AIA_OPERAND_LABEL_ADDR:
46             if (op->iden.op_label->offset)
47                 file_print_message(asm_file, S("$S+% PRId32),
48                     op->iden.op_label->label_name,
49                     op->iden.op_label->offset);
50             else
51                 file_print_message(asm_file, S("$S"),
52                     op->iden.op_label->label_name);
53             break;
54
55         case AIA_OPERAND_REG:
56             assert(x86_32_is_concrete_reg(op));
57             file_print_message(asm_file, S("%%%S"), op->iden.op_name);
58             break;
59
60         case AIA_OPERAND_LABEL:
61             if (op->iden.op_label->offset)
62                 file_print_message(asm_file, S("$S+% PRId32),
63                     op->iden.op_label->label_name,
64                     op->iden.op_label->offset);
65             else
66                 file_print_message(asm_file, S("$S"),
67                     op->iden.op_label->label_name);
68             break;
69
70         case AIA_OPERAND_CONST_INT:
71             if (print_int_dollar)
72                 file_print_message(asm_file, S("$%" PRId32), op->iden.int_const);
73             else
74                 file_print_message(asm_file, S("%" PRId32), op->iden.int_const);
75             break;
76
77         case AIA_OPERAND_ADDR_REF:
78             if (op->iden.addr_ref->label)
79                 aia_operand_dump(asm_file, op->iden.addr_ref->label, false);
80             if (op->iden.addr_ref->disp && op->iden.addr_ref->label)
81                 file_print_message(asm_file, S("+"));
82             if (op->iden.addr_ref->disp)
83                 aia_operand_dump(asm_file, op->iden.addr_ref->disp, false);
84             file_print_message(asm_file, S("("));
85             if (op->iden.addr_ref->base)
86                 aia_operand_dump(asm_file, op->iden.addr_ref->base, false);
87             if (op->iden.addr_ref->index || op->iden.addr_ref->scale)
88                 file_print_message(asm_file, S(","));
89             if (op->iden.addr_ref->index) {
90                 aia_operand_dump(asm_file, op->iden.addr_ref->index, false);
91                 if (op->iden.addr_ref->scale)
92                     file_print_message(asm_file, S(","));
93             }
94             if (op->iden.addr_ref->scale)
95                 aia_operand_dump(asm_file, op->iden.addr_ref->scale, false);
96             file_print_message(asm_file, S(")"));
97             break;
98
99         case AIA_OPERAND_LOCAL_REF:
100             fatal_error(S("unexpected local ref AIA operand. Aborting...\n"));

```

```

101
102     case AIA_OPERAND_ARG:
103         fatal_error(S("unexpected arg ref AIA operand. Aborting...\n"));
104
105     case AIA_OPERAND_DISPLAY_REF:
106         fatal_error(S("unexpected display ref AIA operand. Aborting...\n"));
107
108     case AIA_OPERAND_BLOCK:;
109         Aia_Instr *lbl_instr =
110             __aia_block_peek_first_instr(op->iden.op_block);
111         aia_operand_dump(asm_file, aia_instr_get_dest_op(lbl_instr), false);
112         break;
113
114     default:
115         fatal_error(S("Unexpected operand type for x86-32 emit\n"));
116     }
117 }
118
119 static CONST_STRING(x86_32_instr_postfix_b, "b");
120
121 static CONST_STRING(x86_32_instr_postfix_l, "l");
122
123 typedef struct Pending_Jmp {
124     Aia_Instr *instr;
125     Const_String instr_name;
126     bool emit_second_jump;
127 } Pending_Jmp;
128
129 Pending_Jmp pending_jump;
130
131 static Const_String x86_32_get_instr_postfix(Aia_Instr *in, bool dest)
132 {
133     switch (dest ? aia_instr_get_dest_op_size(in) :
134             aia_instr_get_src_ops_size(in)) {
135     case AIA_BYTE:
136         return x86_32_instr_postfix_b;
137     case AIA_LONG:
138         return x86_32_instr_postfix_l;
139     default:
140         fatal_error(S("unexpected mov instruction. Aborting...\n"));
141     }
142 }
143
144 static inline Const_String x86_32_get_instr_postfix_dest(Aia_Instr *in)
145 {
146     return x86_32_get_instr_postfix(in, true);
147 }
148
149 static void x86_32_pending_jump_emit(FILE *asm_file)
150 {
151     if (!pending_jump.instr)
152         return;
153
154     __x86_32_emit_line_number(asm_file, pending_jump.instr);
155
156     file_print_message(asm_file, S("\t%S\t"), pending_jump.instr_name);
157     x86_32_operand_dump(asm_file,
158         aia_instr_get_src_op(pending_jump.instr, 0), true);
159
160     if (pending_jump.emit_second_jump) {
161         file_print_message(asm_file, S("\n\tjmp\t"));
162         x86_32_operand_dump(asm_file,
163             aia_instr_get_src_op(pending_jump.instr, 1), true);
164     }
165
166     file_print_message(asm_file, S("\n"));
167     pending_jump.instr = NULL;
168 }
169
170 static void x86_32_jump_emit(FILE *asm_file UNUSED, Aia_Instr *instr,
171     Const_String instr_name)
172 {
173     assert(instr->src_ops_size == AIA_LONG);
174     pending_jump.instr = instr;

```

```

175     pending_jump.instr_name = instr_name;
176 }
177
178 static void x86_32_mov_emit(FILE *asm_file, Aia_Instr *instr,
179     Const_String in_name)
180 {
181     assert(aia_instr_get_type(instr) == AIA_MOV);
182     assert(aia_instr_get_dest_op_size(instr) ==
183         aia_instr_get_src_ops_size(instr));
184
185     Const_String instr_postfix = x86_32_get_instr_postfix_dest(instr);
186
187     file_print_message(asm_file, S("\t%S\t"), in_name, instr_postfix);
188     x86_32_operand_dump(asm_file, aia_instr_get_src_op(instr, 0), true);
189     file_print_message(asm_file, S(", "));
190     x86_32_operand_dump(asm_file, aia_instr_get_dest_op(instr), true);
191 }
192
193 static void x86_32_movbl_emit(FILE *asm_file, Aia_Instr *instr,
194     Const_String in_name)
195 {
196     assert(aia_instr_get_dest_op_size(instr) == AIA_LONG &&
197         aia_instr_get_src_ops_size(instr) == AIA_BYTE);
198
199     file_print_message(asm_file, S("\tSbl\t"), in_name);
200     x86_32_operand_dump(asm_file, aia_instr_get_src_op(instr, 0), true);
201     file_print_message(asm_file, S(", "));
202     x86_32_operand_dump(asm_file, aia_instr_get_dest_op(instr), true);
203 }
204
205 static void x86_32_2op_arith_emit(FILE *asm_file, Aia_Instr *instr,
206     Const_String in_name, Aia_Meta_Data *ameta)
207 {
208     assert(aia_instr_get_dest_op_size(instr) ==
209         aia_instr_get_src_ops_size(instr));
210
211     Aia_Operand *dest = aia_instr_get_dest_op(instr);
212     Aia_Operand *src = aia_instr_get_src_op(instr, 1);
213     DEBUGT(def,
214         Aia_Operand *tmp_src = aia_instr_get_src_op(instr, 0);
215         assert(aia_operands_equal(dest, tmp_src));
216     );
217
218     Const_String instr_postfix = x86_32_get_instr_postfix_dest(instr);
219
220     file_print_message(asm_file, S("\t%S\t"), in_name, instr_postfix);
221     x86_32_operand_dump(asm_file, src, true);
222     file_print_message(asm_file, S(", "));
223     x86_32_operand_dump(asm_file, dest, true);
224
225     if (ameta->inside_func && aia_operands_equal(dest, reg_esp)) {
226         assert(aia_operand_get_type(src) == AIA_OPERAND_CONST_INT);
227         switch (aia_instr_get_type(instr)) {
228             case AIA_ADD:
229                 ameta->cfi_offset -= aia_operand_const_int_get_val(src);
230                 break;
231             case AIA_SUB:
232                 ameta->cfi_offset += aia_operand_const_int_get_val(src);
233                 break;
234
235             default:
236                 break;
237         }
238         file_print_message(asm_file, S("\n.cfi_def_cfa_offset %U"),
239             ameta->cfi_offset);
240     }
241 }
242
243 static void x86_32_cmp_emit(FILE *asm_file, Aia_Instr *instr,
244     Const_String in_name)
245 {
246     Const_String instr_postfix = x86_32_get_instr_postfix(instr, false);
247
248     file_print_message(asm_file, S("\t%S\t"), in_name, instr_postfix);

```

```

249     x86_32_operand_dump(asm_file, aia_instr_get_src_op(instr, 1), true);
250     file_print_message(asm_file, S(", "));
251     x86_32_operand_dump(asm_file, aia_instr_get_src_op(instr, 0), true);
252 }
253
254 static void x86_32_set_emit(FILE *asm_file, Aia_Instr *instr,
255     Const_String in_name)
256 {
257     file_print_message(asm_file, S("\t%S\t"), in_name);
258     x86_32_operand_dump(asm_file, aia_instr_get_dest_op(instr), true);
259 }
260
261 static void x86_32_ret_emit(FILE *asm_file, Aia_Instr *instr UNUSED,
262     Const_String in_name)
263 {
264     file_print_message(asm_file, S("\t%S"), in_name);
265 }
266
267 static void x86_32_call_emit(FILE *asm_file, Aia_Instr *instr,
268     Const_String in_name)
269 {
270     file_print_message(asm_file, S("\t%S\t"), in_name);
271     Aia_Operand *call_op = aia_instr_get_src_op(instr, 0);
272     if (aia_operand_get_type(call_op) != AIA_OPERAND_LABEL)
273         file_print_message(asm_file, S("*"));
274     x86_32_operand_dump(asm_file, call_op, true);
275 }
276
277 static void x86_32_neg_emit(FILE *asm_file, Aia_Instr *instr,
278     Const_String in_name)
279 {
280     assert(aia_instr_get_dest_op_size(instr) ==
281         aia_instr_get_src_ops_size(instr));
282     assert(aia_operands_equal(aia_instr_get_dest_op(instr),
283         aia_instr_get_src_op(instr, 0)));
284     Const_String post = x86_32_get_instr_postfix_dest(instr);
285     file_print_message(asm_file, S("\t%S%S\t"), in_name, post);
286     x86_32_operand_dump(asm_file, aia_instr_get_dest_op(instr), true);
287 }
288
289 static void x86_32_cdq_dump(FILE *asm_file, Aia_Instr *instr UNUSED)
290 {
291     assert(aia_operands_equal(aia_instr_get_dest_op(instr), reg_edx));
292     assert(aia_operands_equal(aia_instr_get_src_op(instr, 0), reg_eax));
293     file_print_message(asm_file, S("\tcdq"));
294 }
295
296 static void x86_32_imul_emit(FILE *asm_file, Aia_Instr *instr,
297     Const_String name)
298 {
299     file_print_message(asm_file, S("\t%S1\t"), name);
300     Aia_Operand *op = aia_instr_get_src_op(instr, 1);
301     if (aia_operand_is_integer(op)) {
302         x86_32_operand_dump(asm_file, op, true);
303         file_print_message(asm_file, S(", "));
304
305         op = aia_instr_get_src_op(instr, 0);
306         x86_32_operand_dump(asm_file, op, true);
307         file_print_message(asm_file, S(", "));
308     } else {
309         op = aia_instr_get_src_op(instr, 1);
310         x86_32_operand_dump(asm_file, op, true);
311         file_print_message(asm_file, S(", "));
312     }
313
314     op = aia_instr_get_dest_op(instr);
315     x86_32_operand_dump(asm_file, op, true);
316 }
317
318 static void x86_32_idiv_emit(FILE *asm_file, Aia_Instr *instr,
319     Const_String name)
320 {
321     file_print_message(asm_file, S("\t%S1\t"), name);
322     Aia_Operand *op = aia_instr_get_src_op(instr, 1);

```



```

323     x86_32_operand_dump(asm_file, op, true);
324 }
325
326 static bool x86_32_label_block_stick(Aia_Operand *label_op, Int jmp_op_idx)
327 {
328     Aia_Operand *jmp_op = aia_instr_get_src_op(pending_jmp.instr, jmp_op_idx);
329     if (aia_operand_get_type(jmp_op) == AIA_OPERAND_BLOCK) {
330         Aia_Block *jmp_block = aia_operand_block_get_block(jmp_op);
331         Aia_Instr *lbl_in = __aia_block_peek_first_instr(jmp_block);
332         jmp_op = aia_instr_get_dest_op(lbl_in);
333         assert(jmp_block);
334     }
335     if (aia_operands_equal(label_op, jmp_op))
336         return true;
337     return false;
338 }
339
340 static CONST_STRING(jmp_str, "jmp");
341
342 static CONST_STRING(je_str, "je");
343 static CONST_STRING(jne_str, "jne");
344
345 static CONST_STRING(jge_str, "jge");
346 static CONST_STRING(jl_str, "jl");
347
348 static CONST_STRING(jg_str, "jg");
349 static CONST_STRING(jle_str, "jle");
350
351 static Const_String x86_32_negated_jmp_name(Aia_Instr *jmp_in)
352 {
353     switch (aia_instr_get_type(jmp_in)) {
354     case AIA_JG:
355         return jle_str;
356     case AIA_JLE:
357         return jg_str;
358     case AIA_JNE:
359         return je_str;
360     case AIA_JE:
361         return jne_str;
362     case AIA_JGE:
363         return jl_str;
364     case AIA_JL:
365         return jge_str;
366     default:
367         fatal_error(S("unexpected instruction, expected cond jmp. "
368             "Aborting...\n"));
369     }
370 }
371
372 static void x86_32_instr_pending_jump(FILE *asm_file, Aia_Instr *instr)
373 {
374     if (pending_jmp.instr && aia_instr_get_type(instr) == __AIA_LABEL) {
375         Aia_Operand *lbl = aia_instr_get_dest_op(instr);
376
377         switch (aia_instr_get_type(pending_jmp.instr)) {
378         AIA_CASE_COND_JUMP:
379             if (x86_32_label_block_stick(lbl, 1)) {
380                 pending_jmp.emit_second_jump = false;
381                 break;
382             } else {
383                 pending_jmp.emit_second_jump = true;
384             }
385             if (x86_32_label_block_stick(lbl, 0)) {
386                 pending_jmp.instr_name = x86_32_negated_jmp_name(
387                     pending_jmp.instr);
388                 pending_jmp.emit_second_jump = false;
389                 aia_instr_swap_ops(pending_jmp.instr, 0, 1);
390             }
391             break;
392
393         case AIA_JMP:
394             pending_jmp.emit_second_jump = false;
395             if (x86_32_label_block_stick(lbl, 0)) {
396                 pending_jmp.instr = NULL;

```

```

397         return;
398     }
399     break;
400
401     default:
402         fatal_error(S("unexpected instruction, expected jmp. "
403                     "Aborting...\n"));
404     }
405 }
406
407 }
408 x86_32_pending_jump_emit(asm_file);
409 }
410
411 static void x86_32_emit_integer(FILE *asm_file, Aia_Instr *instr, Aia *aia)
412 {
413     Aia_Operand *op = aia_instr_get_src_op(instr, 0);
414     if (aia_instr_get_src_ops_size(instr) == AIA_BYTE &&
415         aia_operand_get_type(op) == AIA_OPERAND_CONST_INT) {
416         int32_t orig = aia_operand_const_int_get_val(op);
417         int8_t val = orig;
418         if (val != orig) {
419             if (cmdopts.warn_overflow)
420                 report_warning_location(aia_instr_get_location(instr),
421                                         S("constant implicitly truncated to " QFY("%" PRIi8)
422                                           " to fit " QFY("char") " variable\n"), val);
423             Aia_Operand *new_op = aia_operand_const_int_alloc(aia, val);
424             aia_instr_replace_op(instr, 0, new_op);
425         }
426     }
427     aia_instr_integer_dump(asm_file, instr);
428 }
429
430 static void x86_32_emit_main_label(FILE *asm_file, Aia_Instr *instr, Aia *aia)
431 {
432     Aia_Operand *main_op = aia_operand_label_alloc(aia, VITMAIN_STR, 0);
433     Aia_Instr *main_in = aia_instr_alloc_2op(____AIA_LABEL,
434         aia_instr_get_block(instr),
435         aia_instr_get_dest_op_size(instr),
436         aia_instr_get_src_ops_size(instr),
437         aia_instr_get_location(instr));
438
439     aia_instr_set_dest_op(main_in, main_op);
440     aia_instr_set_src_op(main_in, 0, aia_instr_get_src_op(instr, 0));
441     aia_instr_set_src_op(main_in, 1, aia_instr_get_src_op(instr, 1));
442
443     aia_instr_label_dump(asm_file, main_in);
444
445     ____aia_instr_destroy(main_in);
446 }
447
448 static void x86_32_emit_label(FILE *asm_file, Aia_Instr *instr, Aia *aia)
449 {
450     Aia_Operand *data = aia_instr_get_src_op(instr, 0);
451     Aia_Label_Data lbl_data;
452     lbl_data.data = aia_operand_const_int_get_val(data);
453
454     if (lbl_data.linkage == AIA_LINKAGE_GLOBAL) {
455         Aia_Operand *lbl_op = aia_instr_get_dest_op(instr);
456         Const_String lbl_name = aia_operand_label_get_name(lbl_op);
457         if (!string_compare(lbl_name, MAIN_STR)) {
458             x86_32_emit_main_label(asm_file, instr, aia);
459             return;
460         }
461     }
462
463     aia_instr_label_dump(asm_file, instr);
464 }
465
466 static void x86_32_instr_emit(FILE *asm_file, Aia *aia, Aia_Instr *instr)
467 {
468     if (aia_instr_get_type(instr) == AIA_NOP)
469         return;
470

```

```

471     x86_32_instr_pending_jump(asm_file, instr);
472     x86_32_emit_line_number(asm_file, instr);
473
474     Aia_Meta_Data *ameta = aia->meta_data;
475
476     switch (aia_instr_get_type(instr)) {
477     case AIA_MOV:
478         x86_32_mov_emit(asm_file, instr, S("mov"));
479         break;
480     case AIA_MOVS:
481         x86_32_movbl_emit(asm_file, instr, S("movs"));
482         break;
483     case AIA_MOVZ:
484         x86_32_movbl_emit(asm_file, instr, S("movz"));
485         break;
486     case AIA_ADD:
487         x86_32_2op_arith_emit(asm_file, instr, S("add"), ameta);
488         break;
489     case AIA_SUB:
490         x86_32_2op_arith_emit(asm_file, instr, S("sub"), ameta);
491         break;
492     case AIA_IMUL:
493         x86_32_imul_emit(asm_file, instr, S("imul"));
494         break;
495     case AIA_IDIV:
496         x86_32_idiv_emit(asm_file, instr, S("idiv"));
497         break;
498     case AIA_CMP:
499         x86_32_cmp_emit(asm_file, instr, S("cmp"));
500         break;
501     case AIA_SETE:
502         x86_32_set_emit(asm_file, instr, S("sete"));
503         break;
504     case AIA_SETNE:
505         x86_32_set_emit(asm_file, instr, S("setne"));
506         break;
507     case AIA_SETL:
508         x86_32_set_emit(asm_file, instr, S("setl"));
509         break;
510     case AIA_SETG:
511         x86_32_set_emit(asm_file, instr, S("setg"));
512         break;
513     case AIA_SETLE:
514         x86_32_set_emit(asm_file, instr, S("setle"));
515         break;
516     case AIA_SETGE:
517         x86_32_set_emit(asm_file, instr, S("setge"));
518         break;
519     case AIA_RET:
520         x86_32_ret_emit(asm_file, instr, S("ret"));
521         break;
522     case AIA_CALL:
523         x86_32_call_emit(asm_file, instr, S("call"));
524         break;
525     case AIA_NEG:
526         x86_32_neg_emit(asm_file, instr, S("neg"));
527         break;
528     case __AIA_LABEL:
529         x86_32_emit_label(asm_file, instr, aia);
530         break;
531     case __AIA_STRING:
532         aia_instr_string_dump(asm_file, instr);
533         break;
534     case __AIA_INTEGER:
535         x86_32_emit_integer(asm_file, instr, aia);
536         break;
537     case AIA_JLE:
538         x86_32_jmp_emit(asm_file, instr, jle_str);
539         /* Don't print "\n". */
540         return;
541     case AIA_JG:
542         x86_32_jmp_emit(asm_file, instr, jg_str);
543         /* Don't print "\n". */
544         return;

```

```

545     case AIA_JNE:
546         x86_32_jump_emit(asm_file, instr, jne_str);
547         /* Don't print "\n". */
548         return;
549     case AIA_JE:
550         x86_32_jump_emit(asm_file, instr, je_str);
551         /* Don't print "\n". */
552         return;
553     case AIA_JGE:
554         x86_32_jump_emit(asm_file, instr, jge_str);
555         /* Don't print "\n". */
556         return;
557     case AIA_JL:
558         x86_32_jump_emit(asm_file, instr, jl_str);
559         /* Don't print "\n". */
560         return;
561     case AIA_JMP:
562         x86_32_jump_emit(asm_file, instr, jmp_str);
563         /* Don't print "\n". */
564         return;
565     case AIA_CDQ:
566         x86_32_cdq_dump(asm_file, instr);
567         break;
568
569     case __AIA_JNE:
570         /* Fall */
571     case __AIA_JE:
572         /* Fall */
573     case __AIA_JMP:
574         fatal_error(S("Unable to dump unexpected jump instruction. "
575             "Aborting...\n"));
576
577     default:
578         fatal_error(S("Unable to dump unexpected instruction. "
579             "Aborting...\n"));
580 }
581
582 if (!ameta->cfi_offset && ameta->inside_func) {
583     ameta->cfi_offset = 4;
584     file_print_message(asm_file, S("\n.cfi_startproc\n"));
585 } else {
586     file_print_message(asm_file, S("\n"));
587 }
588 }
589
590 static void x86_32_emit_block(FILE *asm_file, Aia_Block *b, Aia *aia)
591 {
592     if (!b->meta_data)
593         return;
594
595     Aia_Instr *instr;
596     AIA_BLOCK_FOR_EACH_INSTRUCTION(b, instr)
597         x86_32_instr_emit(asm_file, aia, instr);
598 }
599
600 static void x86_32_emit_blocks(FILE *asm_file, Aia *aia, Double_List *blist)
601 {
602     Double_List_Node *dbnode;
603     DOUBLE_LIST_FOR_EACH(blist, dbnode)
604         x86_32_emit_block(asm_file, AIA_BLOCK_OF_DBNODE(dbnode), aia);
605 }
606
607 static void x86_32_func_emit(FILE *asm_file, Aia *aia, Aia_Func *func)
608 {
609     Aia_Meta_Data *ameta = aia->meta_data;
610
611     Aia_Func_Trampoline *tramp;
612     AIA_FUNC_FOR_EACH_TRAMPOLINE(func, tramp) {
613         x86_32_emit_prev_line = 0;
614         ameta->cfi_offset = 0;
615         ameta->inside_func = true;
616
617         x86_32_emit_blocks(asm_file, aia, &tramp->blist);

```

```

619     //x86_32_pending_jump_emit(asm_file);
620     file_print_message(asm_file, S(".cfi_endproc\n"));
621     file_print_message(asm_file, S(".size %1$S, . - %1$S\n"),
622         tramp->trampoline_name);
623 }
624
625 x86_32_emit_prev_line = 0;
626 ameta->cfi_offset = 0;
627 ameta->inside_func = true;
628
629 x86_32_emit_blocks(asm_file, aia, &func->blist);
630 x86_32_pending_jump_emit(asm_file);
631 file_print_message(asm_file, S(".cfi_endproc\n"));
632
633 Const_String fname;
634 if (string_compare(func->func_name, MAIN_STR))
635     fname = func->func_name;
636 else
637     fname = VITMAIN_STR;
638 file_print_message(asm_file, S(".size %1$S, . - %1$S\n"), fname);
639 }
640
641 static void x86_32_section_emit(FILE *asm_file, Aia *aia, Aia_Section *sec)
642 {
643     x86_32_emit_prev_line = 0;
644
645     Aia_Meta_Data *ameta = aia->meta_data;
646     ameta->cfi_offset = 0;
647     ameta->inside_func = false;
648
649     file_print_message(asm_file, S(".section %S\n"),
650         aia_section_get_name(sec));
651
652     x86_32_emit_blocks(asm_file, aia, &sec->sec_blist);
653
654     bool any_func = false;
655     Aia_Func *func;
656     AIA_SECTION_FOR_EACH_FUNC(sec, func) {
657         x86_32_func_emit(asm_file, aia, func);
658         any_func = true;
659     }
660     x86_32_pending_jump_emit(asm_file);
661     if (!any_func)
662         file_print_message(asm_file, S("\n"));
663 }
664
665 static String x86_32_get_file_name(Aia *aia)
666 {
667     if (cmdopts.dump_asm)
668         return string_from_format(S("%S.vitaly" ASM1_SUFFIX_CSTR),
669             aia_get_file_name(aia));
670     return string_to_tmp_file(ASM1_SUFFIX_STR);
671 }
672
673 static FILE *x86_32_get_file(Const_String fname)
674 {
675     FILE *f = file_open(fname, S("w"));
676     if (!f)
677         fatal_error(S("unable to create file %S for assembly code [%m]\n"),
678             fname);
679     return f;
680 }
681
682 static void x86_32_emit_sections(Const_String asm_fname, Aia *aia)
683 {
684     FILE *asm_file = x86_32_get_file(asm_fname);
685     Aia_Section *sec;
686
687     Aia_Meta_Data ameta = { 0, false };
688     aia->meta_data = &ameta;
689
690     AIA_FOR_EACH_SECTION(aia, sec)
691         x86_32_section_emit(asm_file, aia, sec);
692     file_close(asm_file);

```

```

693 }
694
695 static void x86_32_reset_blist(Double_List *blist)
696 {
697     Double_List_Node *bnode;
698     DOUBLE_LIST_FOR_EACH(blist, bnode)
699         AIA_BLOCK_OF_DBNODE(bnode)->meta_data = NULL;
700 }
701
702 static void x86_32_emit_reset(Aia *aia)
703 {
704     Aia_Section *sec;
705     AIA_FOR_EACH_SECTION(aia, sec) {
706         x86_32_reset_blist(&sec->sec_blist);
707         Aia_Func *f;
708         AIA_SECTION_FOR_EACH_FUNC(sec, f) {
709             x86_32_reset_blist(&f->blist);
710             Aia_Func_Trampoline *t;
711             AIA_FUNC_FOR_EACH_TRAMPOLINE(f, t)
712                 x86_32_reset_blist(&t->blist);
713         }
714     }
715 }
716
717 static void x86_32_emit_mark_live_block(Aia_Block *b, void *arg UNUSED)
718 {
719     b->meta_data = INT_TO_PTR(1);
720 }
721
722 static void x86_32_emit_mark_live(Aia *aia)
723 {
724     Aia_Section *sec;
725     AIA_FOR_EACH_SECTION(aia, sec) {
726         aia_section_for_each_block_depth(sec,
727             x86_32_emit_mark_live_block, NULL);
728         Aia_Func *f;
729         AIA_SECTION_FOR_EACH_FUNC(sec, f) {
730             aia_func_for_each_block_depth(f,
731                 x86_32_emit_mark_live_block, NULL);
732             Aia_Func_Trampoline *t;
733             AIA_FUNC_FOR_EACH_TRAMPOLINE(f, t)
734                 x86_32_emit_mark_live_block(t->block, NULL);
735         }
736     }
737 }
738
739 String x86_32_emit(Aia *aia)
740 {
741     String fname = NULL;
742     if (!aia_is_valid(aia))
743         goto out;
744
745     x86_32_emit_reset(aia);
746     x86_32_emit_mark_live(aia);
747
748     fname = x86_32_get_file_name(aia);
749     x86_32_emit_sections(fname, aia);
750
751     if (cmdopts.assemble_only) {
752         if (cmdopts.output_name) {
753             x86_32_emit_sections(cmdopts.output_name, aia);
754         } else {
755             String tmp = string_cpy_replace_from(aia_get_file_name(aia), '.', ' ',
756                 ASM1_SUFFIX_STR);
757             x86_32_emit_sections(tmp, aia);
758             string_destroy(tmp);
759         }
760     }
761
762 out:
763     return fname;
764 }

```

:

A.9.10 src/x86_32/x86_32_emit.h

```

1  #ifndef X86_32_EMIT_H
2  #define X86_32_EMIT_H
3
4  #include <aia/aia.h>
5
6  String x86_32_emit(Aia *aia);
7
8  #endif // X86_32_EMIT_H

```

:

A.9.11 src/x86_32/x86_32_func_normalize.c

```

1  #include "x86_32_regs.h"
2  #include "main.h"
3
4  #undef DEBUG_TYPE
5  #define DEBUG_TYPE func-norm
6
7  typedef struct Aia_Func_Meta {
8      Aia_Instr *prologue_in;
9      Aia_Instr *epilogue_in;
10     Aia_Operand *ret_label;
11     Hash_Map func_local_offsets;
12     Int max_arg_count;
13     Int esp_delta_bytes;
14     #ifndef NDEBUG
15         Int prev_arg_idx;
16     #endif
17     bool preserve_reg[X86_32_REG_COUNT];
18     bool uses_display;
19 } Aia_Func_Meta;
20
21 typedef struct Func_Local_Off {
22     Const_String name;
23     Int byte_offset;
24     Hash_Map_Slot hash_slot;
25 } Func_Local_Off;
26
27 #define FUNC_LOCAL_OFF_OF(slot) HASH_MAP_ENTRY(slot, Func_Local_Off, hash_slot)
28
29 typedef struct Aia_Block_Meta {
30     Aia_Func *curr_func;
31     Aia *aia;
32     Aia_Func *called_func;
33     Vector func_ret_instrs;
34     Hash_Map *curr_func_locals;
35     Int esp_byte_increment;
36 } Aia_Block_Meta;
37
38 static bool func_local_hash_compare(String search_str, Hash_Map_Slot *s)
39 {
40     Func_Local_Off *off = FUNC_LOCAL_OFF_OF(s);
41     return !string_compare(off->name, search_str);
42 }
43
44 static void func_local_hash_destroy(Hash_Map_Slot *s)
45 {
46     free_mem(FUNC_LOCAL_OFF_OF(s));
47 }
48
49 static inline void func_local_insert(Hash_Map *local_offsets,
50     Const_String local_name, Int byte_off)
51 {

```

```

52     Func_Local_Off *loc = ALLOC_NEW(Func_Local_Off);
53     loc->name = local_name;
54     loc->byte_offset = byte_off;
55     Uns hash = string_hash_code(local_name);
56     assert(!hash_map_contains(local_offsets,
57         (String)local_name, hash));
58     hash_map_insert(local_offsets, &loc->hash_slot, hash);
59 }
60
61 static Int func_local_get_offset(Hash_Map *local_offsets, Const_String local)
62 {
63     Hash_Map_Slot *s = hash_map_get(local_offsets, (String)local,
64         string_hash_code(local));
65     assert(s);
66     return FUNC_LOCAL_OFF_OF(s)->byte_offset;
67 }
68
69 static Aia_Operand *x86_32_alloc_ref(Aia *aia, Aia_Operand *base, int32_t disp)
70 {
71     Aia_Operand *disp_op;
72     if (disp)
73         disp_op = aia_operand_const_int_alloc(aia, disp);
74     else
75         disp_op = NULL;
76     Aia_Operand *ret = aia_operand_addr_ref_alloc(aia,
77         NULL, disp_op, base, NULL, NULL);
78     return ret;
79 }
80
81 static void x86_32_setup_call_esp_increment(Aia_Instr *call_in,
82     Aia_Block_Meta *bmeta)
83 {
84     Aia_Operand *lbl = aia_instr_get_src_op(call_in, 0);
85     if (aia_operand_get_type(lbl) != AIA_OPERAND_LABEL)
86         goto out_no_inc;
87
88     Const_String name = aia_operand_label_get_name(lbl);
89     bmeta->called_func = aia_func_lookup(bmeta->aia, name);
90     //assert(bmeta->called_func);
91     if (!bmeta->called_func) // then it's an imported function.
92         goto out_no_inc;
93
94     if (bmeta->curr_func && aia_func_is_nested(bmeta->called_func) &&
95         ((Aia_Func_Meta *)bmeta->called_func->meta_data)->uses_display) {
96         Aia_Func_Meta *fmeta = bmeta->curr_func->meta_data;
97
98         bmeta->esp_byte_increment = 4 * (fmeta->max_arg_count -
99             vector_size(&bmeta->called_func->parameters) +
100             aia_func_get_num_display_params(bmeta->called_func));
101     } else {
102         goto out_no_inc;
103     }
104
105     assert(bmeta->esp_byte_increment >= 0);
106     return;
107
108 out_no_inc:
109     bmeta->esp_byte_increment = 0;
110 }
111
112 static void x86_32_setup_called_func(Aia_Instr *in, Aia_Block_Meta *bmeta)
113 {
114     while (aia_instr_get_type(in) != AIA_CALL) {
115         Aia_Instr *tmp = aia_instr_get_sucessor(in);
116         if (!tmp) {
117             Aia_Block *b = aia_instr_get_block(in);
118             Aia_Block *suc = NULL;
119             AIA_BLOCK_FOR_EACH_SUCESSOR(b, suc)
120                 break;
121
122             assert(suc);
123             in = __aia_block_peek_first_instr(suc);
124             assert(in);
125         } else {

```



```

126         in = tmp;
127     }
128 }
129 x86_32_setup_call_esp_increment(in, bmeta);
130 }
131
132 static void x86_32_instr_op_finalize(Aia_Instr *in, Int op_idx,
133     Aia_Block_Meta *bmeta)
134 {
135     Aia_Func_Meta *fmeta;
136     Aia_Operand *op = aia_instr_get_op(in, op_idx);
137     Aia_Operand *ref;
138     Const_String oth_func_name;
139     Int offset;
140     switch (aia_operand_get_type(op)) {
141     case AIA_OPERAND_LOCAL_REF:
142         offset = func_local_get_offset(bmeta->curr_func_locals,
143             aia_operand_local_ref_get_var_name(op));
144         ref = x86_32_alloc_ref(bmeta->aia, reg_esp, offset);
145         break;
146
147     case AIA_OPERAND_DISPLAY_REF:
148         oth_func_name = aia_operand_display_ref_get_func_name(op);
149         Aia_Func *oth = aia_func_lookup(bmeta->aia, oth_func_name);
150         assert(oth);
151         fmeta = oth->meta_data;
152         offset = func_local_get_offset(&fmeta->func_local_offsets,
153             aia_operand_display_ref_get_var_name(op));
154         ref = x86_32_alloc_ref(bmeta->aia,
155             aia_operand_display_ref_get_display_reg(op), offset);
156         break;
157
158     case AIA_OPERAND_ARG:
159
160         if (!bmeta->called_func)
161             x86_32_setup_called_func(in, bmeta);
162
163         offset = 4 * aia_operand_arg_get_idx(op) + bmeta->esp_byte_increment;
164         ref = x86_32_alloc_ref(bmeta->aia, reg_esp, offset);
165         break;
166
167     default:
168         return;
169     }
170     aia_instr_replace_op(in, op_idx, ref);
171 }
172
173 static void x86_32_call_instr_finalize(Aia_Instr *in, Aia_Block_Meta *bmeta)
174 {
175     Aia_Operand *add_const;
176     Aia_Instr *add_in;
177
178     if (!bmeta->called_func)
179         x86_32_setup_call_esp_increment(in, bmeta);
180
181     if (bmeta->esp_byte_increment) {
182         add_const = aia_operand_const_int_alloc(bmeta->aia,
183             bmeta->esp_byte_increment);
184         add_in = aia_instr_alloc_2op(AIA_ADD,
185             aia_instr_get_block(in), AIA_LONG, AIA_LONG,
186             aia_instr_get_location(in));
187         aia_instr_set_dest_op(add_in, reg_esp);
188         aia_instr_set_src_op(add_in, 0, reg_esp);
189         aia_instr_set_src_op(add_in, 1, add_const);
190
191         aia_instr_insert_before(add_in, in);
192
193         add_const = aia_operand_const_int_alloc(bmeta->aia,
194             -bmeta->esp_byte_increment);
195         add_in = aia_instr_alloc_2op(AIA_ADD,
196             aia_instr_get_block(in), AIA_LONG, AIA_LONG,
197             aia_instr_get_location(in));
198         aia_instr_set_dest_op(add_in, reg_esp);
199         aia_instr_set_src_op(add_in, 0, reg_esp);

```

```

200     aia_instr_set_src_op(add_in, 1, add_const);
201
202     aia_instr_insert_after(add_in, in);
203 }
204
205 bmeta->called_func = NULL;
206 }
207
208 static void x86_32_block_finalize(Aia_Block *b, Aia_Block_Meta *bmeta)
209 {
210     Aia_Instr *in;
211     AIA_BLOCK_FOR_EACH_INSTRUCTION(b, in) {
212         Aia_Operand *op;
213         Int op_idx = -1;
214         AIA_INSTR_FOR_EACH_OPERAND(in, op) {
215             if (op)
216                 x86_32_instr_op_finalize(in, op_idx, bmeta);
217             ++op_idx;
218         }
219         switch (aia_instr_get_type(in)) {
220             case AIA_CALL:
221                 x86_32_call_instr_finalize(in, bmeta);
222                 break;
223
224             case AIA_RET:
225                 vector_append(&bmeta->func_ret_instrs, in);
226                 break;
227
228             default:
229                 break;
230         }
231     }
232 }
233
234 static void ____x86_32_section_normalize(Aia *aia, Aia_Section *sec)
235 {
236     Aia_Block_Meta bmeta;
237     bmeta.called_func = NULL;
238     bmeta.curr_func = NULL;
239     bmeta.curr_func_locals = NULL;
240     bmeta.esp_byte_increment = 0;
241     bmeta.aia = aia;
242     aia_section_for_each_block_depth(sec,
243         (Aia_Block_Callback)x86_32_block_finalize, &bmeta);
244 }
245
246 static void set_preserve_regs_from_op(Aia_Operand *op,
247     bool preserve_reg[X86_32_REG_COUNT]);
248
249 static void ____x86_32_trampoline_normalize(Aia *aia,
250     Aia_Func_Trampoline *tramp)
251 {
252     HASH_MAP_SIZE(tramp_locals, HASH_MAP_SIZE_11,
253         (Hash_Map_Comparator)func_local_hash_compare);
254
255     bool preserve_reg[X86_32_REG_COUNT];
256     for (int i = 0; i < X86_32_REG_COUNT; i++)
257         preserve_reg[i] = false;
258
259     Aia_Instr *in;
260     AIA_BLOCK_FOR_EACH_INSTRUCTION(tramp->block, in) {
261         Aia_Operand *op;
262         AIA_INSTR_FOR_EACH_OPERAND(in, op) {
263             if (op)
264                 set_preserve_regs_from_op(op, preserve_reg);
265         }
266     }
267
268     int32_t esp_dec = 0;
269     for (int i = 0; i < X86_32_REG_COUNT; i++) {
270         if (preserve_reg[i])
271             esp_dec += 4;
272     }
273 }

```

```

274     int32_t off = 4 + esp_dec;
275     Const_String loc;
276     VECTOR_FOR_EACH_ENTRY(tramp->func_params, loc) {
277         func_local_insert(&tramp_locals, loc, off);
278         off += 4;
279     }
280
281     Aia_Block_Meta bmeta;
282     bmeta.called_func = NULL;
283     bmeta.curr_func = NULL;
284     bmeta.curr_func_locals = &tramp_locals;
285     bmeta.esp_byte_increment = 0;
286     bmeta.aia = aia;
287     x86_32_block_finalize(tramp->block, &bmeta);
288
289     if (esp_dec) {
290         Aia_Instr *first_in = __aia_block_peek_first_instr(tramp->block);
291         Aia_Instr *last_in = __aia_block_peek_last_instr(tramp->block);
292
293         Aia_Instr *tmp = aia_instr_alloc_2op(AIA_ADD, tramp->block,
294             AIA_LONG, AIA_LONG, aia_instr_get_location(first_in));
295         Aia_Operand *op = aia_operand_const_int_alloc(aia, -esp_dec);
296         aia_instr_set_dest_op(tmp, reg_esp);
297         aia_instr_set_src_op(tmp, 0, reg_esp);
298         aia_instr_set_src_op(tmp, 1, op);
299         aia_instr_insert_after(tmp, first_in);
300
301         off = 0;
302         for (Int i = 0; i < X86_32_REG_COUNT; i++) {
303             if (preserve_reg[i]) {
304                 Const_String reg_str = x86_32_reg_idx_get_32bit_name(i);
305                 Aia_Operand *reg = x86_32_get_reg_operand(reg_str);
306                 Aia_Operand *disp;
307                 if (off) {
308                     disp = aia_operand_const_int_alloc(aia, off);
309                 } else {
310                     disp = NULL;
311                 }
312                 Aia_Operand *esp_addr = aia_operand_addr_ref_alloc(aia,
313                     NULL, disp, reg_esp, NULL, NULL);
314                 aia_mov_after(reg, esp_addr, tmp, AIA_LONG);
315
316                 off += 4;
317             }
318         }
319
320         tmp = aia_instr_alloc_2op(AIA_ADD, tramp->block, AIA_LONG, AIA_LONG,
321             aia_instr_get_location(last_in));
322         op = aia_operand_const_int_alloc(aia, esp_dec);
323         aia_instr_set_dest_op(tmp, reg_esp);
324         aia_instr_set_src_op(tmp, 0, reg_esp);
325         aia_instr_set_src_op(tmp, 1, op);
326         aia_instr_insert_before(tmp, last_in);
327
328         off = 0;
329         for (Int i = 0; i < X86_32_REG_COUNT; i++) {
330             if (preserve_reg[i]) {
331                 Const_String reg_str = x86_32_reg_idx_get_32bit_name(i);
332                 Aia_Operand *reg = x86_32_get_reg_operand(reg_str);
333                 Aia_Operand *disp;
334                 if (off) {
335                     disp = aia_operand_const_int_alloc(aia, off);
336                 } else {
337                     disp = NULL;
338                 }
339                 Aia_Operand *esp_addr = aia_operand_addr_ref_alloc(aia,
340                     NULL, disp, reg_esp, NULL, NULL);
341                 aia_mov_before(esp_addr, reg, tmp, AIA_LONG);
342
343                 off += 4;
344             }
345         }
346     }
347

```

```

348     hash_map_for_each_destroy(&tramp_locals,
349         func_local_hash_destroy);
350 }
351
352
353 static void ____x86_32_func_normalize(Aia *aia, Aia_Func *func)
354 {
355     Aia_Func_Meta *fmeta = func->meta_data;
356
357     Aia_Block_Meta bmeta;
358     bmeta.called_func = NULL;
359     bmeta.curr_func = func;
360     bmeta.curr_func_locals = &fmeta->func_local_offsets;
361     bmeta.esp_byte_increment = 0;
362     bmeta.func_ret_instrs = VECTOR_INIT_SIZE(4);
363     bmeta.aia = aia;
364     aia_func_for_each_block_depth(func,
365         (Aia_Block_Callback)x86_32_block_finalize, &bmeta);
366
367     Aia_Instr *last_ret = ____aia_block_peek_last_instr(func->exit_block);
368     assert(aia_instr_get_type(last_ret) == AIA_RET);
369
370
371     Aia_Instr *jmp_in;
372     Aia_Instr *ret_in;
373     VECTOR_FOR_EACH_ENTRY(&bmeta.func_ret_instrs, ret_in) {
374         if (ret_in != last_ret) {
375             jmp_in = aia_instr_alloc_lop(AIA_JMP,
376                 aia_instr_get_block(ret_in), -1, AIA_LONG,
377                 aia_instr_get_location(ret_in));
378             aia_instr_set_src_op(jmp_in, 0, fmeta->ret_label);
379             aia_instr_replace_destroy(ret_in, jmp_in);
380         }
381     }
382
383     vector_clear(&bmeta.func_ret_instrs);
384 }
385
386 static UNUSED void ____x86_32_trampoline_verify(Aia *aia UNUSED,
387     Aia_Func_Trampoline *tramp UNUSED)
388 {
389     return;
390 #if 0
391     Aia_Instr *in;
392     AIA_BLOCK_FOR_EACH_INSTRUCTION(tramp->block, in) {
393         Aia_Operand *op;
394         AIA_INSTR_FOR_EACH_OPERAND(in, op) {
395             if (!op)
396                 continue;
397
398             switch (aia_operand_get_type(op)) {
399             case AIA_OPERAND_REG:
400                 assert(!x86_32_is_callee_save_reg(op));
401                 break;
402
403             case AIA_OPERAND_DISPLAY_REF:
404                 assert(!x86_32_is_callee_save_reg(
405                     aia_operand_display_ref_get_display_reg(op)));
406                 break;
407
408             case AIA_OPERAND_ADDR_REF:
409                 if (aia_operand_addr_ref_get_base(op))
410                     assert(!x86_32_is_callee_save_reg(
411                         aia_operand_addr_ref_get_base(op)));
412                 if (aia_operand_addr_ref_get_index(op))
413                     assert(!x86_32_is_callee_save_reg(
414                         aia_operand_addr_ref_get_index(op)));
415                 break;
416
417             default:
418                 break;
419             }
420         }
421     }

```

```

422 #endif
423 }
424
425 static void set_preserve_regs_from_op(Aia_Operand *op,
426     bool preserve_reg[X86_32_REG_COUNT])
427 {
428     Aia_Operand *tmp;
429
430     switch (aia_operand_get_type(op)) {
431     case AIA_OPERAND_REG:
432         if (x86_32_is_callee_save_reg(op))
433             preserve_reg[x86_32_reg_get_idx(op)] = true;
434         break;
435
436     case AIA_OPERAND_DISPLAY_REF:
437         tmp = aia_operand_display_ref_get_display_reg(op);
438         if (x86_32_is_callee_save_reg(tmp))
439             preserve_reg[x86_32_reg_get_idx(tmp)] = true;
440         break;
441
442     case AIA_OPERAND_ADDR_REF:
443         if (aia_operand_addr_ref_get_base(op)) {
444             tmp = aia_operand_addr_ref_get_base(op);
445             if (x86_32_is_callee_save_reg(tmp))
446                 preserve_reg[x86_32_reg_get_idx(tmp)] = true;
447         }
448         if (aia_operand_addr_ref_get_index(op)) {
449             tmp = aia_operand_addr_ref_get_index(op);
450             if (x86_32_is_callee_save_reg(tmp))
451                 preserve_reg[x86_32_reg_get_idx(tmp)] = true;
452         }
453         break;
454
455     default:
456         break;
457     }
458 }
459
460 static void x86_32_func_block_init(Aia_Block *b, Aia_Func *func)
461 {
462     Aia_Instr *in;
463     Aia_Func_Meta *fmeta = func->meta_data;
464     AIA_BLOCK_FOR_EACH_INSTRUCTION(b, in) {
465         Aia_Operand *op;
466
467         DEBUGT(def,
468             if (aia_instr_get_type(in) == AIA_CALL)
469                 fmeta->prev_arg_idx = -1;
470         );
471
472         AIA_INSTR_FOR_EACH_OPERAND(in, op) {
473             if (!op)
474                 continue;
475
476             set_preserve_regs_from_op(op, fmeta->preserve_reg);
477
478             if (aia_operand_get_type(op) == AIA_OPERAND_DISPLAY_REF)
479                 fmeta->uses_display = true;
480
481             if (aia_operand_get_type(op) == AIA_OPERAND_ARG) {
482                 if (aia_operand_arg_get_idx(op) >= fmeta->max_arg_count)
483                     fmeta->max_arg_count = aia_operand_arg_get_idx(op) + 1;
484
485                 DEBUGT(def,
486                     assert(fmeta->prev_arg_idx < aia_operand_arg_get_idx(op));
487                     fmeta->prev_arg_idx = aia_operand_arg_get_idx(op);
488                 );
489             }
490         }
491     }
492
493     if (!fmeta->uses_display && vector_size(&func->preserve_display_indices)) {
494         if (vector_size(&func->preserve_display_indices) > 1 ||
495             vector_get(&func->preserve_display_indices, 0) !=

```

```

496         INT_TO_PTR(-1))
497         fmeta->uses_display = true;
498     }
499 }
500
501 static inline void x86_32_func_add_reg_locals(Aia_Func *func)
502 {
503     Aia_Func_Meta *fmeta = func->meta_data;
504     for (Int i = 0; i < X86_32_REG_COUNT; i++) {
505         if (fmeta->preserve_reg[i]) {
506             Const_String reg = x86_32_reg_idx_get_32bit_name(i);
507             aia_func_append_local(func, reg);
508         }
509     }
510 }
511
512 static void x86_32_func_insert_ret(Aia_Func *func, Aia *aia)
513 {
514     Aia_Func_Meta *fmeta = func->meta_data;
515     if (aia_instr_get_type(fmeta->epilogue_in) != AIA_RET) {
516         Aia_Instr *ret = aia_instr_alloc_0op(AIA_RET,
517             aia_instr_get_block(fmeta->epilogue_in), -1,
518             aia_func_get_last_location(func));
519         aia_instr_insert_after(ret, fmeta->epilogue_in);
520         fmeta->epilogue_in = ret;
521     }
522     String tmp_ret = aia_tmp_name_gen(aia, S("RET"));
523     fmeta->ret_label = __aia_operand_label_alloc(aia, tmp_ret,
524         0, AIA_OPERAND_LABEL);
525     aia_private_label_before(aia, fmeta->ret_label, fmeta->epilogue_in);
526 }
527
528 static void x86_32_func_esp_setup(Aia_Func *func, Aia *aia)
529 {
530     Aia_Func_Meta *fmeta = func->meta_data;
531     if (!fmeta->esp_delta_bytes)
532         return;
533
534     Aia_Operand *val = aia_operand_const_int_alloc(aia,
535         -fmeta->esp_delta_bytes);
536     Aia_Instr *add_in = aia_instr_alloc_2op(AIA_ADD,
537         aia_instr_get_block(fmeta->prologue_in),
538         AIA_LONG, AIA_LONG,
539         aia_func_get_location(func));
540
541     aia_instr_set_dest_op(add_in, reg_esp);
542     aia_instr_set_src_op(add_in, 0, reg_esp);
543     aia_instr_set_src_op(add_in, 1, val);
544     aia_instr_insert_after(add_in, fmeta->prologue_in);
545     fmeta->prologue_in = add_in;
546
547     val = aia_operand_const_int_alloc(aia,
548         fmeta->esp_delta_bytes);
549     add_in = aia_instr_alloc_2op(AIA_ADD,
550         aia_instr_get_block(fmeta->epilogue_in),
551         AIA_LONG, AIA_LONG,
552         aia_func_get_location(func));
553
554     aia_instr_set_dest_op(add_in, reg_esp);
555     aia_instr_set_src_op(add_in, 0, reg_esp);
556     aia_instr_set_src_op(add_in, 1, val);
557     aia_instr_insert_before(add_in, fmeta->epilogue_in);
558     fmeta->epilogue_in = add_in;
559 }
560
561 static void x86_32_func_regs_setup(Aia_Func *func, Aia *aia)
562 {
563     Aia_Func_Meta *fmeta = func->meta_data;
564     for (Int i = 0; i < X86_32_REG_COUNT; i++) {
565         if (fmeta->preserve_reg[i]) {
566             Const_String reg_str = x86_32_reg_idx_get_32bit_name(i);
567             Aia_Operand *local =
568                 aia_operand_local_ref_alloc(aia, reg_str /*, AIA_LONG */);
569             Aia_Operand *reg = x86_32_get_reg_operand(reg_str);

```

```

570         fmeta->prologue_in = aia_mov_after(reg, local, fmeta->prologue_in,
571         AIA_LONG);
572         fmeta->epilogue_in = aia_mov_before(local, reg,
573         fmeta->epilogue_in, AIA_LONG);
574     }
575 }
576 }
577
578 static void x86_32_func_display_setup(Aia_Func *func, Aia *aia)
579 {
580     Aia_Func_Meta *fmeta = func->meta_data;
581
582     Int param_esp_off = fmeta->esp_delta_bytes +
583     vector_size(&func->parameters) * 4;
584
585     Int arg_esp_off = 4 * (fmeta->max_arg_count +
586     vector_size(&func->preserve_display_indices)) - 4;
587
588     ssize_t idx UNUSED;
589     VECTOR_FOR_EACH_ENTRY(&func->preserve_display_indices, idx) {
590         Aia_Operand *ref;
591         Aia_Operand *src;
592
593         if (idx != -1) {
594             ref = x86_32_alloc_ref(aia, reg_esp, param_esp_off);
595             fmeta->prologue_in = aia_mov_after(ref, reg_eax,
596             fmeta->prologue_in, AIA_LONG);
597             src = reg_eax;
598             param_esp_off -= 4;
599         } else {
600             src = reg_esp;
601         }
602
603         ref = x86_32_alloc_ref(aia, reg_esp, arg_esp_off);
604         fmeta->prologue_in = aia_mov_after(src, ref,
605         fmeta->prologue_in, AIA_LONG);
606
607         arg_esp_off -= 4;
608     }
609 }
610
611 static void x86_32_func_insert_logue(Aia_Func *func, Aia *aia)
612 {
613     Aia_Func_Meta *fmeta = func->meta_data;
614     x86_32_func_add_reg_locals(func);
615     DLOG("logue for func: %S\n", func->func_name);
616     DLOG("\tnum locals: %D\n", vector_size(&func->locals));
617     DLOG("\tnum preserves: %D\n", vector_size(&func->preserve_display_indices));
618     DLOG("\tmax arg count: %D\n", fmeta->max_arg_count);
619     fmeta->esp_delta_bytes = 4 * (vector_size(&func->locals) +
620     vector_size(&func->preserve_display_indices) + fmeta->max_arg_count);
621
622     fmeta->prologue_in = __aia_block_peek_first_instr(func->entry_block);
623     fmeta->epilogue_in = __aia_block_peek_last_instr(func->exit_block);
624     assert(fmeta->prologue_in);
625     assert(fmeta->epilogue_in);
626     assert(aia_instr_get_type(fmeta->prologue_in) == __AIA_LABEL);
627
628     x86_32_func_insert_ret(func, aia);
629     x86_32_func_esp_setup(func, aia);
630     x86_32_func_regs_setup(func, aia);
631     x86_32_func_display_setup(func, aia);
632
633     Int off = fmeta->esp_delta_bytes - 4;
634     Const_String loc;
635     VECTOR_FOR_EACH_ENTRY(&func->locals, loc) {
636         DLOG("\toffset %D: %S\n", off, loc);
637         func_local_insert(&fmeta->func_local_offsets, loc, off);
638         off -= 4;
639     }
640
641     off = fmeta->esp_delta_bytes + 4;
642     VECTOR_FOR_EACH_ENTRY(&func->parameters, loc) {
643         DLOG("\toffset %D: %S\n", off, loc);

```

```

644     func_local_insert(&fmeta->func_local_offsets, loc, off);
645     off += 4;
646 }
647 }
648
649 static void x86_32_section_set_max_arg_count(Aia_Block *b, Aia_Section *sec)
650 {
651     Aia_Instr *in;
652     Int max = PTR_TO_INT(sec->meta_data);
653     AIA_BLOCK_FOR_EACH_INSTRUCTION(b, in) {
654         Aia_Operand *op;
655         AIA_INSTR_FOR_EACH_OPERAND(in, op) {
656             if (op && aia_operand_get_type(op) == AIA_OPERAND_ARG) {
657                 if (aia_operand_arg_get_idx(op) >= max)
658                     max = aia_operand_arg_get_idx(op) + 1;
659             }
660         }
661     }
662     sec->meta_data = INT_TO_PTR(max);
663 }
664
665 static void x86_32_section_esp_setup(Aia_Section *sec, Aia *aia)
666 {
667     Int count = PTR_TO_INT(sec->meta_data) * 4;
668     if (!count)
669         return;
670
671     Aia_Instr *first_in = __aia_block_peek_first_instr(sec->entry_block);
672     assert(first_in);
673     assert(aia_instr_get_type(first_in) == __AIA_LABEL);
674     Aia_Instr *last_in = __aia_block_peek_last_instr(sec->exit_block);
675     assert(last_in);
676
677     Aia_Operand *val = aia_operand_const_int_alloc(aia, -count);
678     Aia_Instr *add_in = aia_instr_alloc_2op(AIA_ADD,
679         aia_instr_get_block(first_in),
680         AIA_LONG, AIA_LONG,
681         aia_instr_get_location(first_in));
682     aia_instr_set_dest_op(add_in, reg_esp);
683     aia_instr_set_src_op(add_in, 0, reg_esp);
684     aia_instr_set_src_op(add_in, 1, val);
685     aia_instr_insert_after(add_in, first_in);
686
687     val = aia_operand_const_int_alloc(aia, count);
688     add_in = aia_instr_alloc_2op(AIA_ADD,
689         aia_instr_get_block(last_in),
690         AIA_LONG, AIA_LONG,
691         aia_instr_get_location(last_in));
692     aia_instr_set_dest_op(add_in, reg_esp);
693     aia_instr_set_src_op(add_in, 0, reg_esp);
694     aia_instr_set_src_op(add_in, 1, val);
695     aia_instr_insert_after(add_in, last_in);
696 }
697
698 static void aia_section_insert_logue(Aia_Section *sec, Aia *aia)
699 {
700     x86_32_section_esp_setup(sec, aia);
701 }
702
703 static void x86_32_dump_final_ic(Aia *aia)
704 {
705     String fname = string_from_format(S("%S.vitaly.final-x86-32-ic"),
706         aia_get_file_name(aia));
707     FILE *f = file_open(fname, S("w"));
708     if (!f)
709         fatal_error(S("unable to open file %S for intermediate code dump\n"),
710             fname);
711     string_destroy(fname);
712     aia_dump(aia, f);
713     file_close(f);
714 }
715
716 void x86_32_func_normalize(Aia *aia)
717 {

```



```

718     Aia_Section *sec;
719     Aia_Func *func;
720     Aia_Func_Trampoline *tramp;
721     Aia_Func_Meta *fmeta;
722
723     AIA_FOR_EACH_SECTION(aia, sec) {
724         sec->meta_data = NULL;
725         aia_section_for_each_block_depth(sec, (Aia_Block_Callback)
726             x86_32_section_set_max_arg_count, sec);
727         aia_section_insert_logue(sec, aia);
728
729         AIA_SECTION_FOR_EACH_FUNC(sec, func) {
730             func->meta_data = fmeta = alloc_zeros(sizeof(Aia_Func_Meta));
731             fmeta->uses_display = false;
732             fmeta->func_local_offsets = HASH_MAP_INIT_SIZE(HASH_MAP_SIZE_11,
733                 (Hash_Map_Comparator) func_local_hash_compare);
734
735             DEBUGT(def, fmeta->prev_arg_idx = -1);
736             aia_func_for_each_block_depth(func, (Aia_Block_Callback)
737                 x86_32_func_block_init, func);
738
739             x86_32_func_insert_logue(func, aia);
740         }
741     }
742
743     AIA_FOR_EACH_SECTION(aia, sec) {
744         __x86_32_section_normalize(aia, sec);
745
746         AIA_SECTION_FOR_EACH_FUNC(sec, func) {
747             __x86_32_func_normalize(aia, func);
748
749             AIA_FUNC_FOR_EACH_TRAMPOLINE(func, tramp) {
750                 DEBUGT(def, __x86_32_trampoline_reg_verify(aia, tramp));
751                 __x86_32_trampoline_normalize(aia, tramp);
752             }
753         }
754     }
755
756     AIA_FOR_EACH_FUNC(aia, func) {
757         fmeta = func->meta_data;
758         hash_map_for_each_destroy(&fmeta->func_local_offsets,
759             func_local_hash_destroy);
760         free_mem(func->meta_data);
761     }
762
763     if (cmdopts.dump_final_x86_32_ic)
764         x86_32_dump_final_ic(aia);
765 }

```

:

A.9.12 src/x86_32/x86_32_func_normalize.h

```

1  #ifndef X86_32_FUNC_NORMALIZE_H
2  #define X86_32_FUNC_NORMALIZE_H
3
4  void x86_32_func_normalize(Aia *aia);
5
6  #endif // X86_32_FUNC_NORMALIZE_H

```

:

A.9.13 src/x86_32/x86_32_normalize.c

```

1  #include "x86_32_normalize.h"
2  #include "x86_32.h"
3  #include "x86_32_regs.h"

```

```

4  #include <main.h>
5
6  static inline void x86_32_verify_src_operand(Aia_Operand *src_op UNUSED)
7  {
8      DEBUGT(def,
9          switch (aia_operand_get_type(src_op)) {
10             case AIA_OPERAND_CONST_STRING:
11             case AIA_OPERAND_BLOCK:
12                 assert(false);
13             default:
14                 break;
15         }
16     );
17 }
18
19 static inline void x86_32_verify_src_dest_operands(Aia_Operand *dest UNUSED,
20     Uns src_op_count UNUSED, ...)
21 {
22     DEBUGT(def,
23         if (dest)
24             assert(aia_operand_is_dest(dest));
25         VA_SETUP(vl, src_op_count);
26         while (src_op_count--)
27             x86_32_verify_src_operand(va_arg(vl, Aia_Operand *));
28         VA_END(vl);
29     );
30 }
31
32 static inline Aia_Instr *x86_32_mov(Aia_Operand *src, Aia_Operand *dest,
33     uint8_t op_sizes, Aia_Block *block, File_Location *loc)
34 {
35     Aia_Instr *mov_in = aia_instr_alloc_lop(AIA_MOV,
36         block, op_sizes, op_sizes, loc);
37     aia_instr_set_dest_op(mov_in, dest);
38     aia_instr_set_src_op(mov_in, 0, src);
39     return mov_in;
40 }
41
42 static void x86_32_mov_normalize(Aia *aia, Aia_Instr *instr)
43 {
44     Aia_Operand *dest = aia_instr_get_dest_op(instr);
45     Aia_Operand *src = aia_instr_get_src_op(instr, 0);
46     Aia_Operand *tmp_dest = NULL;
47
48     x86_32_verify_src_dest_operands(dest, 1, src);
49
50     switch (aia_instr_get_type(instr)) {
51     case AIA_MOVS:
52         /* Fall. */
53     case AIA_MOVZ:
54         if (aia_operand_is_integer(src)) {
55             Aia_Instr *new_mov = x86_32_mov(src, dest,
56                 aia_instr_get_dest_op_size(instr),
57                 aia_instr_get_block(instr),
58                 aia_instr_get_location(instr));
59             aia_instr_replace(instr, new_mov);
60             vector_append(aia->meta_data, instr);
61             instr = new_mov;
62         }
63         if (aia_operand_get_type(src) != AIA_OPERAND_REG) {
64             Aia_Instr *new_mov = aia_mov_to_tmp_reg_before(aia, src, instr,
65                 aia_instr_get_src_ops_size(instr));
66             aia_instr_replace_op(instr, 0, aia_instr_get_dest_op(new_mov));
67         }
68         if (aia_operand_get_type(dest) != AIA_OPERAND_REG) {
69             tmp_dest = aia_operand_tmp_reg_alloc(aia);
70             aia_instr_replace_op(instr, -1, tmp_dest);
71         }
72         break;
73
74     case AIA_MOV:
75         if (aia_operand_is_mem(src) && aia_operand_is_mem(dest)) {
76             Aia_Instr *new_mov = aia_mov_to_tmp_reg_before(aia, src, instr,
77                 aia_instr_get_src_ops_size(instr));

```

```

78     aia_instr_replace_op(instr, 0, aia_instr_get_dest_op(new_mov));
79 }
80 break;
81
82 default:
83
84     fatal_error(S("unexpected mov instruction. Aborting...\n"));
85     break;
86 }
87 if (aia_instr_get_src_ops_size(instr) == AIA_BYTE &&
88     aia_operand_get_type(src) == AIA_OPERAND_CONST_INT) {
89     int32_t orig = aia_operand_const_int_get_val(src);
90     int8_t val = orig;
91     if (val != orig) {
92         Aia_Operand *nconst = aia_operand_const_int_alloc(aia, 0);
93         aia_instr_replace_op(instr, 0, nconst);
94         if (cmdopts.warn_overflow)
95             report_warning_location(aia_instr_get_location(instr),
96                                   S("constant implicitly truncated to " QFY("%" PRIu8)
97                                     " to fit " QFY("char") " variable\n"), val);
98     }
99 }
100 if (tmp_dest)
101     aia_mov_after(tmp_dest, dest, instr, AIA_LONG);
102 }
103
104 static inline Aia_Operand *x86_32_arith_instr_get_const_int(Aia *aia,
105                                                            Aia_Operand *src_dest, Aia_Operand *src_src, Aia_Instr *instr)
106 {
107     int32_t in_result;
108     int32_t lhs = aia_operand_const_int_get_val(src_dest);
109     int32_t rhs = aia_operand_const_int_get_val(src_src);
110     switch (aia_instr_get_type(instr)) {
111     case AIA_SUB:
112         in_result = lhs - rhs;
113         break;
114     case AIA_ADD:
115         in_result = lhs + rhs;
116         break;
117     case AIA_IMUL:
118         in_result = lhs * rhs;
119         break;
120     case AIA_IDIV:
121         in_result = lhs / rhs;
122         break;
123     default:
124         assert(false);
125     }
126     return aia_operand_const_int_alloc(aia, in_result);
127 }
128
129 static inline Aia_Operand *x86_32_arith_instr_get_label_addr(Aia *aia,
130                                                            Aia_Operand *src_label, Aia_Operand *src_int, Aia_Instr *instr)
131 {
132     int32_t val = aia_operand_const_int_get_val(src_int);
133     switch (aia_instr_get_type(instr)) {
134     case AIA_SUB:
135         val = -val;
136         break;
137     case AIA_ADD:
138         break;
139     default:
140         assert(false);
141     }
142     return aia_operand_label_addr_alloc(aia,
143                                         aia_operand_label_get_name(src_label),
144                                         aia_operand_label_get_offset(src_label) + val);
145 }
146
147 static inline Aia_Operand *x86_32_arith_instr_get_src_integer(Aia *aia,
148                                                            Aia_Operand *src_dest, Aia_Operand *src_src, Aia_Instr *instr)
149 {
150     Aia_Operand_Type dest_type = aia_operand_get_type(src_dest);
151     Aia_Operand_Type src_type = aia_operand_get_type(src_src);

```

```

152     if (dest_type == src_type) {
153         assert(dest_type != AIA_OPERAND_LABEL_ADDR);
154         return x86_32_arith_instr_get_const_int(aia, src_dest, src_src, instr);
155     } else {
156         if (dest_type == AIA_OPERAND_LABEL_ADDR)
157             return x86_32_arith_instr_get_label_addr(aia, src_dest,
158                 src_src, instr);
159         return x86_32_arith_instr_get_label_addr(aia, src_src,
160             src_dest, instr);
161     }
162 }
163
164 static inline void x86_32_arith_instr_to_mov(Aia *aia, Aia_Instr *instr)
165 {
166     Aia_Operand *dest = aia_instr_get_dest_op(instr);
167     Aia_Operand *src_dest = aia_instr_get_src_op(instr, 0);
168     Aia_Operand *src_src = aia_instr_get_src_op(instr, 1);
169
170     Aia_Operand *src_int = x86_32_arith_instr_get_src_integer(aia, src_dest,
171         src_src, instr);
172
173     Aia_Instr *mov_in = x86_32_mov(src_int, dest,
174         aia_instr_get_dest_op_size(instr),
175         aia_instr_get_block(instr),
176         aia_instr_get_location(instr));
177
178     aia_instr_replace(instr, mov_in);
179     vector_append(aia->meta_data, instr);
180 }
181
182 static void x86_32_add_sub_normalize(Aia *aia, Aia_Instr *instr)
183 {
184     Aia_Operand *dest = aia_instr_get_dest_op(instr);
185     Aia_Operand *src_dest = aia_instr_get_src_op(instr, 0);
186     Aia_Operand *src_src = aia_instr_get_src_op(instr, 1);
187
188     x86_32_verify_src_dest_operands(dest, 2, src_dest, src_src);
189     assert(aia_instr_get_dest_op_size(instr) ==
190         aia_instr_get_src_ops_size(instr));
191
192     if (aia_operand_is_integer(src_dest) && aia_operand_is_integer(src_src)) {
193         x86_32_arith_instr_to_mov(aia, instr);
194         return;
195     } else if (!aia_operands_equal(src_dest, dest)) {
196         Aia_Instr *mov_in = x86_32_mov(src_dest, dest,
197             aia_instr_get_dest_op_size(instr),
198             aia_instr_get_block(instr),
199             aia_instr_get_location(instr));
200         aia_instr_insert_before(mov_in, instr);
201         x86_32_mov_normalize(aia, mov_in);
202         aia_instr_replace_op(instr, 0, dest);
203     }
204     if (aia_operand_is_mem(src_src) && aia_operand_is_mem(dest)) {
205         Aia_Instr *new_mov = aia_mov_to_tmp_reg_before(aia, src_src, instr,
206             aia_instr_get_dest_op_size(instr));
207         aia_instr_replace_op(instr, 1, aia_instr_get_dest_op(new_mov));
208     }
209 }
210
211 static void x86_32_imul_normalize(Aia *aia, Aia_Instr *instr)
212 {
213     Aia_Operand *dest = aia_instr_get_dest_op(instr);
214     Aia_Operand *src_dest = aia_instr_get_src_op(instr, 0);
215     Aia_Operand *src_src = aia_instr_get_src_op(instr, 1);
216
217     x86_32_verify_src_dest_operands(dest, 2, src_dest, src_src);
218
219     if (aia_operand_is_integer(src_dest) && aia_operand_is_integer(src_src)) {
220         x86_32_arith_instr_to_mov(aia, instr);
221         return;
222     }
223
224     if (!aia_operand_is_reg(dest)) {
225         Aia_Operand *tmp = aia_operand_tmp_reg_alloc(aia);

```

```

226     Aia_Instr *mov_in = x86_32_mov(tmp, dest, AIA_LONG,
227         aia_instr_get_block(instr),
228         aia_instr_get_location(instr));
229     dest = tmp;
230     aia_instr_replace_op(instr, -1, dest);
231     aia_instr_insert_after(mov_in, instr);
232 }
233
234 if (aia_operand_is_integer(src_dest)) {
235     aia_instr_swap_ops(instr, 0, 1);
236     return;
237 } else if (aia_operand_is_integer(src_src)) {
238     return;
239 }
240
241 Aia_Instr *mov_in = x86_32_mov(src_dest, dest, AIA_LONG,
242     aia_instr_get_block(instr),
243     aia_instr_get_location(instr));
244 aia_instr_insert_before(mov_in, instr);
245 aia_instr_replace_op(instr, 0, dest);
246 }
247
248 static void x86_32_idiv_normalize(Aia *aia, Aia_Instr *instr)
249 {
250     if (aia_instr_is_normalized(instr))
251         return;
252
253     Aia_Operand *dest = aia_instr_get_dest_op(instr);
254     Aia_Operand *src_dest = aia_instr_get_src_op(instr, 0);
255     Aia_Operand *src_src = aia_instr_get_src_op(instr, 1);
256
257     x86_32_verify_src_dest_operands(dest, 2, src_dest, src_src);
258
259     if (aia_operand_is_integer(src_dest) && aia_operand_is_integer(src_src)) {
260         if (aia_operand_get_type(src_src) != AIA_OPERAND_CONST_INT ||
261             aia_operand_const_int_get_val(src_src) != 0) {
262             x86_32_arith_instr_to_mov(aia, instr);
263             return;
264         }
265     }
266
267     Aia_Block *block = aia_instr_get_block(instr);
268
269     Aia_Instr *mov_in = x86_32_mov(src_dest, reg_eax, AIA_LONG, block,
270         aia_instr_get_location(instr));
271     aia_instr_insert_before(mov_in, instr);
272
273     Aia_Instr *cdq = aia_instr_alloc_lol(AIA_CDQ, block,
274         AIA_LONG, AIA_LONG, aia_instr_get_location(instr));
275     aia_instr_set_src_op(cdq, 0, reg_eax);
276     aia_instr_set_dest_op(cdq, reg_edx);
277     aia_instr_insert_before(cdq, instr);
278
279     if (aia_operand_is_integer(src_src)) {
280         Aia_Operand *tmp_reg = aia_operand_tmp_reg_alloc(aia);
281         mov_in = x86_32_mov(src_src, tmp_reg, AIA_LONG, block,
282             aia_instr_get_location(instr));
283         aia_instr_insert_before(mov_in, instr);
284         src_src = tmp_reg;
285     }
286
287     mov_in = x86_32_mov(reg_eax, dest, AIA_LONG, block,
288         aia_instr_get_location(instr));
289     aia_instr_insert_after(mov_in, instr);
290
291     aia_instr_replace_op(instr, -1, reg_eax);
292     aia_instr_replace_op(instr, 0, reg_eax);
293     aia_instr_replace_op(instr, 1, src_src);
294 }
295
296 static inline bool x86_32_cmp_is_commutative(Aia_Instr *instr)
297 {
298     Aia_Instr *suc = aia_instr_get_sucessor(instr);
299     assert(suc);

```

```

300     switch (aia_instr_get_type(suc)) {
301     case AIA_JE:
302         /* Fall through. */
303     case AIA_JNE:
304         /* Fall through. */
305     case AIA_SETE:
306         /* Fall through. */
307     case AIA_SETNE:
308         return true;
309
310     default:
311         return false;
312     }
313 }
314
315 static void x86_32_cmp_normalize(Aia *aia, Aia_Instr *instr)
316 {
317     Aia_Operand *src_dest = aia_instr_get_src_op(instr, 0);
318     Aia_Operand *src_src = aia_instr_get_src_op(instr, 1);
319
320     x86_32_verify_src_dest_operands(NULL, 2, src_dest, src_src);
321
322     if ((aia_operand_is_mem(src_dest) && aia_operand_is_mem(src_src)) ||
323         (aia_operand_is_integer(src_dest) &&
324          (!x86_32_cmp_is_commutative(instr) ||
325           aia_operand_is_integer(src_src)))) {
326         Aia_Instr *mov_in = aia_mov_to_tmp_reg_before(aia, src_dest, instr,
327             aia_instr_get_src_ops_size(instr));
328         aia_instr_replace_op(instr, 0, aia_instr_get_dest_op(mov_in));
329     } else if (aia_operand_is_integer(src_dest)) {
330         aia_instr_swap_ops(instr, 0, 1);
331     }
332 }
333
334 static void x86_32_ret_normalize(Aia *aia UNUSED, Aia_Instr *instr)
335 {
336     if (!aia_instr_get_src_op_count(instr) || aia_instr_is_normalized(instr))
337         return;
338
339     Aia_Operand *ret = aia_instr_get_src_op(instr, 0);
340
341     x86_32_verify_src_dest_operands(NULL, 1, ret);
342
343     Aia_Block *block = aia_instr_get_block(instr);
344
345     Aia_Instr *mov_in;
346     Aia_Operand *a_reg;
347     if (aia_instr_get_src_ops_size(instr) == AIA_LONG) {
348         mov_in = x86_32_mov(ret, reg_eax, AIA_LONG, block,
349             aia_instr_get_location(instr));
350         a_reg = reg_eax;
351     } else {
352         assert(aia_instr_get_src_ops_size(instr) == AIA_BYTE);
353         mov_in = x86_32_mov(ret, reg_al, AIA_BYTE, block,
354             aia_instr_get_location(instr));
355         a_reg = reg_al;
356     }
357     aia_instr_insert_before(mov_in, instr);
358     aia_instr_replace_op(instr, 0, a_reg);
359 }
360
361 static void x86_32_call_normalize(Aia *aia UNUSED, Aia_Instr *instr)
362 {
363     if (aia_instr_is_normalized(instr))
364         return;
365
366     Aia_Operand *ret = aia_instr_get_dest_op(instr);
367     Aia_Operand *src = aia_instr_get_src_op(instr, 0);
368
369     x86_32_verify_src_dest_operands(ret, 1, src);
370
371     if (ret) {
372         Aia_Block *block = aia_instr_get_block(instr);
373

```

```

374     Aia_Instr *mov_in;
375     Aia_Operand *a_reg;
376     if (aia_instr_get_dest_op_size(instr) == AIA_LONG) {
377         mov_in = x86_32_mov(reg_eax, ret, AIA_LONG, block,
378             aia_instr_get_location(instr));
379         a_reg = reg_eax;
380     } else {
381         mov_in = x86_32_mov(reg_al, ret, AIA_BYTE, block,
382             aia_instr_get_location(instr));
383         a_reg = reg_al;
384     }
385     aia_instr_insert_after(mov_in, instr);
386     aia_instr_replace_op(instr, -1, a_reg);
387 }
388 }
389
390 static void x86_32_neg_normalize(Aia *aia, Aia_Instr *instr)
391 {
392     Aia_Operand *src = aia_instr_get_src_op(instr, 0);
393     Aia_Operand *dest = aia_instr_get_dest_op(instr);
394
395     x86_32_verify_src_dest_operands(dest, 1, src);
396     if (aia_operand_is_integer(src)) {
397         assert(aia_operand_get_type(src) == AIA_OPERAND_CONST_INT);
398         Aia_Operand *cint = aia_operand_const_int_alloc(aia,
399             -aia_operand_const_int_get_val(src));
400         Aia_Instr *mov_in = x86_32_mov(cint, dest, AIA_LONG,
401             aia_instr_get_block(instr), aia_instr_get_location(instr));
402         aia_instr_replace(instr, mov_in);
403         vector_append(aia->meta_data, instr);
404         return;
405     }
406     if (!aia_operands_equal(src, dest)) {
407         Aia_Instr *mov_in = x86_32_mov(src, dest, AIA_LONG,
408             aia_instr_get_block(instr),
409             aia_instr_get_location(instr));
410         aia_instr_insert_before(mov_in, instr);
411         aia_instr_replace_op(instr, 0, dest);
412     }
413 }
414
415 static void x86_32_instr_normalize(Aia *aia, Aia_Instr *instr)
416 {
417     switch (aia_instr_get_type(instr)) {
418     case AIA_MOV:
419         assert(aia_instr_get_dest_op_size(instr) ==
420             aia_instr_get_src_ops_size(instr));
421         x86_32_mov_normalize(aia, instr);
422         break;
423
424     case AIA_MOVZ:
425         /* Fall through. */
426     case AIA_MOVS:
427         assert(aia_instr_get_dest_op_size(instr) == AIA_LONG &&
428             aia_instr_get_src_ops_size(instr) == AIA_BYTE);
429         x86_32_mov_normalize(aia, instr);
430         break;
431
432     case AIA_ADD:
433         /* Fall through. */
434     case AIA_SUB:
435         assert(aia_instr_get_dest_op_size(instr) == AIA_LONG &&
436             aia_instr_get_src_ops_size(instr) == AIA_LONG);
437         x86_32_add_sub_normalize(aia, instr);
438         break;
439
440     case AIA_IMUL:
441         assert(aia_instr_get_dest_op_size(instr) == AIA_LONG &&
442             aia_instr_get_src_ops_size(instr) == AIA_LONG);
443         x86_32_imul_normalize(aia, instr);
444         break;
445
446     case AIA_IDIV:
447         assert(aia_instr_get_dest_op_size(instr) == AIA_LONG &&

```

```

448         aia_instr_get_src_ops_size(instr) == AIA_LONG);
449     x86_32_idiv_normalize(aia, instr);
450     break;
451
452     case AIA_CMP:
453         x86_32_cmp_normalize(aia, instr);
454         break;
455
456     AIA_CASE_COND_JUMP:
457     case AIA_JMP:
458         break;
459     case AIA_NOP:
460         break;
461
462     case AIA_SETE:
463         /* Fall through. */
464     case AIA_SETNE:
465         /* Fall through. */
466     case AIA_SETL:
467         /* Fall through. */
468     case AIA_SETG:
469         /* Fall through. */
470     case AIA_SETLE:
471         /* Fall through. */
472     case AIA_SETGE:
473         x86_32_verify_src_dest_operands(aia_instr_get_dest_op(instr), 0);
474         break;
475
476     case AIA_RET:
477         x86_32_ret_normalize(aia, instr);
478         break;
479     case AIA_CALL:
480         assert(aia_instr_get_src_ops_size(instr) == AIA_LONG);
481         x86_32_call_normalize(aia, instr);
482         break;
483     case AIA_NEG:
484         assert(aia_instr_get_dest_op_size(instr) == AIA_LONG &&
485             aia_instr_get_src_ops_size(instr) == AIA_LONG);
486         x86_32_neg_normalize(aia, instr);
487         break;
488     case AIA_CDQ:
489         break;
490
491     /* Special AIA implementation specific instructions. */
492     case __AIA_LABEL:
493         break;
494     case __AIA_STRING:
495         break;
496     case __AIA_INTEGER:
497         break;
498
499     default:
500         fatal_error(S("unimplemented x86-32 instruction normalization\n"));
501     }
502
503     aia_instr_set_normalized(instr);
504 }
505
506 static void x86_32_block_callback(Aia_Block *b, Aia *aia)
507 {
508     Aia_Instr *instr;
509     AIA_BLOCK_FOR_EACH_INSTRUCTION(b, instr)
510         x86_32_instr_normalize(aia, instr);
511     vector_for_each_destroy(aia->meta_data,
512         (Vector_Destructor)aia_instr_destroy);
513 }
514
515 static void x86_32_section_normalize(Aia *aia, Aia_Section *sec)
516 {
517     Double_List_Node *bnode;
518     DOUBLE_LIST_FOR_EACH(&sec->sec_blist, bnode)
519         x86_32_block_callback(AIA_BLOCK_OF_DBNODE(bnode), aia);
520
521     Aia_Func *func;

```

```

522     AIA_SECTION_FOR_EACH_FUNC(sec, func) {
523         DOUBLE_LIST_FOR_EACH(&func->blist, bnode)
524             x86_32_block_callback(AIA_BLOCK_OF_DBNODE(bnode), aia);
525
526         Aia_Func_Trampoline *tramp;
527         AIA_FUNC_FOR_EACH_TRAMPOLINE(func, tramp)
528             x86_32_block_callback(tramp->block, aia);
529     }
530 }
531
532 static void x86_32_normalize_dump(Aia *aia)
533 {
534     String fname = string_from_format(S("%S.vitaly.norm-x86-32-ic"),
535         aia_get_file_name(aia));
536     FILE *f = file_open(fname, S("w"));
537     if (!f)
538         fatal_error(S("unable to open file %S for intermediate code dump\n"),
539             fname);
540     aia_dump(aia, f);
541     file_close(f);
542     string_destroy(fname);
543 }
544
545 void x86_32_normalize_instructions(Aia *aia, Vector *instructions)
546 {
547     void *saved_data = aia->meta_data;
548     aia->meta_data = vector_alloc();
549     Aia_Instr *in;
550     VECTOR_FOR_EACH_ENTRY(instructions, in)
551         x86_32_instr_normalize(aia, in);
552     vector_destroy(aia->meta_data, (Vector_Destructor)aia_instr_destroy);
553     aia->meta_data = saved_data;
554 }
555
556 void x86_32_normalize(Aia *aia)
557 {
558     if (!aia_is_valid(aia))
559         return;
560
561     aia->meta_data = vector_alloc();
562     Aia_Section *sec;
563     AIA_FOR_EACH_SECTION(aia, sec)
564         x86_32_section_normalize(aia, sec);
565     vector_destroy(aia->meta_data, (Vector_Destructor)aia_instr_destroy);
566     if (cmdopts.dump_norm_x86_32_ic)
567         x86_32_normalize_dump(aia);
568 }

```

:

A.9.14 src/x86_32/x86_32_normalize.h

```

1  #ifndef X86_32_NORMALIZE_H
2  #define X86_32_NORMALIZE_H
3
4  #include <aia/aia.h>
5
6  typedef struct Vector Vector;
7
8  //void x86_32_normalize_instructions(Aia *aia, Vector *instructions);
9
10 /* Scrambles aia->meta_data. */
11 void x86_32_normalize(Aia *aia);
12
13 #endif // X86_32_NORMALIZE_H

```

A.10 Compiler Library

:

A.10.1 src/alloc.c

```

1  #include <std_include.h>
2  #include <stdlib.h>
3
4  void *alloc_zeros(Uns size)
5  {
6      void *ptr;
7      if (! (ptr = calloc(1, size)) && size > 0)
8          fatal_error(S("unable to allocate memory\n"));
9      return ptr;
10 }
11
12 void *alloc_mem(Uns size)
13 {
14     void *ptr;
15     if (! (ptr = malloc(size)) && size > 0)
16         fatal_error(S("unable to allocate memory\n"));
17     return ptr;
18 }
19
20 void *realloc_mem(void *old_ptr, Uns size)
21 {
22     void *ptr;
23     if (! (ptr = realloc(old_ptr, size)) && size > 0)
24         fatal_error(S("unable to allocate memory\n"));
25     return ptr;
26 }
27
28 void free_mem(void *ptr)
29 {
30     free(ptr);
31 }

```

:

A.10.2 src/alloc.h

```

1  #ifndef ALLOC_HANDLER_H
2  #define ALLOC_HANDLER_H
3
4  void *alloc_mem(Uns size) __attribute__((malloc));
5
6  void *realloc_mem(void *old_ptr, Uns size);
7
8  void *alloc_zeros(Uns size) __attribute__((malloc));
9
10 void free_mem(void *ptr);
11
12 #define ALLOC_NEW(obj_name) alloc_mem(sizeof(obj_name))
13
14 #endif // ALLOC_HANDLER_H

```

:

A.10.3 src/bit_vector.h

```

1  #ifndef BIT_VECTOR_H
2  #define BIT_VECTOR_H
3
4  #include <vector.h>

```

```

5
6 typedef Vector Bit_Vector;
7
8 static inline Bit_Vector *bit_vector_alloc()
9 {
10     return vector_alloc();
11 }
12
13 static inline Uns __bit_vector_get_vector_idx(Bit_Vector *v, Uns idx)
14 {
15     Uns v_idx = idx / PTR_BITS;
16     while (v_idx >= vector_size(v))
17         vector_append(v, INT_TO_PTR(0));
18     return v_idx;
19 }
20
21 static inline void bit_vector_set(Bit_Vector *v, Uns idx)
22 {
23     Uns v_idx = __bit_vector_get_vector_idx(v, idx);
24     v->array[v_idx] |= 1 << (idx - v_idx * PTR_BITS);
25 }
26
27 static inline void bit_vector_clear(Bit_Vector *v, Uns idx)
28 {
29     Uns v_idx = __bit_vector_get_vector_idx(v, idx);
30     v->array[v_idx] &= ~(1 << (idx - v_idx * PTR_BITS));
31 }
32
33 static inline bool bit_vector_test(Bit_Vector *v, Uns idx)
34 {
35     Uns v_idx = __bit_vector_get_vector_idx(v, idx);
36     return v->array[v_idx] & (1 << (idx - v_idx * PTR_BITS));
37 }
38
39 static inline void bit_vector_complement(Bit_Vector *v, Uns idx)
40 {
41     if (bit_vector_test(v, idx))
42         bit_vector_clear(v, idx);
43     else
44         bit_vector_set(v, idx);
45 }
46
47 #endif // BIT_VECTOR_H

```

:

A.10.4 src/debug.c

```

1 #ifndef NDEBUG
2 #include <debug.h>
3
4 #include <hash_map.h>
5 #include <stdlib.h>
6 #include <string.h>
7 #include <vector.h>
8
9 typedef struct Debug_Type {
10     Hash_Map_Slot slot;
11     String type;
12 } Debug_Type;
13
14 #define DEBUG_TYPE_OF(s) HASH_MAP_ENTRY(s, Debug_Type, slot)
15
16 static bool debug_comparator(void *sobj, Hash_Map_Slot *mslot)
17 {
18     Const_String s = (Const_String)sobj;
19     Debug_Type *t = DEBUG_TYPE_OF(mslot);
20     return string_compare(s, t->type) == 0;
21 }
22
23 HASH_MAP_SIZE(debug_hmap, HASH_MAP_SIZE_11, debug_comparator);

```

```

24
25 void __debug_enable_type(Const_String type)
26 {
27     Uns hash = string_hash_code(type);
28     if (!hash_map_contains(&debug_hmap, (void *)type, hash)) {
29         Debug_Type *t = ALLOC_NEW(Debug_Type);
30         t->type = string_duplicate(type);
31         hash_map_insert(&debug_hmap, &t->slot, hash);
32     }
33 }
34
35 void __debug_disable_type(Const_String type)
36 {
37     if (string_compare(type, DEBUG_DEFAULT_STR) == 0)
38         return;
39     Hash_Map_Slot *slot = hash_map_remove(&debug_hmap, (void *)type,
40         string_hash_code(type));
41     if (slot) {
42         string_destroy(DEBUG_TYPE_OF(slot)->type);
43         free_mem(DEBUG_TYPE_OF(slot));
44     }
45 }
46
47 bool __debug_type_is_enabled(Const_String type)
48 {
49     Vector *v = string_split(type, DEBUG_DELIMITER_STR);
50     String s;
51     bool valid = false;
52     VECTOR_FOR_EACH_ENTRY(v, s) {
53         if (hash_map_contains(&debug_hmap, s, string_hash_code(s))) {
54             valid = true;
55             break;
56         }
57     }
58     vector_destroy(v, (Vector_Destructor)string_destroy);
59     return valid;
60 }
61
62 void __debug_log_enabled_types()
63 {
64     Hash_Map_Slot *s;
65     HASH_MAP_FOR_EACH(&debug_hmap, s)
66         DLOGT(DEBUG_DEFAULT, "DEBUG TYPE ENABLED: %S\n",
67             DEBUG_TYPE_OF(s)->type);
68 }
69
70 static void debug_insert_values() CONSTRUCTOR;
71 static void debug_insert_values()
72 {
73     Vector *v;
74     String s;
75     char *debug_types = getenv(DEBUG_ENVIRON_CSTR);
76     if (debug_types) {
77         v = string_split(S(debug_types), DEBUG_DELIMITER_STR);
78         VECTOR_FOR_EACH_ENTRY(v, s)
79             __debug_enable_type(s);
80         vector_destroy(v, (Vector_Destructor)string_destroy);
81     }
82     __debug_enable_type(DEBUG_DEFAULT_STR);
83 }
84
85 static void debug_destructor(Hash_Map_Slot *slot)
86 {
87     Debug_Type *t = DEBUG_TYPE_OF(slot);
88     string_destroy(t->type);
89     free_mem(t);
90 }
91
92 static void debug_shutdown() DESTRUCTOR;
93 static void debug_shutdown()
94 {
95     hash_map_for_each_destroy(&debug_hmap, debug_destructor);
96 }
97

```

```
98 #endif // NDEBUG
```

```
:
```

A.10.5 src/debug.h

```
1 #ifndef DEBUG_H
2 #define DEBUG_H
3
4 #include <io.h>
5
6 #define DEBUG_DEFAULT def
7
8 #ifndef DEBUG_TYPE
9 #define DEBUG_TYPE DEBUG_DEFAULT
10 #endif
11
12 #define DEBUG_DEFAULT_STR STRINGIFY(DEBUG_DEFAULT)
13 #define DEBUG_DELIMITER_STR S(":")
14 #define DEBUG_ENVIRON_CSTR "DEBUG"
15
16 #ifndef NDEBUG
17
18 void __debug_enable_type(Const_String type);
19
20 void __debug_disable_type(Const_String type);
21
22 bool __debug_type_is_enabled(Const_String type);
23
24 void __debug_log_enabled_types();
25
26 #define DEBUGT(type, statements) \
27     do { \
28         if (__debug_type_is_enabled(STRINGIFY(type))) { \
29             statements; \
30         } \
31     } while (0)
32
33 #define DEBUG(statements) DEBUGT(DEBUG_TYPE, statements)
34
35 #define DLOGT(type, fmt, ...) \
36     DEBUGT(type, file_print_message(stderr, S(fmt), ## __VA_ARGS__));
37
38 #define DLOG(fmt, ...) \
39     DEBUG(file_print_message(stderr, S(fmt), ## __VA_ARGS__));
40
41 /* Print enabled debugging types to stderr. */
42 #define DEBUG_PRINT_TYPES() __debug_log_enabled_types()
43
44 #if 0
45 #define DEBUG_ENABLE(type) __debug_enable_type(STRINGIFY(type))
46 #define DEBUG_DISABLE(type) __debug_disable_type(STRINGIFY(type))
47 #endif
48
49 #else
50
51 #define DEBUGT(type, statements) do {} while (0)
52 #define DEBUG(statements) do {} while (0)
53 #define DLOGT(type, fmt, ...) do {} while (0)
54 #define DLOG(fmt, ...) do {} while (0)
55 #define DEBUG_PRINT_TYPES() do {} while (0)
56 #if 0
57 #define DEBUG_ENABLE(type) do {} while (0)
58 #define DEBUG_DISABLE(type) do {} while (0)
59 #endif
60
61 #endif // NDEBUG
62
63 #endif // DEBUG_H
```

:

A.10.6 src/dot_printer.c

```

1  #include <dot_printer.h>
2  #include <alloc.h>
3
4  Dot_Printer *dot_printer_init(Const_String file_name_prefix,
5                               Const_String file_name_suffix, Const_String rank_dir)
6  {
7      String file_name = string_from_format(S("%S.%S.dot"),
8      file_name_prefix, file_name_suffix);
9
10     Dot_Printer *dot_printer = ALLOC_NEW(Dot_Printer);
11     dot_printer->output_file = file_open(file_name, S("w"));
12     dot_printer->id_stack = SINGLE_LIST_INIT();
13     dot_printer->file_name_prefix = file_name_prefix;
14     dot_printer->file_name_suffix = file_name_suffix;
15
16     if (!dot_printer->output_file)
17         fatal_error(S("unable to open file '%S' for graph dump\n"),
18                     file_name);
19     string_destroy(file_name);
20     file_print_message(dot_printer->output_file,
21                        S("strict digraph {\n\ttrankdir=%S;\n"}, rank_dir);
22
23     dot_printer_push_id(dot_printer, (void *) 0);
24     return dot_printer;
25 }
26
27 void stack_destructor(Single_List_Node *node)
28 {
29     free_mem(CONTAINER_OF(node, Dot_Printer_Id_Stack_Node, list_node));
30 }
31
32 void dot_printer_destroy(Dot_Printer *dot_printer)
33 {
34     single_list_for_each_destroy(&dot_printer->id_stack, stack_destructor);
35     free_mem(dot_printer);
36 }
37
38 void dot_printer_fin_com_des(Dot_Printer *p, Const_String type)
39 {
40     dot_printer_finalize(p);
41     dot_printer_compile(p, type);
42     dot_printer_destroy(p);
43 }
44
45 void dot_printer_push_id(Dot_Printer *dot_printer, void *id)
46 {
47     Dot_Printer_Id_Stack_Node *node = ALLOC_NEW(Dot_Printer_Id_Stack_Node);
48     node->id = id;
49     void *from = dot_printer_peek_current_id(dot_printer);
50     if (id && from)
51         dot_printer_insert_relation(dot_printer, from, id);
52     single_list_prepend(&dot_printer->id_stack, &node->list_node);
53 }
54
55 void *dot_printer_pop_current_id(Dot_Printer *dot_printer)
56 {
57     void *ret = NULL;
58     Single_List_Node *node = single_list_pop_first(&dot_printer->id_stack);
59     if (node) {
60         Dot_Printer_Id_Stack_Node *stack_node =
61             SINGLE_LIST_ENTRY(node, Dot_Printer_Id_Stack_Node, list_node);
62         ret = stack_node->id;
63         free_mem(stack_node);
64     }
65     return ret;
66 }
67
68 void *dot_printer_peek_current_id(Dot_Printer *dot_printer)

```

```

69 {
70     void * ret = NULL;
71     Single_List_Node *node = single_list_peek_first(&dot_printer->id_stack);
72     if (node) {
73         Dot_Printer_Id_Stack_Node *stack_node =
74             SINGLE_LIST_ENTRY(node, Dot_Printer_Id_Stack_Node, list_node);
75         ret = stack_node->id;
76     }
77     return ret;
78 }
79
80 void dot_printer_begin_table(Dot_Printer *p, void *id, int c)
81 {
82     p->current_table_size = c;
83     file_print_message(p->output_file,
84         S("\t%lu[label="
85             "<<table BORDER=\\"0\\" CELLBORDER=\\"1\\" CELLSPACING=\\"0\\">\n"),
86         (long unsigned) id);
87 }
88
89 void dot_printer_insert_row(Dot_Printer *p, int c, ...)
90 {
91     Const_String cell;
92     String tmp_str;
93
94     if (c-p->current_table_size > 0)
95         c = p->current_table_size;
96
97     String_Builder sb = STRING_BUILDER_INIT();
98
99     string_builder_append(&sb, S("\t\t<tr>"));
100
101     VA_SETUP(vl, c);
102     for ( ; (cell = va_arg(vl, Const_String)) && c--; ) {
103         tmp_str = string_from_format(S("<td>%S</td>"), cell);
104         string_builder_append(&sb, tmp_str);
105         string_destroy(tmp_str);
106     }
107     VA_END(vl);
108
109     string_builder_append(&sb, S("</tr>\n"));
110
111     file_print_message(p->output_file, string_builder_const_str(&sb));
112
113     string_builder_clear(&sb);
114 }
115
116 void dot_printer_insert_merge_row(Dot_Printer *p, Const_String header)
117 {
118     file_print_message(p->output_file,
119         S("\t\t<tr><td BGCOLOR=\\"gray\\" COLSPAN=\\"%U\\">%S</td></tr>\n"),
120         p->current_table_size, header);
121 }
122
123
124 void dot_printer_end_table(Dot_Printer *p)
125 {
126     file_print_message(p->output_file, S("\t</table>> shape=plaintext]\n"));
127 }
128
129 void dot_printer_compile(Dot_Printer *p, Const_String type)
130 {
131     String cmd = string_from_format(
132         S("dot -T%S %2$S.%3$S.dot -o %2$S.%3$S.pdf"),
133         type, p->file_name_prefix, p->file_name_suffix);
134
135     //TODO handle error
136     int ret = system(string_to_cstr(cmd));
137     if (ret == -1) {
138         report_error(p->file_name_prefix, S("Call to system() failed!\n"));
139     } else if (WEXITSTATUS(ret)) {
140         report_error(p->file_name_prefix, S("Call to dot returned: %d\n"),
141             WEXITSTATUS(ret));
142     }
143 }

```

```

143
144     string_destroy(cmd);
145 }

```

```

:
```

A.10.7 src/dot_printer.h

```

1  #ifndef DOT_PRINTER_H
2  #define DOT_PRINTER_H
3
4  #include <single_list.h>
5  #include <std_include.h>
6  #include <alloc.h>
7  #include <string_builder.h>
8  #include <sys/wait.h>
9
10 typedef struct Dot_Printer_Id_Stack_Node {
11     Single_List_Node list_node;
12     void *id;
13 } Dot_Printer_Id_Stack_Node;
14
15 typedef struct Dot_Printer {
16     Single_List id_stack;
17     FILE *output_file;
18     Const_String file_name_prefix;
19     Const_String file_name_suffix;
20     Uns current_table_size;
21 } Dot_Printer;
22
23 /* Allocates and initializes a new dot_printer.
24  *
25  * file_name_prefix specifies the prefix of the filename, to which the output
26  * will be written. The new file created will have the name
27  * file_name_prefix.graph.dot
28  *
29  * rank_dir specifies the direction of the graph created. This should be
30  * either "LR", "RL", "TB" or "BT". For directions left-right, right-left,
31  * top-bottom and bottom-top. */
32 Dot_Printer *dot_printer_init(Const_String file_name_prefix,
33                               Const_String file_name_suffix,
34                               Const_String rank_dir);
35
36 /* Destroys and deallocate a Dot_Printer previously created with
37  * dot_printer_init.
38  *
39  * This function also finishes up the .graph.dot file, so that it is ready for
40  * compilation. */
41 void dot_printer_compile_destroy(Dot_Printer *dot_printer);
42
43 /* Used to enter a new scope, with the scopes parent beeing id. */
44 void dot_printer_push_id(Dot_Printer *dot_printer, void * id);
45
46 /* Used to leave current scope. */
47 void *dot_printer_pop_current_id(Dot_Printer *dot_printer);
48
49 /* Used to identify the currents scope's parent. */
50 void *dot_printer_peek_current_id(Dot_Printer *dot_printer);
51
52 /* Inserts a new node which is identified by id.
53  *
54  * label specifies how the node is reppresented on the final graph. */
55 inline static void dot_printer_insert_node(Dot_Printer *dot_printer,
56                                             void *id, Const_String label)
57 {
58     file_print_message(dot_printer->output_file,
59                       S("\t%lu[label=\"%S\" shape=box];\n"), (long unsigned)id, label);
60 }
61
62 /* Inserts a new node into the graph as a child to the current scope, also
63  * enters a new scope. So that nodes that are inserted, before

```



```

64  * dot_printer_pop_current_id() is called, are added as children
65  * of this node. */
66  inline static void dot_printer_push_insert(Dot_Printer *p, void *id,
67      Const_String str)
68  {
69      dot_printer_push_id(p, id);
70      dot_printer_insert_node(p, id, str);
71  }
72
73  /* Insert a relation from node with id1 to another node identified by id2.
74  *
75  * If id1 and id2 does not exist as nodes in the graph, arbitrary nodes will
76  * be inserted so that the relation holds. (Functionality of dot). */
77  inline static void dot_printer_insert_relation(Dot_Printer *dot_printer,
78      void *id1, void *id2)
79  {
80      if(id1 && id2)
81          file_print_message(dot_printer->output_file,
82              S("\t%lu -> %lu;\n"),
83              (long unsigned) id1,
84              (long unsigned) id2);
85  }
86
87  /* Finalizes and closes the output file. */
88  inline static void dot_printer_finalize(Dot_Printer *p)
89  {
90      file_print_message(p->output_file, S("{}"));
91      file_close(p->output_file);
92  }
93
94  /* Compiles the .dot file associated with the Dot_Printer using Graphviz Dot.
95  *
96  * type specifies the filetype which to compile to. Normally "pdf", see man
97  * pages for dot for more valid types. */
98  void dot_printer_compile(Dot_Printer *p, Const_String type);
99
100 /* Used to finalize, then compile and finally destroy the Dot_Printer pointed
101 * to by p. */
102 void dot_printer_fin_com_des(Dot_Printer *p, Const_String type);
103
104 /***** HANDLE TABLE NODES *****/
105
106 /* Begin a new table node, with id=id and c columns.
107 * Note that this also creates a new scope, until dot_printer_pop_current_id()
108 * is called.
109 *
110 * Must be followed by dot_printer_end_table(), only dot_printer_insert_row()
111 * or dot_printer_insert_merge_row() are allowed in between. */
112 void dot_printer_begin_table(Dot_Printer *p, void *id, int c);
113
114 inline static void dot_printer_push_begin_table(Dot_Printer *p,
115     void *id, int c)
116 {
117     dot_printer_push_id(p, id);
118     dot_printer_begin_table(p, id, c);
119 }
120
121 /* Inserts a row in the current table, containing c columns.
122 *
123 * Note if c is larger than the current table size, only the table size first
124 * columns are inserted. */
125 void dot_printer_insert_row(Dot_Printer *p, int c, ...);
126
127 /* Inserts a row in the current table, where all cells are merged together.
128 * Usefull for adding section headers in the table. */
129 void dot_printer_insert_merge_row(Dot_Printer *p, Const_String header);
130
131 /* Finishes up the current table. After this, other nodes can be inserted.
132 *
133 * If this function isn't called at somepoint after dot_printer_begin_table(),
134 * the ouput file might be corrupted. */
135 void dot_printer_end_table(Dot_Printer *p);
136
137 #endif

```

:

A.10.8 src/double_list.c

1 // #include <double_list.h>

:

A.10.9 src/double_list.h

1 #ifndef DOUBLE_LIST_H
2 #define DOUBLE_LIST_H
3
4 #include <std_defines.h>
5
6 typedef struct Double_List_Node Double_List_Node;
7
8 struct Double_List_Node {
9 Double_List_Node *prev;
10 Double_List_Node *next;
11 };
12
13 typedef Double_List_Node Double_List;
14
15 #define DOUBLE_LIST_STATIC_INIT(list) { \
16 .prev = (Double_List_Node *)&(list), \
17 .next = (Double_List_Node *)&(list) \
18 }
19
20 #define DOUBLE_LIST_INIT(list) \
21 ((Double_List)DOUBLE_LIST_STATIC_INIT(list))
22
23 #define DOUBLE_LIST(list) Double_List list = DOUBLE_LIST_STATIC_INIT(list)
24
25 #define DOUBLE_LIST_ENTRY(node_ptr, container_type, node_member_name) \
26 CONTAINER_OF(node_ptr, container_type, node_member_name)
27
28 static inline Double_List *double_list_alloc()
29 {
30 Double_List *list = ALLOC_NEW(Double_List);
31 *list = DOUBLE_LIST_INIT(*list);
32 return list;
33 }
34
35 static inline void double_list_insert(Double_List_Node *node,
36 Double_List_Node *prev, Double_List_Node *next)
37 {
38 next->prev = node;
39 node->next = next;
40 node->prev = prev;
41 prev->next = node;
42 }
43
44 static inline void double_list_append(Double_List *list,
45 Double_List_Node *node)
46 {
47 double_list_insert(node, list->prev, (void *)list);
48 }
49
50 static inline void double_list_prepend(Double_List *list,
51 Double_List_Node *node)
52 {
53 double_list_insert(node, (void *)list, list->next);
54 }
55
56 static inline bool double_list_is_empty(const Double_List *list)
57 {
58 return list->next == (void *)list;
59 }

```

59 }
60
61 static inline Double_List_Node *double_list_peek_last(Double_List *list)
62 {
63     if (!double_list_is_empty(list))
64         return list->prev;
65     return NULL;
66 }
67
68 static inline Double_List_Node *double_list_peek_first(Double_List *list)
69 {
70     if (!double_list_is_empty(list))
71         return list->next;
72     return NULL;
73 }
74
75 static inline void double_list_remove(Double_List_Node *node)
76 {
77     node->prev->next = node->next;
78     node->next->prev = node->prev;
79 }
80
81 static inline Double_List_Node *double_list_pop_last(Double_List *list)
82 {
83     Double_List_Node *node;
84     if (!double_list_is_empty(list)) {
85         node = list->prev;
86         double_list_remove(node);
87         return node;
88     }
89     return NULL;
90 }
91
92 static inline Double_List_Node *double_list_pop_first(Double_List *list)
93 {
94     Double_List_Node *node;
95     if (!double_list_is_empty(list)) {
96         node = list->next;
97         double_list_remove(node);
98         return node;
99     }
100     return NULL;
101 }
102
103 typedef void (*Double_List_Destructor)(Double_List_Node *node);
104
105 static inline void double_list_for_each_destroy(Double_List *list,
106 Double_List_Destructor destructor)
107 {
108     for (Double_List_Node *n = double_list_pop_first(list); n;
109         n = double_list_pop_last(list))
110         destructor(n);
111 }
112
113 /* Don't free elements from the list while iterating this loop.
114  * At least you need to break the loop right after. */
115 #define DOUBLE_LIST_FOR_EACH(list_ptr, node_ptr) \
116     for (node_ptr = (list_ptr)->next; \
117         node_ptr != (void *) (list_ptr); \
118         node_ptr = (node_ptr)->next)
119
120 /* Don't free elements from the list while iterating this loop.
121  * At least you need to break the loop right after. */
122 #define DOUBLE_LIST_FOR_EACH_REVERSED(list_ptr, node_ptr) \
123     for (node_ptr = (list_ptr)->prev; \
124         node_ptr != (void *) (list_ptr); \
125         node_ptr = (node_ptr)->prev)
126
127 /* Don't free elements from the list while iterating this loop.
128  * At least you need to break the loop right after. */
129 #define DOUBLE_LIST_FOR_EACH_AFTER(list_ptr, init_node, node_ptr) \
130     for (node_ptr = (init_node)->next; \
131         node_ptr != (void *) (list_ptr); \
132         node_ptr = (node_ptr)->next)

```

```

133
134 /* Don't free elements from the list while iterating this loop.
135  * At least you need to break the loop right after. */
136 #define DOUBLE_LIST_FOREACH_FROM(list_ptr, init_node, node_ptr) \
137     for (node_ptr = init_node; \
138          node_ptr != (void *) (list_ptr); \
139          node_ptr = (node_ptr)->next)
140
141 #endif // DOUBLE_LIST_H

```

:

A.10.10 src/file_location.c

```

1 #include <str.h>
2
3 CONST_STRING(file_location_stdin_file_name, "stdin");

```

:

A.10.11 src/file_location.h

```

1 #ifndef FILE_LOCATION_H
2 #define FILE_LOCATION_H
3
4 #include <str.h>
5
6 typedef struct File_Location {
7     Const_String __file_name;
8     Uns line;
9     Uns column;
10 } File_Location;
11
12 #define FILE_LOCATION_STATIC_INIT(fname, lineno, colno) { \
13     .__file_name = fname, \
14     .line = lineno, \
15     .column = colno \
16 }
17
18 #define FILE_LOCATION_INIT(fname, lineno, colno) \
19     ((File_Location)FILE_LOCATION_STATIC_INIT(fname, lineno, colno))
20
21 static inline void file_location_set_file_name(File_Location *f,
22     Const_String file)
23 {
24     f->__file_name = file;
25 }
26
27 static inline Const_String file_location_get_file_name(const File_Location *f)
28 {
29     extern Const_String file_location_stdin_file_name;
30     (void)file_location_stdin_file_name;
31
32     Const_String ret;
33 #ifdef STDIN_INPUT_ENABLED
34     if (string_ends_with(f->__file_name, STDIN_FILE_SUFFIX_STR))
35         ret = file_location_stdin_file_name;
36     else
37         ret = f->__file_name;
38 #else
39     ret = f->__file_name;
40 #endif
41     return ret;
42 }
43
44 /* Compare two file locations based on line number and column number.
45  * Returns -1 if lhs < rhs,

```

```

46  * returns 1 if lhs > rhs,
47  * else returns 0. */
48  static inline Int file_location_cmp_lncol(File_Location *lhs,
49      File_Location *rhs)
50  {
51      if (lhs->line < rhs->line)
52          return -1;
53      if (rhs->line < lhs->line)
54          return 1;
55      if (lhs->column < rhs->column)
56          return -1;
57      if (rhs->column < lhs->column)
58          return 1;
59      return 0;
60  }
61
62  /* Compare two file locations based based on file name, line number
63  * and column number.
64  * Returns -1 if lhs < rhs,
65  * returns 1 if lhs > rhs,
66  * else returns 0. */
67  static inline Int file_location_cmp(File_Location *lhs, File_Location *rhs)
68  {
69      Int res = string_compare(file_location_get_file_name(lhs),
70          file_location_get_file_name(rhs));
71      if (res)
72          return res;
73      return file_location_cmp_lncol(lhs, rhs);
74  }
75
76  #endif // FILE_LOCATION_H

```

:

A.10.12 src/hash_map.c

```

1  #include <hash_map.h>
2  #include <std_include.h>
3
4  const Uns HASH_MAP_PRIME_LIST[HASH_MAP_PRIME_LIST_SIZE] = {
5      2,
6      5,
7      11,
8      23,
9      47,
10     97,
11     197,
12     397,
13     797,
14     1597,
15     3203,
16     6421,
17     12853,
18     25717,
19     51437,
20     102877,
21     205759,
22     411527,
23     823117,
24     1646237,
25     3292489,
26     6584983,
27     13169977,
28     26339969,
29     52679969,
30     105359939,
31     210719881,
32     421439783,
33     842879579
34 };
35

```

```

36 static inline Uns hash_map_get_hash_idx(Hash_Map *map, Uns hash_code)
37 {
38     return hash_code % map->num_slots;
39 }
40
41 static inline void hash_map_slot_set_hash_code(Hash_Map_Slot *slot,
42     Uns hash_code)
43 {
44     slot->hash_code = hash_code;
45 }
46
47 static inline void hash_map_hash(Hash_Map *map, Hash_Map_Slot *slot)
48 {
49     Uns idx = hash_map_get_hash_idx(map, slot->hash_code);
50     single_list_prepend(&map->slots[idx], &slot->neighbor);
51 }
52
53 static void hash_map_rehash(Hash_Map *map, Uns new_size_idx)
54 {
55     Single_List *prev_slots;
56     Single_List *current;
57     Single_List_Node *node;
58     Uns prev_num_slots;
59
60     prev_slots = map->slots;
61     prev_num_slots = map->num_slots;
62     map->num_slots = HASH_MAP_PRIME_LIST[new_size_idx];
63     map->slots = alloc_zeros(sizeof(Single_List) * map->num_slots);
64     map->size_idx = new_size_idx;
65
66     if (prev_slots) {
67         for (Uns idx = 0; idx < prev_num_slots; idx++) {
68             current = &prev_slots[idx];
69             while ((node = single_list_pop_first(current)))
70                 hash_map_hash(map, HASH_MAP_SLOT_OF(node));
71         }
72         free_mem(prev_slots);
73     } else {
74         map->num_elements = 0;
75     }
76 }
77
78 void hash_map_insert(Hash_Map *map, Hash_Map_Slot *slot, Uns hash_code)
79 {
80     if ((float)map->num_elements / map->num_slots >=
81         HASH_MAP_MAX_LOAD_FACTOR &&
82         map->size_idx < HASH_MAP_PRIME_LIST_SIZE) {
83         hash_map_rehash(map, map->size_idx + 1);
84     }
85     hash_map_slot_set_hash_code(slot, hash_code);
86     hash_map_hash(map, slot);
87     ++map->num_elements;
88 }
89
90 bool hash_map_contains(Hash_Map *map, void *search_obj,
91     Uns search_hash_code)
92 {
93     if (!map->slots)
94         return false;
95
96     Single_List_Node *n;
97     Uns idx = hash_map_get_hash_idx(map, search_hash_code);
98     Single_List *bucket = &map->slots[idx];
99     SINGLE_LIST_FOR_EACH(bucket, n)
100         if (map->comparator(search_obj, HASH_MAP_SLOT_OF(n)))
101             return true;
102
103     return false;
104 }
105
106 Hash_Map_Slot *hash_map_get(Hash_Map *map, void *search_obj,
107     Uns search_hash_code)
108 {
109     if (!map->slots)

```

```

110     return NULL;
111
112     Uns idx = hash_map_get_hash_idx(map, search_hash_code);
113     Hash_Map_Slot *slot;
114     Single_List *bucket = &map->slots[idx];
115     Single_List_Node *n;
116
117     SINGLE_LIST_FOR_EACH(bucket, n) {
118         slot = HASH_MAP_SLOT_OF(n);
119         if (map->comparator(search_obj, slot))
120             return slot;
121     }
122
123     return NULL;
124 }
125
126 Hash_Map_Slot *hash_map_remove(Hash_Map *map, void *search_obj,
127                               Uns search_hash_code)
128 {
129     if (!map->slots)
130         return NULL;
131
132     Uns idx = hash_map_get_hash_idx(map, search_hash_code);
133     Hash_Map_Slot *slot;
134     Single_List *bucket = &map->slots[idx];
135     Single_List_Node *n, *prev = NULL;
136
137     SINGLE_LIST_FOR_EACH(bucket, n) {
138         slot = HASH_MAP_SLOT_OF(n);
139         if (map->comparator(search_obj, slot)) {
140             single_list_remove(bucket, n, prev);
141             --map->num_elements;
142
143             if ((float)map->num_elements / map->num_slots <
144                 HASH_MAP_MIN_LOAD_FACTOR &&
145                 map->size_idx > map->initial_size) {
146                 hash_map_rehash(map, map->size_idx - 1);
147             }
148
149             return slot;
150         }
151         prev = n;
152     }
153
154     return NULL;
155 }
156
157 void hash_map_for_each_destroy(Hash_Map *map, Hash_Map_Destructor destructor)
158 {
159     void single_destructor(Single_List_Node *node)
160     {
161         destructor(HASH_MAP_SLOT_OF(node));
162     }
163
164     if (!map->slots)
165         return;
166
167     Single_List *bucket;
168     Uns num_slots = map->num_slots;
169     for (Uns i = 0; i < num_slots; i++) {
170         bucket = &map->slots[i];
171         single_list_for_each_destroy(bucket, single_destructor);
172     }
173
174     hash_map_clear(map);
175 }

```

:

A.10.13 src/hash_map.h

```

1  #ifndef HASH_MAP_H
2  #define HASH_MAP_H
3
4  #include <std_defines.h>
5  #include <single_list.h>
6  #include <alloc.h>
7
8  /* XXX Don't use debug macros to debug hash map implementation since the
9   * debug implementation uses hash map. */
10
11 typedef enum Hash_Map_Size {
12     HASH_MAP_SIZE_2,
13     HASH_MAP_SIZE_5,
14     HASH_MAP_SIZE_11,
15     HASH_MAP_SIZE_23,
16     HASH_MAP_SIZE_47,
17     HASH_MAP_SIZE_97,
18     HASH_MAP_SIZE_197,
19     HASH_MAP_SIZE_397,
20     HASH_MAP_SIZE_797,
21     HASH_MAP_SIZE_1_597,
22     HASH_MAP_SIZE_3_203,
23     HASH_MAP_SIZE_6_421,
24     HASH_MAP_SIZE_12_853,
25     HASH_MAP_SIZE_25_717,
26     HASH_MAP_SIZE_51_437,
27     HASH_MAP_SIZE_102_877,
28     HASH_MAP_SIZE_205_759,
29     HASH_MAP_SIZE_411_527,
30     HASH_MAP_SIZE_823_117,
31     HASH_MAP_SIZE_1_646_237,
32     HASH_MAP_SIZE_3_292_489,
33     HASH_MAP_SIZE_6_584_983,
34     HASH_MAP_SIZE_13_169_977,
35     HASH_MAP_SIZE_26_339_969,
36     HASH_MAP_SIZE_52_679_969,
37     HASH_MAP_SIZE_105_359_939,
38     HASH_MAP_SIZE_210_719_881,
39     HASH_MAP_SIZE_421_439_783,
40     HASH_MAP_SIZE_842_879_579
41 } Hash_Map_Size;
42
43 #define HASH_MAP_PRIME_LIST_SIZE (HASH_MAP_SIZE_842_879_579 + 1)
44
45 #define HASH_MAP_MAX_LOAD_FACTOR 0.75f
46
47 #define HASH_MAP_MIN_LOAD_FACTOR (HASH_MAP_MAX_LOAD_FACTOR / 4.0f)
48
49 extern const Uns HASH_MAP_PRIME_LIST[HASH_MAP_PRIME_LIST_SIZE];
50
51 typedef struct Hash_Map_Slot {
52     Single_List_Node neighbor;
53     Uns hash_code;
54 } Hash_Map_Slot;
55
56 static inline Uns hash_map_slot_get_hash_code(Hash_Map_Slot *s)
57 {
58     return s->hash_code;
59 }
60
61 #define HASH_MAP_SLOT_OF(list_node) \
62     SINGLE_LIST_ENTRY(list_node, Hash_Map_Slot, neighbor)
63
64 /* Compare search_obj with tree_node in some specified way.
65  * Return true if search_obj == tree_node.
66  * Otherwise return false. */
67 typedef bool (*Hash_Map_Comparator)(void *search_obj,
68     Hash_Map_Slot *map_slot);
69
70 typedef struct Hash_Map {
71     Single_List *slots;
72     Hash_Map_Comparator comparator;
73     Int size_idx;

```



```

74     Int initial_size;
75     Uns num_slots;
76     Uns num_elements;
77 } Hash_Map;
78
79 #define HASH_MAP_DEFAULT_SIZE HASH_MAP_SIZE_197
80
81 #define INITIAL_NUM_SLOTS 1
82 #define INITIAL_NUM_ELEMENTS INITIAL_NUM_SLOTS
83
84 #define HASH_MAP_GET_INIT_SIZE_IDX(idx) (idx - 1)
85
86 #define HASH_MAP_STATIC_INIT_SIZE(size_index, comp_fun) { \
87     .slots = NULL, \
88     .comparator = comp_fun, \
89     .size_idx = HASH_MAP_GET_INIT_SIZE_IDX(size_index), \
90     .initial_size = size_index, \
91     .num_slots = INITIAL_NUM_SLOTS, \
92     .num_elements = INITIAL_NUM_ELEMENTS \
93 }
94
95 #define HASH_MAP_STATIC_INIT(comp_fun) \
96     HASH_MAP_STATIC_INIT_SIZE(HASH_MAP_DEFAULT_SIZE, comp_fun)
97
98 #define HASH_MAP_INIT(comp_fun) \
99     ((Hash_Map)HASH_MAP_STATIC_INIT(comp_fun))
100
101 #define HASH_MAP_INIT_SIZE(size_index, comp_fun) \
102     ((Hash_Map)HASH_MAP_STATIC_INIT_SIZE(size_index, comp_fun))
103
104 #define HASH_MAP(hmname, comp_fun) \
105     Hash_Map hmname = HASH_MAP_STATIC_INIT(comp_fun)
106
107 #define HASH_MAP_SIZE(hmname, size_index, comp_fun) \
108     Hash_Map hmname = HASH_MAP_STATIC_INIT_SIZE(size_index, comp_fun)
109
110 #define HASH_MAP_ENTRY(slot, type_name, field_name) \
111     CONTAINER_OF(slot, type_name, field_name)
112
113 /* Insert a slot into the hash map.
114  * This is a multi map, so you can insert multiple slots
115  * with the same value. */
116 void hash_map_insert(Hash_Map *map, Hash_Map_Slot *slot, Uns hash_code);
117
118 bool hash_map_contains(Hash_Map *map, void *search_obj,
119     Uns search_hash_code);
120
121 /* Uses Hash_Map_Comparator to compare search_obj with slots.
122  * If Hash_Map_Comparator return 0 for some slot then remove the
123  * slot and return the slot else no slot is removed and returns NULL. */
124 Hash_Map_Slot *hash_map_remove(Hash_Map *map, void *search_obj,
125     Uns search_hash_code);
126
127 /* Uses Hash_Map_Comparator to compare search_obj with slots.
128  * If Hash_Map_Comparator return 0 for some slot then
129  * return the slot else returns NULL. */
130 Hash_Map_Slot *hash_map_get(Hash_Map *map, void *search_obj,
131     Uns search_hash_code);
132
133 typedef void (*Hash_Map_Destructor)(Hash_Map_Slot *slot);
134
135 /* Free memory allocated by the hash map.
136  * See hash_map_for_each_destroy(). */
137 static inline void hash_map_clear(Hash_Map *map)
138 {
139     free_mem(map->slots);
140     map->slots = NULL;
141     map->num_slots = INITIAL_NUM_SLOTS;
142     map->num_elements = INITIAL_NUM_ELEMENTS;
143     map->size_idx = HASH_MAP_GET_INIT_SIZE_IDX(map->initial_size);
144 }
145
146 /* Free memory allocated by the hash map and
147  * call destructor with each slot in the map to free the slot containers. */

```

```

148 void hash_map_for_each_destroy(Hash_Map *map, Hash_Map_Destructor destructor);
149
150 static inline Uns hash_map_size(Hash_Map *map)
151 {
152     return map->slots ? map->num_elements : 0;
153 }
154
155 // XXX - Use goto to break this loop.
156 #define HASH_MAP_FOR_EACH(map, slot) \
157     if ((map)->slots) \
158         for (Single_List_Node *__n, *__i = 0; \
159             PTR_TO_UINT(__i) < (map)->num_slots; \
160             __i = INT_TO_PTR(PTR_TO_UINT(__i) + 1)) \
161             SINGLE_LIST_FOR_EACH(&(map)->slots[PTR_TO_UINT(__i)], __n) \
162                 if ((slot = HASH_MAP_SLOT_OF(__n)) || !slot)
163
164 /* Get hash code of pointer which is sizeof(void *) aligned. */
165 static inline Uns hash_map_aligned_ptr_hash(void *p)
166 {
167     #if PTR_SIZE == 8
168     # define __HASH_SHIFT 3
169     #elif PTR_SIZE == 4
170     # define __HASH_SHIFT 2
171     #else
172     # define __HASH_SHIFT 1
173     #endif
174     return (Uns) (PTR_TO_INT(p) >> __HASH_SHIFT);
175 #undef __HASH_SHIFT
176 }
177
178 #endif // HASH_MAP_H

```

:

A.10.14 src/help_msg.c

```

1  #include <str.h>
2
3  CONST_STRING(help_msg,
4  #ifdef STDIN_INPUT_ENABLED
5  "Usage: %S [OPTION]... [FILE]...\n"
6  #else
7  "Usage: %S [OPTION]... FILE...\n"
8  #endif
9  " OPTION\n"
10 "  --help (-h)\n"
11 "    display this message\n"
12 "  --help=optimize\n"
13 "    display information about optimization options\n"
14 "  --help=warning\n"
15 "    display information about warning options\n"
16 "  --help=dump\n"
17 "    display information about options for dumping internal\n"
18 "    data structures\n"
19 "\n"
20 "  --output=OUTFILE (-o OUTFILE)\n"
21 "    write output to OUTFILE\n"
22 "  --main (-m)\n"
23 "    insert code that calls function main when linking program\n"
24 "  --ign-main\n"
25 "    ignore the --main (-m) option\n"
26 "  --lib-init\n"
27 "    put code in the global scope into an initialization section\n"
28 "    executed before the normal initialization section\n"
29 "  --ign-lib-init\n"
30 "    ignore the --lib-init option\n"
31 "  --thread\n"
32 "    include support for libvitaly threads\n"
33 "  --ign-thread\n"
34 "    ignore the --thread option\n"
35 "  --no-libvit (-x)\n"

```

```

36 "    do not link with libvitaly (standard library becomes inaccessible)\n"
37 "    --ign-no-libvitaly\n"
38 "    ignore the --no-libvitaly (-x) option\n"
39 "    --lib-path=PATH (-L PATH)\n"
40 "    add PATH to the list of paths with library files\n"
41 "    --lib=LIB (-l LIB)\n"
42 "    link against library LIB accessible through one of the library\n"
43 "    search paths\n"
44 "    --import-path=PATH (-i PATH)\n"
45 "    add PATH to the list of paths searched when looking for import files\n"
46 "    --max-msg=n\n"
47 "    do not produce more than n error and warning messages\n"
48 "    --stubborn\n"
49 "    keep compiling new input source files, even when errors are detected\n"
50 "    --ign-stubborn\n"
51 "    ignore the --stubborn option\n"
52 "    --compile-only (-c)\n"
53 "    assemble, compile and keep object files, but do not link\n"
54 "    --ign-compile-only\n"
55 "    ignore the --compile-only (-c) option\n"
56 "    --asm-only (-s)\n"
57 "    produce assembly output, do not compile or link\n"
58 "    --ign-asm-only\n"
59 "    ignore the --asm-only (-s) option\n"
60 "    --keep-obj (-k)\n"
61 "    keep generated object files\n"
62 "    --ign-keep-obj\n"
63 "    ignore the --keep-obj (-k) option\n"
64 "    --verbose (-v)\n"
65 "    print verbose output to standard output\n"
66 "    --ign-verbose\n"
67 "    ignore the --verbose (-v) option\n"
68 "    --recursive (-r)\n"
69 "    recursive compilation, automatically compile imported files\n"
70 "    --ign-recursive\n"
71 "    ignore the --recursive (-r) option\n"
72 "    --gen-viti (-I)\n"
73 "    generate a .viti file used as import file instead of the\n"
74 "    corresponding .vit file\n"
75 "    --ign-gen-viti\n"
76 "    ignore the --gen-viti (-I) option\n"
77 );
78
79 CONST_STRING(dump_help_msg,
80 "Options for dumping internal data structures\n"
81 "  --dump=OPTION[,OPTION]...\n"
82 "  OPTION\n"
83 "    parse-tree\n"
84 "    dump XML parse tree\n"
85 "    parse-tree-graph\n"
86 "    dump PDF parse tree\n"
87 "    symbol-table\n"
88 "    dump ACSII symbol table\n"
89 "    symbol-table-graph\n"
90 "    dump PDF symbol table\n"
91 "    init-ic\n"
92 "    dump initial intermediate code representation\n"
93 "    norm-addr-ic\n"
94 "    dump intermediate code representation right after initial\n"
95 "    normalization of addressing instructions\n"
96 "    const-prop-ic\n"
97 "    dump intermediate code after each constant propagation pass\n"
98 "    instr-elim-ic\n"
99 "    dump intermediate code after dead instruction elimination passes\n"
100 "    def-to-use-ic\n"
101 "    dump intermediate code after def-to-use optimization pass\n"
102 "    norm-x86-32-ic\n"
103 "    dump intermediate code after instructions have been converted to\n"
104 "    x86-32 compatible instructions\n"
105 "    unused-mov-ic\n"
106 "    dump intermediate code after elimination of unused mov instructions\n"
107 "    reg-alloc-x86-32-ic\n"
108 "    dump intermediate code right after register allocation\n"
109 "    reg-vars-liveness-ic\n"

```

```

110 "    dump liveness analysis before deciding which variables should\n"
111 "    stay in registers\n"
112 "    reg-vars-ic\n"
113 "    dump liveness analysis after register variables have been chosen\n"
114 "    init-liveness-x86-32-ic\n"
115 "    dump liveness analysis before register allocation\n"
116 "    liveness-x86-32-ic\n"
117 "    dump liveness analysis right before assigning pseudo registers to\n"
118 "    x86-32 registers\n"
119 "    reg-alloc-x86-32-ic\n"
120 "    dump intermediate code right after register allocation\n"
121 "    warn-uninit-liveness-ic\n"
122 "    dump liveness analysis before locating variables which might be\n"
123 "    uninitialized before use\n"
124 "    final-x86-32-ic\n"
125 "    dump final intermediate code representation before emitting\n"
126 "    machine code\n"
127 "    c-header\n"
128 "    dump C header file with vitaly record declarations as C structs\n"
129 "    asm\n"
130 "    dump assembly source file before assembling\n"
131 ");
132
133 CONST_STRING(warn_help_msg,
134 "Options for enabling and disabling warnings\n"
135 " --warning=OPTION[,OPTION]... (-w OPTION[,OPTION]...) \n"
136 "  OPTION\n"
137 "  all\n"
138 "    enable all warning options\n"
139 "  ign-all\n"
140 "    disable all warning options\n"
141 "\n"
142 "  is-error\n"
143 "    treat warnings as errors\n"
144 "  ign-is-error\n"
145 "    ignore the is-error warning option\n"
146 "\n"
147 "  no-finalize\n"
148 "    show warning when extended records doesn't have a finalize function\n"
149 "  ign-no-finalize\n"
150 "    ignore the no-finalize warning option\n"
151 "  implicit-cast [enabled by default]\n"
152 "    give warning about implicit type casts\n"
153 "  ign-implicit-cast\n"
154 "    ignore the implicit-cast warning option\n"
155 "  ref-compare [enabled by default]\n"
156 "    give warning when comparing reference types\n"
157 "  ign-ref-compare\n"
158 "    ignore the ref-compare warning option\n"
159 "  overflow [enabled by default]\n"
160 "    give warning when overflow is detected\n"
161 "  ign-overflow\n"
162 "    ignore the overflow warning option\n"
163 "  div-zero [enabled by default]\n"
164 "    give warning when division by zero is detected\n"
165 "  ign-div-zero\n"
166 "    ignore the div-zero warning option\n"
167 "  uninitialized [enabled by default]\n"
168 "    give warning when variables might be uninitialized before use\n"
169 "  ign-uninitialized\n"
170 "    ignore the unitialized warning option\n"
171 ");
172
173 CONST_STRING(opt_help_msg,
174 "Options for enabling and disabling optimizations\n"
175 " --optimize=OPTION[,OPTION]... (-O OPTION[,OPTION]...) \n"
176 "  OPTION\n"
177 "  all [enabled by default]\n"
178 "    enable all optimizations\n"
179 "  ign-all\n"
180 "    disable all optimization options\n"
181 "\n"
182 "  const-prop [enabled by default]\n"
183 "    do constant propagation optimization\n"

```

```

184 "    ign-const-prop\n"
185 "    ignore the const-prop optimization option\n"
186 "    instr-elim [enabled by default]\n"
187 "    do instruction elimination optimization (dead code elimination)\n"
188 "    ign-instr-elim\n"
189 "    ignore the instr-elim optimization option\n"
190 "    unused-mov [enabled by default]\n"
191 "    try to minimize the number register to register mov instructions\n"
192 "    ign-unused-mov\n"
193 "    ignore the unused-mov optimization option\n"
194 "    unused-set [enabled by default]\n"
195 "    try to eliminate useless set instructions\n"
196 "    ign-unused-set\n"
197 "    ignore the unused-set optimization option\n"
198 "    reg-vars [enabled by default]\n"
199 "    try to keep variables in registers when possible\n"
200 "    ign-reg-vars\n"
201 "    ignore the reg-vars optimization option\n"
202 "    def-to-use [enabled by default]\n"
203 "    move definitions of variables closer to where they are used\n"
204 "    ign-def-to-use\n"
205 "    ignore the def-to-use optimization option\n"
206 "    func-access [enabled by default]\n"
207 "    detect information about which variables functions are using\n"
208 "    ign-func-access\n"
209 "    ignore the func-access optimization option\n"
210 );

```

:

A.10.15 src/help_msg.h

```

1  #ifndef HELP_MSG_H
2  #define HELP_MSG_H
3
4  #include <str.h>
5
6  extern Const_String help_msg;
7  extern Const_String dump_help_msg;
8  extern Const_String warn_help_msg;
9  extern Const_String opt_help_msg;
10
11 #endif // HELP_MSG_H

```

:

A.10.16 src/io.c

```

1  #include <printf.h>
2  #include <main.h>
3  #include <unistd.h>
4  #include <sys/stat.h>
5
6  static int string_printf_function(FILE *stream,
7      const struct printf_info *info, const void *const *args)
8  {
9      const Const_String s = *((const Const_String *) (args[0]));
10     return fprintf(stream, "%*s",
11         info->left ? -info->width : info->width,
12         string_to_cstr(s));
13 }
14
15 static int string_printf_arginfo(const struct printf_info *info UNUSED,
16     size_t n, int *argtypes, int *size UNUSED)
17 {
18     if (n > 0)
19         argtypes[0] = PA_POINTER;

```

```

20     return 1;
21 }
22
23 static int uns_printf_function(FILE *stream,
24     const struct printf_info *info, const void *const *args)
25 {
26     const Uns u = *((const Uns *) (args[0]));
27     return fprintf(stream, "%*" PRIu32,
28         info->left ? -info->width : info->width,
29         u);
30 }
31
32 static int uns_printf_arginfo(const struct printf_info *info UNUSED,
33     size_t n, int *argtypes, int *size UNUSED)
34 {
35     if (n > 0)
36         argtypes[0] = PA_INT;
37     return 1;
38 }
39
40 static int int_printf_function(FILE *stream,
41     const struct printf_info *info, const void *const *args)
42 {
43     const Int i = *((const Int *) (args[0]));
44     return fprintf(stream, "%*" PRId32,
45         info->left ? -info->width : info->width,
46         i);
47 }
48
49 static int int_printf_arginfo(const struct printf_info *info UNUSED,
50     size_t n, int *argtypes, int *size UNUSED)
51 {
52     if (n > 0)
53         argtypes[0] = PA_INT;
54     return 1;
55 }
56
57 static int file_location_printf_function(FILE *stream,
58     const struct printf_info *info, const void *const *args)
59 {
60     const File_Location *loc = *((const File_Location **) (args[0]));
61     const char *file_name = string_to_cstr(file_location_get_file_name(loc));
62     if (loc->line == 0 || loc->column == 0)
63         return fprintf(stream, "%*s",
64             info->left ? -info->width : info->width,
65             file_name);
66     else
67         return fprintf(stream, "%*s:%" PRIu32 ":%" PRIu32,
68             info->left ? -info->width : info->width,
69             file_name,
70             loc->line,
71             loc->column);
72 }
73
74 static int file_location_printf_arginfo(const struct printf_info *info UNUSED,
75     size_t n, int *argtypes, int *size UNUSED)
76 {
77     if (n > 0)
78         argtypes[0] = PA_POINTER;
79     return 1;
80 }
81
82 void io_handler_constructor() CONSTRUCTOR;
83 void io_handler_constructor()
84 {
85     register_printf_specifier(String_Printf_Specifier, string_printf_function,
86         string_printf_arginfo);
87     register_printf_specifier(Uns_Printf_Specifier, uns_printf_function,
88         uns_printf_arginfo);
89     register_printf_specifier(Int_Printf_Specifier, int_printf_function,
90         int_printf_arginfo);
91     register_printf_specifier(File_Location_Printf_Specifier,
92         file_location_printf_function, file_location_printf_arginfo);
93 }

```

```

94
95
96 static const char FATAL_ERROR_PREFIX[] = "(fatal error) ";
97
98 /* Do not call any function that might call fatal_error() from fatal_error().
99  * The function will try to tell the user that something went wrong before
100  * quickly aborting. */
101 void fatal_error(Const_String format, ...)
102 {
103     VA_SETUP(vl, format);
104     if (fprintf(stderr, "%s: %s",
105                 string_to_cstr(cmdopts.vitaly_program_name),
106                 FATAL_ERROR_PREFIX) < 0 ||
107         vfprintf(stderr, string_to_cstr(format), vl) < 0) {
108         printf("\n");
109         printf("%s: %s",
110                string_to_cstr(cmdopts.vitaly_program_name),
111                FATAL_ERROR_PREFIX);
112         vprintf(string_to_cstr(format), vl);
113     }
114     VA_END(vl);
115     exit_failure();
116 }
117
118 void print_vamessage(Const_String format, va_list vl)
119 {
120     if (vprintf(string_to_cstr(format), vl) < 0)
121         fatal_error(S("was unable to print to stdout\n"));
122 }
123
124 void print_message(Const_String format, ...)
125 {
126     VA_SETUP(vl, format);
127     print_vamessage(format, vl);
128     VA_END(vl);
129 }
130
131 void file_print_vamessage(FILE *stream, Const_String format, va_list vl)
132 {
133     if (vfprintf(stream, string_to_cstr(format), vl) < 0)
134         fatal_error(S("was unable to print to file stream\n"));
135 }
136
137 void file_print_message(FILE *stream, Const_String format, ...)
138 {
139     VA_SETUP(vl, format);
140     file_print_vamessage(stream, format, vl);
141     VA_END(vl);
142 }
143
144 void print_error(Const_String format, ...)
145 {
146     VA_SETUP(vl, format);
147     String new_fmt = string_from_format(S("%S: %S"),
148                                         cmdopts.vitaly_program_name, format);
149     file_print_vamessage(stderr, new_fmt, vl);
150     string_destroy(new_fmt);
151     VA_END(vl);
152 }
153
154 time_t file_get_mtime(Const_String fname)
155 {
156     struct stat buf;
157     if (stat(string_to_cstr(fname), &buf)) {
158         return (time_t) -1;
159     }
160     return buf.st_mtime;
161 }
162
163 bool file_access(Const_String fname, int mode)
164 {
165     return !access(string_to_cstr(fname), mode);
166 }
167

```

```

168 bool file_access_read(Const_String fname)
169 {
170     return file_access(fname, R_OK);
171 }
172
173 bool file_unlink(Const_String f)
174 {
175     return unlink(string_to_cstr(f)) != -1;
176 }

```

:

A.10.17 src/io.h

```

1  #ifndef IO_HANDLER_H
2  #define IO_HANDLER_H
3
4  #include <str.h>
5  #include <std_define.h>
6  #include <stdio.h>
7  #include <report.h>
8
9  #define STRING_PRINTF_SPECIFIER 'S'
10 #define UNS_PRINTF_SPECIFIER 'U'
11 #define INT_PRINTF_SPECIFIER 'D'
12 #define FILE_LOCATION_PRINTF_SPECIFIER 'F'
13
14 NORETURN void fatal_error(Const_String format, ...);
15
16 void print_vamessage(Const_String, va_list vl);
17
18 void print_message(Const_String format, ...);
19
20 void file_print_vamessage(FILE *stream, Const_String format, va_list vl);
21
22 void file_print_message(FILE *stream, Const_String format, ...);
23
24 void print_error(Const_String format, ...);
25
26 /* See fopen ($ man fopen). */
27 static inline FILE *file_open(Const_String fname, Const_String mode)
28 {
29     return fopen(string_to_cstr(fname), string_to_cstr(mode));
30 }
31
32 /* See fclose ($ man fclose). */
33 static inline int file_close(FILE *file)
34 {
35     return fclose(file);
36 }
37
38 /* See man page for unlink(), returns true on success. */
39 bool file_unlink(Const_String f);
40
41 /* Unlink a temp file. Report warning on failure. */
42 static inline void file_unlink_temp(Const_String file)
43 {
44     if (!file_unlink(file))
45         report_warning(file, S("unable to remove temporary file "
46             QFY("%S") " [%m]"), file);
47 }
48
49 /* check errno see stat ($ man stat). */
50 time_t file_get_mtime(Const_String fname);
51
52 /* check errno see access ($ man access). */
53 bool file_access(Const_String fname, int mode);
54
55 /* see file_access */
56 bool file_access_read(Const_String fname);
57

```

```

58 static inline Int sys_cmd(Const_String cmd)
59 {
60     extern void inc_error_count();
61     int sys_ret = system(string_to_cstr(cmd));
62     if (sys_ret == -1)
63         fatal_error(S("unable to execute command:\n\t%S\n"), cmd);
64     if (sys_ret)
65         inc_error_count();
66     return sys_ret;
67 }
68
69 #endif // IO_HANDLER_H

```

:

A.10.18 src/pointer_hash.c

```

1  #include <pointer_hash.h>
2
3  bool pointer_hash_map_compare(void *search_obj, Hash_Map_Slot *map_slot)
4  {
5      Pointer_Slot *ms = POINTER_SLOT_OF(map_slot);
6      return search_obj == ms->key;
7  }
8
9  void pointer_hash_map_destructor(Hash_Map_Slot *slot)
10 {
11     free_mem(POINTER_SLOT_OF(slot));
12 }

```

:

A.10.19 src/pointer_hash.h

```

1  #ifndef POINTER_HASH_H
2  #define POINTER_HASH_H
3
4  #include <hash_map.h>
5
6  typedef struct Pointer_Slot {
7      Hash_Map_Slot slot;
8      void *key;
9      void *val;
10 } Pointer_Slot;
11
12 #define POINTER_SLOT_OF(s) CONTAINER_OF(s, Pointer_Slot, slot)
13
14 bool pointer_hash_map_compare(void *search_obj, Hash_Map_Slot *map_slot);
15
16 void pointer_hash_map_destructor(Hash_Map_Slot *slot);
17
18 #endif

```

:

A.10.20 src/rb_tree.c

```

1  #include <rb_tree.h>
2
3  /*
4   * Right rotate:
5   *      p          p          p          p
6   *     / \        / \        / \        / \
7   *    n   t      t   n      n   t      t   n

```

```

8  * / \ => / \ or / \ => / \
9  * t   l n   t   l n
10 * / \   /   / \   /
11 * l r   r   l r   r
12 */
13 static ALWAYS_INLINE void rb_tree_right_rotate(Rb_Tree *tree, Rb_Tree_Node *node)
14 {
15     Rb_Tree_Node *tmp = node->left, *parent = node->parent;
16     node->left = tmp->right;
17     tmp->right->parent = node;
18     tmp->parent = parent;
19     if (parent == RB_TREE_NULL(tree))
20         tree->root = tmp;
21     else if (node == parent->right)
22         parent->right = tmp;
23     else
24         parent->left = tmp;
25     tmp->right = node;
26     node->parent = tmp;
27 }
28
29 // Left rotate (reverse of right rotate).
30 static ALWAYS_INLINE void rb_tree_left_rotate(Rb_Tree *tree, Rb_Tree_Node *node)
31 {
32     Rb_Tree_Node *tmp = node->right, *parent = node->parent;
33     node->right = tmp->left;
34     tmp->left->parent = node;
35     tmp->parent = parent;
36     if (parent == RB_TREE_NULL(tree))
37         tree->root = tmp;
38     else if (node == parent->left)
39         parent->left = tmp;
40     else
41         parent->right = tmp;
42     tmp->left = node;
43     node->parent = tmp;
44 }
45
46 void rb_tree_insert(Rb_Tree *tree, Rb_Tree_Node *node)
47 {
48     Rb_Tree_Node *tmp, *gparent, *parent = tree->root;
49     assert(tree->size < UINT32_MAX);
50     ++tree->size;
51
52     if (parent == RB_TREE_NULL(tree)) {
53         tree->root = node;
54         node->color = RB_TREE_BLACK;
55         node->left = node->right = node->parent = RB_TREE_NULL(tree);
56         return;
57     }
58
59     for (;;) {
60         if (tree->comparator(node, parent)) {
61             if (parent->left != RB_TREE_NULL(tree)) {
62                 parent = parent->left;
63             } else {
64                 parent->left = node;
65                 break;
66             }
67         } else {
68             if (parent->right != RB_TREE_NULL(tree)) {
69                 parent = parent->right;
70             } else {
71                 parent->right = node;
72                 break;
73             }
74         }
75     }
76
77     node->color = RB_TREE_RED;
78     node->left = node->right = RB_TREE_NULL(tree);
79     node->parent = parent;
80     if (parent->color == RB_TREE_BLACK)
81         return;

```

```

82
83     for (;;) {
84         gparent = parent->parent;
85
86         if (parent == gparent->left) {
87             if (gparent->right->color == RB_TREE_RED) {
88                 // Case 1.
89                 gparent->right->color = RB_TREE_BLACK;
90                 parent->color = RB_TREE_BLACK;
91                 gparent->color = RB_TREE_RED;
92                 // Fall through to end of loop.
93             } else {
94                 if (node == parent->right) {
95                     tmp = parent->right;
96
97                     // Case 2 (left rotate):
98                     parent->right = tmp->left;
99                     tmp->left->parent = parent;
100                    gparent->left = tmp;
101                    tmp->parent = gparent;
102                    tmp->left = parent;
103                    parent->parent = tmp;
104
105                    node->color = RB_TREE_BLACK;
106                } else {
107                    parent->color = RB_TREE_BLACK;
108                }
109                // Case 3.
110                gparent->color = RB_TREE_RED;
111                rb_tree_right_rotate(tree, gparent);
112                break;
113            }
114        } else { // Then: parent == gparent->right
115            if (gparent->left->color == RB_TREE_RED) {
116                // Case 1.
117                gparent->left->color = RB_TREE_BLACK;
118                parent->color = RB_TREE_BLACK;
119                gparent->color = RB_TREE_RED;
120                // Fall through to end of loop.
121            } else {
122                if (node == parent->left) {
123                    tmp = parent->left;
124
125                    // Case 2 (right rotate).
126                    parent->left = tmp->right;
127                    tmp->right->parent = parent;
128                    gparent->right = tmp;
129                    tmp->parent = gparent;
130                    tmp->right = parent;
131                    parent->parent = tmp;
132
133                    node->color = RB_TREE_BLACK;
134                } else {
135                    parent->color = RB_TREE_BLACK;
136                }
137                // Case 3.
138                gparent->color = RB_TREE_RED;
139                rb_tree_left_rotate(tree, gparent);
140                break;
141            }
142        }
143
144        if (gparent->parent->color == RB_TREE_RED) {
145            parent = gparent->parent;
146            node = gparent;
147        } else {
148            tree->root->color = RB_TREE_BLACK;
149            break;
150        }
151    }
152 }
153 }
154
155 void rb_tree_for_each(Rb_Tree *tree, RB_Tree_Callback callback)

```

```

156 {
157     inline void rb_tree_inorder_call(Rb_Tree_Node *node)
158     {
159         if (node->left != RB_TREE_NULL(tree))
160             rb_tree_inorder_call(node->left);
161         callback(node);
162         if (node->right != RB_TREE_NULL(tree))
163             rb_tree_inorder_call(node->right);
164     }
165
166     if (tree->root != RB_TREE_NULL(tree))
167         rb_tree_inorder_call(tree->root);
168 }
169
170 void rb_tree_for_each_reversed(Rb_Tree *tree, Rb_Tree_Callback callback)
171 {
172     inline void rb_tree_inorder_call(Rb_Tree_Node *node)
173     {
174         if (node->right != RB_TREE_NULL(tree))
175             rb_tree_inorder_call(node->right);
176         callback(node);
177         if (node->left != RB_TREE_NULL(tree))
178             rb_tree_inorder_call(node->left);
179     }
180
181     if (tree->root != RB_TREE_NULL(tree))
182         rb_tree_inorder_call(tree->root);
183 }
184
185 void rb_tree_for_each_destroy(Rb_Tree *tree, Rb_Tree_Destructor destructor)
186 {
187     inline void rb_tree_delete_nodes(Rb_Tree_Node *node)
188     {
189         if (node->left != RB_TREE_NULL(tree))
190             rb_tree_delete_nodes(node->left);
191         if (node->right != RB_TREE_NULL(tree))
192             rb_tree_delete_nodes(node->right);
193         destructor(node);
194     }
195
196     if (tree->root != RB_TREE_NULL(tree)) {
197         rb_tree_delete_nodes(tree->root);
198         tree->size = 0;
199         tree->root = RB_TREE_NULL(tree);
200         tree->color = RB_TREE_BLACK;
201     }
202 }
203
204 static ALWAYS_INLINE void rb_tree_transplant(Rb_Tree *tree, Rb_Tree_Node *node,
205 Rb_Tree_Node *child)
206 {
207     if (node->parent == RB_TREE_NULL(tree))
208         tree->root = child;
209     else if (node == node->parent->left)
210         node->parent->left = child;
211     else
212         node->parent->right = child;
213     child->parent = node->parent;
214 }
215
216 void rb_tree_remove(Rb_Tree *tree, Rb_Tree_Node *node)
217 {
218     Rb_Tree_Node *transp, *sibling, *parent, *tmp;
219     Rb_Tree_Color del_color;
220     --tree->size;
221
222     if (node->left == RB_TREE_NULL(tree)) {
223         del_color = node->color;
224         transp = node->right;
225         rb_tree_transplant(tree, node, transp);
226     } else if (node->right == RB_TREE_NULL(tree)) {
227         del_color = node->color;
228         transp = node->left;
229         rb_tree_transplant(tree, node, transp);

```

```

230     } else {
231         tmp = node->right;
232         while (tmp->left != RB_TREE_NULL(tree))
233             tmp = tmp->left;
234         del_color = tmp->color;
235         transp = tmp->right;
236         if (tmp->parent == node) {
237             transp->parent = tmp;
238         } else {
239             tmp->parent->left = transp;
240             transp->parent = tmp->parent;
241             tmp->right = node->right;
242             tmp->right->parent = tmp;
243         }
244         rb_tree_transplant(tree, node, tmp);
245         tmp->left = node->left;
246         tmp->left->parent = tmp;
247         tmp->color = node->color;
248     }
249
250     if (del_color == RB_TREE_RED)
251         return;
252     if (transp->color == RB_TREE_RED || transp == tree->root) {
253         transp->color = RB_TREE_BLACK;
254         return;
255     }
256
257     for ( ; ; ) {
258         /*
259          * Loop invariant:
260          * 1. transp != tree->root.
261          * 2. transp->color == RB_TREE_BLACK.
262          * 3. The simple paths from tree->root through transp to leaf
263          *    contains one less black node than the other simple paths
264          *    from root to leaf.
265          */
266         parent = transp->parent;
267         if (transp != parent->right) {
268             // Then: transp == parent->left.
269             sibling = parent->right;
270             if (sibling->color == RB_TREE_RED) {
271                 // Case 1.
272                 sibling->color = RB_TREE_BLACK;
273                 parent->color = RB_TREE_RED;
274                 rb_tree_left_rotate(tree, parent);
275                 sibling = parent->right;
276             }
277             if (sibling->right->color == RB_TREE_BLACK) {
278                 tmp = sibling->left;
279                 if (tmp->color == RB_TREE_BLACK) {
280                     // Case 2.
281                     sibling->color = RB_TREE_RED;
282                     if (parent->color == RB_TREE_RED) {
283                         parent->color = RB_TREE_BLACK;
284                     } else if (parent != tree->root) {
285                         transp = parent;
286                         continue;
287                     }
288                     break;
289                 }
290                 tmp->color = RB_TREE_BLACK;
291                 sibling->color = RB_TREE_RED;
292
293                 // Case 3 (right rotate):
294                 sibling->left = tmp->right;
295                 tmp->right->parent = sibling;
296                 if (parent == RB_TREE_NULL(tree))
297                     tree->root = tmp;
298                 else
299                     parent->right = tmp;
300                 tmp->parent = parent;
301                 tmp->right = sibling;
302                 sibling->parent = tmp;
303             }

```

```

304         sibling = tmp;
305     }
306     // Case 4.
307     sibling->color = parent->color;
308     parent->color = RB_TREE_BLACK;
309     sibling->right->color = RB_TREE_BLACK;
310     rb_tree_left_rotate(tree, parent);
311     break;
312 } else {
313     // Then: transp == parent->right.
314     sibling = parent->left;
315     if (sibling->color == RB_TREE_RED) {
316         // Case 1.
317         sibling->color = RB_TREE_BLACK;
318         parent->color = RB_TREE_RED;
319         rb_tree_right_rotate(tree, parent);
320         sibling = parent->left;
321     }
322     if (sibling->left->color == RB_TREE_BLACK) {
323         tmp = sibling->right;
324         if (tmp->color == RB_TREE_BLACK) {
325             // Case 2.
326             sibling->color = RB_TREE_RED;
327             if (parent->color == RB_TREE_RED) {
328                 parent->color = RB_TREE_BLACK;
329             } else if (parent != tree->root) {
330                 transp = parent;
331                 continue;
332             }
333             break;
334         }
335         tmp->color = RB_TREE_BLACK;
336         sibling->color = RB_TREE_RED;
337
338         // Case 3 (left rotate):
339         sibling->right = tmp->left;
340         tmp->left->parent = sibling;
341         if (parent == RB_TREE_NULL(tree))
342             tree->root = tmp;
343         else
344             parent->left = tmp;
345         tmp->parent = parent;
346         tmp->left = sibling;
347         sibling->parent = tmp;
348
349         sibling = tmp;
350     }
351     // Case 4.
352     sibling->color = parent->color;
353     parent->color = RB_TREE_BLACK;
354     sibling->left->color = RB_TREE_BLACK;
355     rb_tree_right_rotate(tree, parent);
356     break;
357 }
358 }
359 }

```

:

A.10.21 src/rb_tree.h

```

1  #ifndef RB_TREE_H
2  #define RB_TREE_H
3
4  #include <std_defines.h>
5  #include <alloc.h>
6
7  typedef struct Rb_Tree_Node Rb_Tree_Node;
8
9  /* Should return true when: search_node <= tree_node,
10   * and return false when: search_node > tree_node

```

```

11  * Where search_node is the data we are inserting for and tree_node
12  * is a node from the tree. */
13  typedef bool (*Rb_Tree_Node_Comparator) (Rb_Tree_Node *search_node,
14      Rb_Tree_Node *tree_node);
15
16  #define RB_TREE_INTERFACE \
17      Rb_Tree_Color color; \
18      Rb_Tree_Node *parent
19
20  typedef enum Rb_Tree_Color {
21      RB_TREE_RED,
22      RB_TREE_BLACK
23  } Rb_Tree_Color;
24
25  struct Rb_Tree_Node {
26      RB_TREE_INTERFACE;
27      Rb_Tree_Node *left;
28      Rb_Tree_Node *right;
29  };
30
31  typedef struct Rb_Tree {
32      RB_TREE_INTERFACE;
33      Rb_Tree_Node *root;
34      Rb_Tree_Node_Comparator comparator;
35      Uns size;
36  } Rb_Tree;
37
38  #define RB_TREE_NULL(tree_ptr) ((Rb_Tree_Node *) (tree_ptr))
39
40  #define RB_TREE_STATIC_INIT(tree_name, comp_fun) { \
41      .root = RB_TREE_NULL(&(tree_name)), \
42      .parent = RB_TREE_NULL(&(tree_name)), \
43      .comparator = comp_fun, \
44      .color = RB_TREE_BLACK, \
45      .size = 0 \
46  }
47
48  #define RB_TREE_INIT(tree_name, comp_fun) \
49      ((Rb_Tree)RB_TREE_STATIC_INIT(tree_name, comp_fun))
50
51  #define RB_TREE(tree, comp_fun) \
52      Rb_Tree tree = RB_TREE_STATIC_INIT(tree, comp_fun)
53
54  #define RB_TREE_ENTRY(node_ptr, container_type, node_member_name) \
55      CONTAINER_OF(node_ptr, container_type, node_member_name)
56
57  void rb_tree_insert(Rb_Tree *tree, Rb_Tree_Node *node);
58
59  static inline bool rb_tree_is_empty(const Rb_Tree *tree)
60  {
61      return tree->root == RB_TREE_NULL(tree);
62  }
63
64  static inline Uns rb_tree_size(Rb_Tree *tree)
65  {
66      return tree->size;
67  }
68
69  /* Compare arg with tree_node in some specified way.
70   * Return -1 if arg < tree_node.
71   * Return 1 if arg > tree_node.
72   * Return 0 if arg == tree_node */
73  typedef Int (*Rb_Tree_Search_Comparator) (void *arg, Rb_Tree_Node *tree_node);
74
75  /* Search the tree using comparator until comparator returns 0.
76   * Return the node which cause comparator to return 0.
77   * If comparator never returns 0 then rb_tree_search returns NULL. */
78  static inline Rb_Tree_Node *rb_tree_search(Rb_Tree *tree,
79      Rb_Tree_Search_Comparator comparator, void *comparator_arg)
80  {
81      for (Rb_Tree_Node *n = tree->root; n != RB_TREE_NULL(tree);) {
82          Int res = comparator(comparator_arg, n);
83          if (res < 0)
84              n = n->left;

```

```

85     else if (res > 0)
86         n = n->right;
87     else
88         return n;
89     }
90     return NULL;
91 }
92
93 /* Return true if data is in the tree. */
94 static inline bool rb_tree_contains(Rb_Tree *tree,
95     Rb_Tree_Search_Comparator comparator, void *comparator_arg)
96 {
97     return rb_tree_search(tree, comparator, comparator_arg);
98 }
99
100 typedef void (*Rb_Tree_Destructor)(Rb_Tree_Node *node);
101
102 /* Remove node from the tree. */
103 void rb_tree_remove(Rb_Tree *tree, Rb_Tree_Node *node);
104
105 /* Search for a node where comparator returns 0.
106  * If such a node is found remove the node from the tree and return the node.
107  * Otherwise return NULL. */
108 static inline Rb_Tree_Node *rb_tree_search_remove(Rb_Tree *tree,
109     Rb_Tree_Search_Comparator comparator, void *comparator_arg)
110 {
111     Rb_Tree_Node *n = rb_tree_search(tree, comparator, comparator_arg);
112     if (n)
113         rb_tree_remove(tree, n);
114     return n;
115 }
116
117 typedef void (*RB_Tree_Callback)(Rb_Tree_Node *node);
118
119 /* Inorder tree walk, call callback for each node in the tree. */
120 void rb_tree_for_each(Rb_Tree *tree, RB_Tree_Callback callback);
121
122 /* Reversed inorder tree walk, call callback for each node in the tree. */
123 void rb_tree_for_each_reversed(Rb_Tree *tree, RB_Tree_Callback callback);
124
125 void rb_tree_for_each_destroy(Rb_Tree *tree, Rb_Tree_Destructor destructor);
126
127 static inline Rb_Tree *rb_tree_alloc(Rb_Tree_Node_Comparator comparator)
128 {
129     Rb_Tree *t = ALLOC_NEW(Rb_Tree);
130     *t = RB_TREE_INIT(*t, comparator);
131     return t;
132 }
133
134 static inline void rb_tree_destroy(Rb_Tree *tree,
135     Rb_Tree_Destructor destructor)
136 {
137     if (destructor)
138         rb_tree_for_each_destroy(tree, destructor);
139     free_mem(tree);
140 }
141
142 #endif // RB_TREE_H

```

:

A.10.22 src/report.c

```

1 #include <main.h>
2 #include <std_include.h>
3 #include <rb_tree.h>
4
5 typedef enum Report_Type {
6     REPORT_TYPE_WARNING,
7     REPORT_TYPE_ERROR
8 } Report_Type;

```



```

9
10 typedef struct Report {
11     Rb_Tree_Node rb_node;
12     String message;
13     File_Location location;
14     Report_Type type;
15 } Report;
16
17 static int32_t error_count;
18 static int32_t warning_count;
19 static int32_t reset_error_count;
20 static int32_t reset_warning_count;
21
22 #define REPORT_OF(node) RB_TREE_ENTRY(node, Report, rb_node)
23
24 static bool report_comparator(Rb_Tree_Node *search_node,
25                               Rb_Tree_Node *tree_node)
26 {
27     Report *se = REPORT_OF(search_node);
28     Report *te = REPORT_OF(tree_node);
29     Int res = file_location_cmp(&se->location, &te->location);
30     if (!res)
31         return se->type <= te->type;
32     return res == -1;
33 }
34
35 static inline Int search_comparator(void *location, Rb_Tree_Node *tree_node,
36                                     Report_Type search_type)
37 {
38     Report *rep = REPORT_OF(tree_node);
39     File_Location *loc = location;
40     Int res = file_location_cmp(loc, &rep->location);
41     if (!res) {
42         if (rep->type == search_type)
43             return 0;
44         if (rep->type < search_type)
45             return -1;
46         return 1;
47     }
48     return res;
49 }
50
51 static Int error_search_comparator(void *location, Rb_Tree_Node *tree_node)
52 {
53     return search_comparator(location, tree_node, REPORT_TYPE_ERROR);
54 }
55
56 static Int warning_search_comparator(void *location, Rb_Tree_Node *tree_node)
57 {
58     return search_comparator(location, tree_node, REPORT_TYPE_WARNING);
59 }
60
61 static RB_TREE(report_tree, report_comparator);
62
63 void report_reset()
64 {
65     reset_error_count = error_count;
66     reset_warning_count = warning_count;
67 }
68
69 bool report_exhausted()
70 {
71     return get_error_count() + get_warning_count() >= cmdopts.max_msg;
72 }
73
74 void inc_error_count()
75 {
76     ++error_count;
77 }
78
79 static inline void inc_warning_count()
80 {
81     ++warning_count;
82 }

```

```

83
84 int32_t get_error_count()
85 {
86     return error_count;
87 }
88
89 int32_t get_warning_count()
90 {
91     return warning_count;
92 }
93
94 bool was_error_reported()
95 {
96     return reset_error_count != get_error_count();
97 }
98
99 bool was_warning_reported()
100 {
101     return reset_warning_count != get_warning_count();
102 }
103
104 bool is_error_reported_here(File_Location *loc)
105 {
106     return rb_tree_contains(&report_tree, error_search_comparator, loc);
107 }
108
109 bool is_warning_reported_here(File_Location *loc)
110 {
111     if (cmdopts.warn_is_error)
112         return is_error_reported_here(loc);
113     return rb_tree_contains(&report_tree, warning_search_comparator, loc);
114 }
115
116 void report_vaerror(Const_String file_name, Const_String format, va_list vl)
117 {
118     File_Location loc = FILE_LOCATION_INIT(file_name, 0, 0);
119     report_vaerror_location(&loc, format, vl);
120 }
121
122 void report_error(Const_String file_name, Const_String format, ...)
123 {
124     VA_SETUP(vl, format);
125     report_vaerror(file_name, format, vl);
126     VA_END(vl);
127 }
128
129 void report_vaerror_location(File_Location *loc,
130     Const_String format, va_list vl)
131 {
132     if (report_exhausted())
133         return;
134
135     String new_format;
136
137     Report *rep = ALLOC_NEW(Report);
138     rep->location = *loc;
139     rep->type = REPORT_TYPE_ERROR;
140
141     new_format = string_from_format(S("%F: " ERROR_PREFIX "%S"), loc, format);
142     rep->message = string_from_vaformat(new_format, vl);
143     string_destroy(new_format);
144
145     rb_tree_insert(&report_tree, &rep->rb_node);
146
147     inc_error_count();
148 }
149
150 void report_error_location(File_Location *loc, Const_String format, ...)
151 {
152     VA_SETUP(vl, format);
153     report_vaerror_location(loc, format, vl);
154     VA_END(vl);
155 }
156

```

```

157 void report_vawarning(Const_String file_name, Const_String format, va_list vl)
158 {
159     File_Location loc = FILE_LOCATION_INIT(file_name, 0, 0);
160     report_vawarning_location(&loc, format, vl);
161 }
162
163 void report_warning(Const_String file_name, Const_String format, ...)
164 {
165     VA_SETUP(vl, format);
166     report_vawarning(file_name, format, vl);
167     VA_END(vl);
168 }
169
170 void report_vawarning_location(File_Location *loc,
171                               Const_String format, va_list vl)
172 {
173     if (cmdopts.warn_is_error) {
174         report_vaerror_location(loc, format, vl);
175         return;
176     }
177
178     if (report_exhausted())
179         return;
180
181     String new_format;
182
183     Report *rep = ALLOC_NEW(Report);
184     rep->location = *loc;
185     rep->type = REPORT_TYPE_WARNING;
186
187     new_format = string_from_format(S("%F: " WARNING_PREFIX "%S"),
188                                   loc, format);
189     rep->message = string_from_vformat(new_format, vl);
190     string_destroy(new_format);
191
192     rb_tree_insert(&report_tree, &rep->rb_node);
193
194     inc_warning_count();
195 }
196
197 void report_warning_location(File_Location *loc, Const_String format, ...)
198 {
199     VA_SETUP(vl, format);
200     report_vawarning_location(loc, format, vl);
201     VA_END(vl);
202 }
203
204 static void report_destructor(Rb_Tree_Node *rb_node)
205 {
206     Report *rep = REPORT_OF(rb_node);
207     string_destroy(rep->message);
208     free_mem(rep);
209 }
210
211 void clear_reports()
212 {
213     rb_tree_for_each_destroy(&report_tree, report_destructor);
214 }
215
216 static void show_report_callback(Rb_Tree_Node *rb_node)
217 {
218     Report *rep = REPORT_OF(rb_node);
219     file_print_message(stderr, rep->message);
220 }
221
222 void show_reports()
223 {
224     rb_tree_for_each(&report_tree, show_report_callback);
225 }
226
227 void report_print()
228 {
229     STRING(msg, "");
230     int32_t e = get_error_count();

```

```

231     int32_t w = get_warning_count();
232     if (w && e)
233         string_append_format(msg, S("%U error%s, %U warning%s\n"),
234             e, e > 1 ? "s" : "", w, w > 1 ? "s" : "");
235     else if (get_error_count())
236         string_append_format(msg, S("%U error%s\n"), e, e > 1 ? "s" : "");
237     else if (get_warning_count())
238         string_append_format(msg, S("%U warning%s\n"), w, w > 1 ? "s" : "");
239     else
240         goto out;
241
242     print_error(msg);
243
244 out:
245     string_clear(msg);
246 }

```

:

A.10.23 src/report.h

```

1  #ifndef ERROR_HANDLER_H
2  #define ERROR_HANDLER_H
3
4  #include <file_location.h>
5
6  #define ERROR_PREFIX "(error) "
7
8  #define WARNING_PREFIX "(warning) "
9
10 void report_error_location(File_Location *loc,
11     Const_String format, ...);
12
13 void report_warning_location(File_Location *loc, Const_String format, ...);
14
15 void report_vaerror_location(File_Location *loc,
16     Const_String format, va_list vl);
17
18 void report_vawarning_location(File_Location *loc,
19     Const_String format, va_list vl);
20
21 void report_vaerror(Const_String file_name, Const_String format, va_list vl);
22
23 void report_vawarning(Const_String file_name, Const_String format, va_list vl);
24
25 void report_error(Const_String file_name, Const_String format, ...);
26
27 void report_warning(Const_String file_name, Const_String format, ...);
28
29 /* Returns true if there has been reported an error since the
30  * report_reset() function was called last.
31  * If report_reset() has never been called was_error_reported()
32  * returns true if an error has been reported. */
33 bool was_error_reported();
34
35 bool was_warning_reported();
36
37 bool is_error_reported_here(File_Location *loc);
38
39 bool is_warning_reported_here(File_Location *loc);
40
41 int32_t get_error_count();
42
43 int32_t get_warning_count();
44
45 void clear_reports();
46
47 void show_reports();
48
49 void report_reset();
50

```

```

51 void report_print();
52
53 bool report_exhausted();
54
55 static inline void show_reports_clear()
56 {
57     show_reports();
58     clear_reports();
59 }
60
61 void force_error_reported();
62
63 #endif // ERROR_HANDLER_H

```

:

A.10.24 src/single_list.c

```

1  #include <main.h>
2  #include <single_list.h>
3
4  void single_list_prepend(Single_List *list, Single_List_Node *node)
5  {
6      if (single_list_is_empty(list)) {
7          list->head = node;
8          node->next = NULL;
9      } else {
10         node->next = list->head;
11         list->head = node;
12     }
13 }
14
15 Single_List_Node *single_list_pop_first(Single_List *list)
16 {
17     if (!single_list_is_empty(list)) {
18         Single_List_Node *h = single_list_peek_first(list);
19         list->head = h->next;
20         return h;
21     }
22     return NULL;
23 }

```

:

A.10.25 src/single_list.h

```

1  #ifndef SINGLE_LIST_H
2  #define SINGLE_LIST_H
3
4  #include <stdint.h>
5  #include <stdbool.h>
6
7  typedef struct Single_List_Node {
8      struct Single_List_Node *next;
9  } Single_List_Node;
10
11 /* You may assume that sizeof(Single_List) == sizeof(size_t) and
12  * that head == NULL when the list is empty. Might be usefull to know. */
13 typedef struct Single_List {
14     Single_List_Node *head;
15 } Single_List;
16
17 #define SINGLE_LIST_STATIC_INIT() { .head = NULL }
18
19 #define SINGLE_LIST_INIT() ((Single_List)SINGLE_LIST_STATIC_INIT())
20
21 #define SINGLE_LIST(list) Single_List list = SINGLE_LIST_STATIC_INIT()

```

```

22
23 #define SINGLE_LIST_ENTRY(node_ptr, container_type, node_member_name) \
24     CONTAINER_OF(node_ptr, container_type, node_member_name)
25
26 static inline bool single_list_is_empty(const Single_List *list)
27 {
28     return !list->head;
29 }
30
31 static inline Single_List_Node *single_list_peek_first(Single_List *list)
32 {
33     return list->head;
34 }
35
36 Single_List_Node *single_list_pop_first(Single_List *list);
37
38 typedef void (*Single_List_Destructor)(Single_List_Node *node);
39
40 static inline void single_list_for_each_destroy(Single_List *list,
41     Single_List_Destructor destructor)
42 {
43     for (Single_List_Node *n = single_list_pop_first(list); n;
44         n = single_list_pop_first(list))
45         destructor(n);
46 }
47
48 /* Remove node from the single list. Node prev is the single list node
49  * before node in the list. If prev == NULL then node is assumed to be
50  * the first element in the list. */
51 static inline void single_list_remove(Single_List *list,
52     Single_List_Node *node, Single_List_Node *prev)
53 {
54     if (prev)
55         prev->next = node->next;
56     else
57         single_list_pop_first(list);
58 }
59
60 void single_list_prepend(Single_List *list, Single_List_Node *node);
61
62 /* Don't free elements from the list while iterating this loop.
63  * At least you need to break the loop right after. */
64 #define SINGLE_LIST_FOR_EACH(list_ptr, node_ptr) \
65     for (node_ptr = (list_ptr)->head; node_ptr; node_ptr = (node_ptr)->next)
66
67 #endif // SINGLE_LIST_H

```

A.10.26 src/std_define.h

```

1 #ifndef STD_DEFINE_H
2 #define STD_DEFINE_H
3
4 #include <stdlib.h>
5 #include <stdbool.h>
6 #include <stdint.h>
7 #include <stddef.h>
8 #include <assert.h>
9 #include <stdarg.h>
10 #include <inttypes.h>
11
12 #define CONTAINER_OF(ptr, type, member) ({      \
13     const __typeof__(((type *)0)->member) *__mptr = (ptr); \
14     (type *)((char *)__mptr - offsetof(type, member)); \
15 })
16
17 #define __CSTRINGIFY(arg) # arg
18 #define CSTRINGIFY(arg) __CSTRINGIFY(arg)
19
20 #define QFY(str) " " str " "
21

```

```

22 #define UNLIKELY(x) __builtin_expect((long)!!(x), true)
23 #define LIKELY(x) __builtin_expect((long)!!(x), false)
24
25 #define ALWAYS_INLINE __attribute__((always_inline)) inline
26 #define REGPARAM(n) __attribute__((regparm(n)))
27 #define UNUSED __attribute__((unused))
28 #define CONSTRUCTOR __attribute__((constructor))
29 #define DESTRUCTOR __attribute__((destructor))
30 #define NORETURN __attribute__((noreturn))
31 #define ALIGNED(size) __attribute__((aligned(size)))
32
33 #define PTR_TO_INT(ptr) ((ptrdiff_t)(ptr))
34 #define PTR_TO_UINT(ptr) ((size_t)(ptr))
35 #define INT_TO_PTR(i) ((void *) (size_t)(i))
36
37 #define VA_SETUP(vl, last) va_list vl; va_start(vl, last)
38 #define VA_END(vl) va_end(vl)
39
40 typedef uint32_t Uns;
41 typedef int32_t Int;
42
43 #define UNSIGNED_MAX UINT32_MAX
44 #define INTEGER_MAX INT32_MAX
45
46 #define PTR_SIZE __SIZEOF_POINTER__
47 #define PTR_BITS (PTR_SIZE * 8)
48
49 #define SOURCE_SUFFIX_CSTR ".vit"
50 #define INTERFACE_SUFFIX_CSTR ".viti"
51 #define OBJ_SUFFIX_CSTR ".o"
52 #define STATIC_LIB_SUFFIX_CSTR ".a"
53 #define ASM1_SUFFIX_CSTR ".s"
54 #define ASM2_SUFFIX_CSTR ".asm"
55 #define SOURCE_SUFFIX_STR S(SOURCE_SUFFIX_CSTR)
56 #define INTERFACE_SUFFIX_STR S(INTERFACE_SUFFIX_CSTR)
57 #define OBJ_SUFFIX_STR S(OBJ_SUFFIX_CSTR)
58 #define STATIC_LIB_SUFFIX_STR S(STATIC_LIB_SUFFIX_CSTR)
59 #define ASM1_SUFFIX_STR S(ASM1_SUFFIX_CSTR)
60 #define ASM2_SUFFIX_STR S(ASM2_SUFFIX_CSTR)
61
62 #define MAIN_FUNC_CSTR "main"
63 #define MAIN_FUNC_STR S(MAIN_FUNC_CSTR)
64
65 #define STDIN_FILE_SUFFIX_CSTR ".vitaly.vit"
66 #define STDIN_FILE_SUFFIX_STR S(STDIN_FILE_SUFFIX_CSTR)
67
68 #endif // STD_DEFINE_H

```

:

A.10.27 src/std_include.h

```

1 #ifndef STD_INCLUDE_H
2 #define STD_INCLUDE_H
3
4 #include <report.h>
5 #include <io.h>
6 #include <alloc.h>
7 #include <str.h>
8 #include <file_location.h>
9
10 __attribute__((noreturn)) static inline void exit_failure()
11 {
12     exit(EXIT_FAILURE);
13 }
14
15 #endif // STD_INCLUDE_H

```

:

A.10.28 src/str.c

```

1  #include <io.h>
2  #include <vector.h>
3  #include <errno.h>
4  #include <libgen.h>
5  #include <ctype.h>
6
7  String string_from_vformat(Const_String format, va_list vl)
8  {
9      String dest = ALLOC_STRING();
10     if (vasprintf(&dest->str, string_to_cstr(format), vl) == -1)
11         fatal_error(S("Unable to allocate memory for new string\n"));
12     dest->dynamic = true;
13     return dest;
14 }
15
16 void string_append_vformat(String s, Const_String format, va_list vl)
17 {
18     char *tmp;
19     if (vasprintf(&tmp, string_to_cstr(format), vl) == -1)
20         fatal_error(S("Unable to allocate memory for new string\n"));
21     string_append(s, S(tmp));
22     free_mem(tmp);
23 }
24
25 void string_assign(String s, Const_String other)
26 {
27     if (string_to_cstr(s) != string_to_cstr(other)) {
28         string_clear(s);
29         string_append(s, other);
30     } else {
31         String oth = string_duplicate(other);
32         string_clear(s);
33         string_append(s, oth);
34         string_destroy(oth);
35     }
36 }
37
38 void string_assign_vformat(String s, Const_String format, va_list vl)
39 {
40     if (string_to_cstr(s) != string_to_cstr(format)) {
41         string_clear(s);
42         string_append_vformat(s, format, vl);
43     } else {
44         String fmt = string_duplicate(format);
45         string_clear(s);
46         string_append_vformat(s, fmt, vl);
47         string_destroy(fmt);
48     }
49 }
50
51 int32_t string_base10_to_int32(Const_String s)
52 {
53     const char *cstr = string_to_cstr(s);
54     char *endptr;
55     long val = strtol(cstr, &endptr, 10);
56     if (UNLIKELY(endptr == cstr)) {
57         errno = EINVAL;
58         return 0;
59     }
60     if (UNLIKELY(*endptr != '\0')) {
61         errno = EINVAL;
62         return 0;
63     }
64     if (UNLIKELY(val > INT32_MAX)) {
65         errno = ERANGE;
66         return INT32_MAX;
67     }
68     if (UNLIKELY(val < INT32_MIN)) {
69         errno = ERANGE;
70         return INT32_MIN;
71     }

```



```

72     return val;
73 }
74
75 String string_substring(Const_String s, Uns from, Uns to)
76 {
77     String ret = ALLOC_STRING();
78     Uns len = to - from;
79     Uns size = len + 1;
80     const char *cstr = string_to_cstr(s);
81     char *new_cstr = alloc_mem(size);
82
83     strncpy(new_cstr, cstr + from, size);
84     new_cstr[len] = '\0';
85
86     ret->str = new_cstr;
87     ret->dynamic = true;
88     return ret;
89 }
90
91 Vector *string_split(Const_String s, Const_String delimiter)
92 {
93     const char *start, *end;
94     const char *cstr = string_to_cstr(s), *delim = string_to_cstr(delimiter);
95     Uns delim_len = string_length(delimiter);
96     Uns cstr_len = string_length(s);
97     const char *const cstr_end = cstr + cstr_len;
98
99     Vector *v = vector_alloc();
100
101     start = cstr;
102     while (start + delim_len <= cstr_end) {
103         if (strncmp(start, delim, delim_len)) {
104             end = strstr(start + delim_len, delim);
105             if (!end)
106                 break;
107             vector_append(v,
108                 string_substring(s, start - cstr, end - cstr));
109             start = end + delim_len;
110         } else {
111             start += delim_len;
112         }
113     }
114     if (start != cstr_end)
115         vector_append(v, string_substring(s, start - cstr, cstr_len));
116     return v;
117 }
118
119 Uns string_count(Const_String s, Const_String delimiter)
120 {
121     Uns ret = 0;
122     const char *start, *end;
123     const char *cstr = string_to_cstr(s), *delim = string_to_cstr(delimiter);
124     Uns delim_len = string_length(delimiter);
125     Uns cstr_len = string_length(s);
126     const char *const cstr_end = cstr + cstr_len;
127
128     start = cstr;
129     while (start + delim_len <= cstr_end) {
130         if (strncmp(start, delim, delim_len)) {
131             end = strstr(start + delim_len, delim);
132             if (!end)
133                 break;
134             ret++;
135             start = end + delim_len;
136         } else {
137             start += delim_len;
138         }
139     }
140     return ret;
141 }
142
143 void string_remove_first_last(String s)
144 {
145     Uns i;

```

```

146     if (!s->dynamic)
147         string_assign(s, s);
148
149     char *cstr = s->str;
150     if (*cstr) {
151         i = 1;
152         if (cstr[1]) {
153             for (; cstr[i + 1]; i++)
154                 cstr[i - 1] = cstr[i];
155         }
156         cstr[i - 1] = '\0';
157     }
158 }
159
160 void string_remove_all(String s, char c)
161 {
162     Uns i, j;
163     if (!s->dynamic)
164         string_assign(s, s);
165
166     char *cstr = s->str;
167     for (i = 0, j = 0; cstr[j]; j++) {
168         if (cstr[j] != c)
169             cstr[i++] = cstr[j];
170     }
171     cstr[i] = cstr[j];
172 }
173
174 String string_basename(Const_String s)
175 {
176     String tmp = string_duplicate(s);
177     String ret = string_alloc(S(basename(tmp->str)));
178     string_destroy(tmp);
179     return ret;
180 }
181
182 String string_dirname(Const_String s)
183 {
184     String tmp = string_duplicate(s);
185     String ret = string_alloc(S(dirname(tmp->str)));
186     string_destroy(tmp);
187     return ret;
188 }
189
190 bool string_ends_with(Const_String s, Const_String ending)
191 {
192     Uns end_len = string_length(ending);
193     Uns s_len = string_length(s);
194     if (s_len < end_len)
195         return false;
196     return !strcmp(string_to_cstr(ending), &s->const_str[s_len - end_len]);
197 }
198
199 String string_to_module_name(Const_String s)
200 {
201     String ret = NULL;
202     Int idx = string_last_index_char(s, '.');
203     if (idx == -1)
204         goto out;
205     if (!strcmp(&s->const_str[idx], SOURCE_SUFFIX_CSTR) ||
206         !strcmp(&s->const_str[idx], INTERFACE_SUFFIX_CSTR)) {
207         ret = string_duplicate(s);
208         string_set(ret, idx, '\0');
209     }
210 out:
211     return ret;
212 }
213
214 String string_to_unique_file(Const_String suffix)
215 {
216     static const char tmp_suffix[] = "XXXXXX";
217     char *tmp = alloc_mem(sizeof(tmp_suffix));
218     memcpy(tmp, tmp_suffix, sizeof(tmp_suffix));
219

```

```

220     tmp = mktemp(tmp);
221     if (!*tmp)
222         fatal_error(S("unable to generate unique file name [%m]\n"));
223
224     String ret = string_alloc(S(tmp));
225     free_mem(tmp);
226
227     string_append(ret, suffix);
228     return ret;
229 }
230
231 String string_to_tmp_file(Const_String suffix)
232 {
233     String ret = string_alloc(S("/tmp/"));
234     String tmp = string_to_unique_file(suffix);
235     string_append(ret, tmp);
236     string_destroy(tmp);
237     return ret;
238 }
239
240 void string_replace_from(String s, char c, Const_String new_end)
241 {
242     Int idx = string_last_index_char(s, c);
243     if (idx != -1)
244         s->str[idx] = '\0';
245     string_append(s, new_end);
246 }
247
248 void string_toupper(String s)
249 {
250     if (!s->dynamic)
251         string_assign(s, s);
252     char *str = s->str;
253     for (Uns i = 0; str[i]; i++)
254         str[i] = toupper(str[i]);
255 }

```

:

A.10.29 src/str.h

```

1  #ifndef STR_H
2  #define STR_H
3
4  /* Implementation of a semi dynamic string structure.
5   *
6   * XXX Note: The DEBUG* DLOG* macros use the dynamic string implementation. So
7   * don't use the debugging macros to debug dynamic string implementation code.
8   *
9   * How to create strings:
10  * STRING(s1, "Hello"); // Allocate string statically or on the stack.
11  * String s2 = string_duplicate(S("Hello")); // Allocate string dynamically
12  * String s3 = STRING_INIT(s3, "Hello"); // Same as with s1
13  * CONST_STRING(s4, "Hello"); // Same as with s1, but this string is const.
14  *
15  * Note that it is your job to determine whether to call
16  * string_destroy() to free the memory used by strings.
17  * In general if you initialize a string with STRING() or STRING_INIT()
18  * you should NOT call string_destroy() (it will cause a seg fault).
19  *
20  * If you initialize a Const_String with CONST_STRING(), CONST_STRING_INIT()
21  * or even S() you must NOT call string_clear() to free memory used
22  * by the string. After initializing a string with STRING() or STRING_INIT()
23  * you should call string_clear() to free the memory used by the string.
24  *
25  * If you for some reason pass a dynamically allocated C string to STRING(),
26  * STRING_INIT(), CONST_STRING(), CONST_STRING_INIT() or S() you must NOT
27  * deallocate the C string before the String (or Const_String) is not needed
28  * anymore. Also note that it is your job to deallocate the C string at some
29  * point. As a general rule don't pass a dynamically allocated C string to
30  * any of the String (or Const_String) initialization macros like STRING()

```

```

31  * unless you know what you are doing, use string_duplicate() instead.
32  *
33  * If you allocate a String with string_duplicate (for example) you must
34  * call string_destroy() to deallocate the string when not needed anymore.
35  *
36  * If you want to pass a Const_String to a function you can use the S() macro.
37  * For example:
38  *   print_message(S("%D\n"), 1);
39  * Will output:
40  *   1
41  *
42  * Also note that if you would like to print a String (or Const_String)
43  * You can use the %S printf specifier. For example:
44  *   CONST_STRING(s, "Hello ");
45  *   report_error(S("file-name"), S("%S %S\n"), s, S("world"));
46  * Will output:
47  *   Hello world
48  *
49  * A bigger example:
50  *   STRING(s1, "Hello");
51  *   String s2 = string_duplicate(S("world"));
52  *   String s3 = string_from_format(S("%S %S %D"), s1, s2, 2);
53  *   string_append(s1, S(" world 1"));
54  *
55  *   print_message("%S\n%S\n", s1, s2);
56  *
57  * // Free memory used by the strings:
58  *   string_clear(s1); // This one was allocated on the stack.
59  *   string_destroy(s2); // This one was dynamically allocated.
60  *   string_destroy(s3);
61  *
62  * Output:
63  *   Hello world 1
64  *   Hello world 2 */
65
66 #include <std_define.h>
67 #include <alloc.h>
68 #include <string.h>
69
70 /* Don't mind this String_Struct. (Implementation details). */
71 typedef struct String_Struct {
72     union {
73         struct {
74             union {
75                 char *str;
76                 const char *const_str;
77             };
78             bool dynamic;
79         };
80         const char *const_string_value;
81     };
82 } *String;
83
84 typedef const struct String_Struct *Const_String;
85
86 /* Allocate memory for a new string. */
87 #define ALLOC_STRING() ALLOC_NEW(struct String_Struct)
88
89 /* Used by the implementation. You probably should not use this macro. */
90 #define ALLOC_CSTR(length) alloc_mem((length) + 1)
91
92 /* Macros to initialize strings and const strings. */
93 #define STRING_STATIC_INIT(value) \
94     (&(struct String_Struct){ { { .const_str = (value) }, .dynamic = false } })
95
96 #define CONST_STRING_STATIC_INIT(value) \
97     (&(const struct String_Struct){ { .const_string_value = (value) } })
98
99 #define STRING_INIT(value) STRING_STATIC_INIT(value)
100
101 #define CONST_STRING_INIT(value) CONST_STRING_STATIC_INIT(value)
102
103 #define STRING(sname, value) String sname = STRING_STATIC_INIT(value)
104

```

```

105 #define CONST_STRING(sname, value) \
106     Const_String sname = CONST_STRING_STATIC_INIT(value)
107
108 #define STRING_EMPTY(sname) STRING(sname, "")
109
110 #define CONST_STRING_EMPTY(sname) CONST_STRING(sname, "")
111
112 /* Convenience macro to create a Const_String. */
113 #define S(value) CONST_STRING_INIT(value)
114
115 #define STRINGIFY(arg) S(CSTRINGIFY(arg))
116
117 /* Get C string representation.
118  * Respect that the returned C string is const, and don't
119  * deallocate the C string at any point. */
120 static inline const char *string_to_cstr(Const_String s)
121 {
122     return s->const_str;
123 }
124
125 static inline bool string_is_empty(Const_String s)
126 {
127     return !*string_to_cstr(s);
128 }
129
130 /* Allocate a new string and assign it with a format which is
131  * a vprintf like format String with a corresponding va_list.
132  * See string_from_format() for usage example.
133  * Also take a look at the documentation for vprintf ($ man vprintf). */
134 String string_from_vaformat(Const_String format, va_list vl);
135
136 /* Allocate a new string and assign it with a printf format with
137  * a variable number of arguments.
138  * Usage:
139  *   String s = string_from_format(S("%S%S%c"), S("A"), "B", 'C');
140  *   ...
141  *   string_destroy(s);
142  * Output:
143  *   ABC */
144 static inline String string_from_format(Const_String format, ...)
145 {
146     VA_SETUP(vl, format);
147     String s = string_from_vaformat(format, vl);
148     VA_END(vl);
149     return s;
150 }
151
152 /* XXX - Don't use this function directly.
153  * Take a look at string_clear() instead. */
154 static inline void __string_free_cstr(String s)
155 {
156     if (s->dynamic)
157         free_mem(s->str);
158 }
159
160 /* Free memory used by the String s.
161  * Don't pass Const_String to this function.
162  * After call to this function s == "".
163  * So you can still use s after a calling this function.
164  * See string_destroy() if you want to deallocate the string completely. */
165 static inline void string_clear(String s)
166 {
167     __string_free_cstr(s);
168     s->dynamic = false;
169     s->const_str = "";
170 }
171
172 /* Deallocate a string along with it's contents.
173  * XXX - Important not to try to deallocate statically allocated strings
174  * which have been initialized with one of the string initialization
175  * macros like STRING() or STRING_INIT(). */
176 static inline void string_destroy(String s)
177 {
178     if (s) {

```

```

179     __string_free_cstr(s);
180     free_mem(s);
181 }
182 }
183
184 static inline size_t string_length(Const_String s)
185 {
186     return strlen(string_to_cstr(s));
187 }
188
189 /* Append other to s. */
190 static inline void string_append(String s, Const_String other)
191 {
192     char *tmp = ALLOC_CSTR(string_length(s) + string_length(other));
193     strcpy(tmp, string_to_cstr(s));
194     strcat(tmp, string_to_cstr(other));
195     __string_free_cstr(s);
196     s->str = tmp;
197     s->dynamic = true;
198 }
199
200 /* Like string_from_vaformat(), but append s with the formatted string. */
201 void string_append_vaformat(String s, Const_String format, va_list vl);
202
203 /* Like string_from_format(), but append s with the formatted string. */
204 static inline void string_append_format(String s, Const_String format, ...)
205 {
206     VA_SETUP(vl, format);
207     string_append_vaformat(s, format, vl);
208     VA_END(vl);
209 }
210
211 /* Like string_from_format, but append the format to String s. */
212 void string_append_format(String s, Const_String format, ...);
213
214 /* Duplicate String (other) and return a new String == other.
215  * You can freely deallocate other whenever you want after calling this
216  * function.
217  * Remember to free the returned String with string_destroy() when
218  * it's not needed anymore. */
219 static inline String string_duplicate(Const_String other)
220 {
221     String s = ALLOC_STRING();
222     s->str = ALLOC_CSTR(string_length(other));
223     strcpy(s->str, string_to_cstr(other));
224     s->dynamic = true;
225     return s;
226 }
227
228 /* Same as string_duplicate(). See string_duplicate() and string_destroy(). */
229 static inline String string_alloc(Const_String contents)
230 {
231     return string_duplicate(contents);
232 }
233
234 /* String compare s1 with s2.
235  * Returns -1 if s1 < s2,
236  * returns 0 if s1 == s2,
237  * returns 1 if s1 > s2. */
238 static inline Int string_compare(Const_String s1, Const_String s2)
239 {
240     return strcmp(string_to_cstr(s1), string_to_cstr(s2));
241 }
242
243 /* Case insensitive version of string_compare() */
244 static inline Int string_compare_nocase(Const_String s1, Const_String s2)
245 {
246     return strcasecmp(string_to_cstr(s1), string_to_cstr(s2));
247 }
248
249 /* Clear the String s and afterwards let s == other.
250  * You can freely deallocate other whenever you want after calling
251  * this function. */
252 void string_assign(String s, Const_String other);

```

```

253
254 /* Like string_from_vaformat(), but assign s with the formatted string. */
255 void string_assign_vaformat(String s, Const_String format, va_list vl);
256
257 /* Like string_from_format(), but assign s with the formatted string. */
258 static inline void string_assign_format(String s, Const_String format, ...)
259 {
260     VA_SETUP(vl, format);
261     string_assign_vaformat(s, format, vl);
262     VA_END(vl);
263 }
264
265 /* Convert s into a base 10 integer.
266  * You might have to check errno after calling this function.
267  * If s is not an integer the function returns 0 with errno == EINVAL.
268  * If the integer in s is too big i.e. the int32_t overflows the
269  * function returns INT32_MAX with errno == ERANGE.
270  * If the integer in s is too small i.e. the int32_t underflows the
271  * function returns INT32_MIN with errno == ERANGE.
272  * Otherwise the function returns s converted to an Int. */
273 int32_t string_base10_to_int32(Const_String s);
274
275 /* Get the hash code of a string. (Idea stolen from java String class). */
276 static inline Uns string_hash_code(Const_String s)
277 {
278     Uns hash = 0;
279     for (const char *val = string_to_cstr(s); *val; val++)
280         hash = (hash << 5) - hash + *val;
281     return hash;
282 }
283
284 /* Warning, no bounds checking.
285  * Slow operation if the string is not already dynamic.
286  * (When s->dynamic == false). */
287 static inline void string_set(String s, Uns idx, char val)
288 {
289     if (!s->dynamic)
290         string_assign(s, s);
291     s->str[idx] = val;
292 }
293
294 /* Set last char of s to val. */
295 static inline void string_set_last_char(String s, char val)
296 {
297     string_set(s, string_length(s) - 1, val);
298 }
299
300 /* Warning, no bounds checking. */
301 static inline char string_get(Const_String s, Uns idx)
302 {
303     return string_to_cstr(s)[idx];
304 }
305
306 /* Get last char of s. */
307 static inline char string_get_last(Const_String s)
308 {
309     Uns len = string_length(s);
310     if (len)
311         return string_get(s, len - 1);
312     return '\0';
313 }
314
315 /* Returns -1 if char c is not in the string.
316  * Else returns index of first occurrence of c in s. */
317 static inline Int string_first_index_char(Const_String s, char c)
318 {
319     const char *cstr = string_to_cstr(s);
320     const char *tmp = strchr(cstr, c);
321     if (!tmp)
322         return -1;
323     return tmp - cstr;
324 }
325
326 /* Returns -1 if char c is not in the string.

```

```

327  * Else returns index of last occurrence of c in s. */
328  static inline Int string_last_index_char(Const_String s, char c)
329  {
330      const char *cstr = string_to_cstr(s);
331      const char *tmp = strrchr(cstr, c);
332      if (!tmp)
333          return -1;
334      return tmp - cstr;
335  }
336
337  /* Replace first occurrence of search_char with replace_char. */
338  static inline void string_search_replace_char(String s, char search_char,
339      char replace_char)
340  {
341      const char *cstr = string_to_cstr(s);
342      const char *tmp = strchr(cstr, search_char);
343      if (tmp)
344          string_set(s, tmp - cstr, replace_char);
345  }
346
347  /* Get substring of s with first char at index 'from' and last char
348   * at index 'to' - 1.
349   * Warning, no bounds checking:
350   * So make sure 'from' <= length of string and to <= length of string. */
351  String string_substring(Const_String s, Uns from, Uns to);
352
353  typedef struct Vector Vector;
354
355  /* Remember to destroy the vector with
356   * vector_destroy(vector, (Vector_Destructor)string_destroy). */
357  Vector *string_split(Const_String s, Const_String delimiter);
358
359  /* Count the number of occurrences of delimiter */
360  Uns string_count(Const_String s, Const_String delimiter);
361
362  /* Remove first and last character from s. */
363  void string_remove_first_last(String s);
364
365  /* Remove all occurrences of c from s. */
366  void string_remove_all(String s, char c);
367
368  /* Shift String s left with 'amount' characters. */
369  static inline void string_shift_left(String s, Uns amount)
370  {
371      Uns len = string_length(s);
372      if (len < amount)
373          string_clear(s);
374      else
375          for (Uns i = 0; i < len - amount + 1; i++)
376              string_set(s, i, string_get(s, i + amount));
377  }
378
379  static inline void string_replace_all(String s, char old_c, char new_c)
380  {
381      if (!s->dynamic)
382          string_assign(s, s);
383      char *c = s->str;
384      while (*c != '\0') {
385          if (*c == old_c)
386              *c = new_c;
387          ++c;
388      }
389  }
390
391  static inline String string_cpy_replace_all(Const_String s, char old_c,
392      char new_c)
393  {
394      String ret = string_duplicate(s);
395      string_replace_all(ret, old_c, new_c);
396      return ret;
397  }
398
399  bool string_ends_with(Const_String s, Const_String ending);
400

```



```

401 String string_basename(Const_String s);
402
403 String string_dirname(Const_String s);
404
405 String string_to_module_name(Const_String s);
406
407 static inline String string_dir_concat(Const_String dir, Const_String lhs)
408 {
409     return string_from_format(S("%S/%S"), dir, lhs);
410 }
411
412 static inline const char *__string_after_last(Const_String s, char c)
413 {
414     const char *result;
415     Int idx = string_last_index_char(s, c);
416     if (idx == -1)
417         result = string_to_cstr(s);
418     else
419         result = &string_to_cstr(s)[idx + 1];
420     return result;
421 }
422
423 static inline const char *__string_after_first(Const_String s, char c)
424 {
425     const char *result;
426     Int idx = string_first_index_char(s, c);
427     if (idx == -1)
428         result = string_to_cstr(s);
429     else
430         result = &string_to_cstr(s)[idx + 1];
431     return result;
432 }
433
434 #define STRING_AFTER_LAST(str, ch) S(__string_after_last(str, ch))
435
436 #define STRING_AFTER_DOT(str) S(__string_after_last(str, '.'))
437
438 #define STRING_AFTER_FIRST(str, ch) S(__string_after_first(str, ch))
439
440 static inline String string_between_alloc(Const_String s, char c)
441 {
442     String ret = string_duplicate(STRING_AFTER_FIRST(s, c));
443     Int last_idx = string_last_index_char(ret, c);
444     if (last_idx != -1)
445         ret->str[last_idx] = '\0';
446     return ret;
447 }
448
449 /* Get a unique file name name ending with 'suffix'. */
450 String string_to_unique_file(Const_String suffix);
451
452 /* Get a unique file name name starting with "/tmp/" and
453  * ending with 'suffix'. */
454 String string_to_tmp_file(Const_String suffix);
455
456 /* Replace characters from last c with new_end.
457  * If c is not in the string new_end is just appended to the end of s.
458  * Example:
459  *   if s == "file.vit" then
460  *   string_replace_after(s, '.', S(".o")) results in
461  *   s == "file.o" */
462 void string_replace_from(String s, char c, Const_String new_end);
463
464 static inline String string_cpy_replace_from(Const_String s, char c,
465     Const_String new_end)
466 {
467     String ret = string_duplicate(s);
468     string_replace_from(ret, c, new_end);
469     return ret;
470 }
471
472 void string_toupper(String s);
473
474 #endif // STR_H

```

:

A.10.30 src/string_builder.c

```

1  #include <string_builder.h>
2
3  void string_builder_append(String_Builder *b, Const_String s)
4  {
5      Uns slen, new_len;
6
7      if (!b->str_buf) {
8          __string_builder_assign(b, s);
9          return;
10     }
11
12     slen = string_length(s);
13     new_len = b->str_len + slen;
14
15     if (b->buf_size <= new_len) {
16         char *nbuf;
17         b->buf_size = (new_len << STRING_BUILDER_SHIFT) + 1;
18         nbuf = alloc_mem(b->buf_size);
19         memcpy(nbuf, b->str_buf, b->str_len + 1);
20         free_mem(b->str_buf);
21         b->str_buf = nbuf;
22     }
23
24     memcpy(b->str_buf + b->str_len, string_to_cstr(s), slen + 1);
25     b->str_len = new_len;
26 }

```

:

A.10.31 src/string_builder.h

```

1  #ifndef STRING_BUILDER_H
2  #define STRING_BUILDER_H
3
4  #include <str.h>
5
6  typedef struct String_Builder {
7      union {
8          struct {
9              char *str_buf;
10             Uns buf_size;
11             Uns str_len;
12         };
13         struct String_Struct string_struct;
14     };
15 } String_Builder;
16
17 #define STRING_BUILDER_STATIC_INIT() {{{ \
18     .str_buf = NULL,           \
19     .buf_size = 0,             \
20     .str_len = 0               \
21 }}}
22
23 #define STRING_BUILDER_INIT() ((String_Builder)STRING_BUILDER_STATIC_INIT())
24
25 #define STRING_BUILDER(name) String_Builder name = STRING_BUILDER_STATIC_INIT()
26
27 #define STRING_BUILDER_SHIFT 2
28
29 static inline void string_builder_clear(String_Builder *b)
30 {
31     free_mem(b->str_buf);
32     b->buf_size = 0;
33     b->str_len = 0;

```

```

34     b->str_buf = NULL;
35 }
36
37 static inline void __string_builder_assign(String_Builder *b, Const_String s)
38 {
39     b->str_len = string_length(s);
40     b->buf_size = (b->str_len << STRING_BUILDER_SHIFT) + 1;
41     b->str_buf = alloc_mem(b->buf_size);
42     memcpy(b->str_buf, string_to_cstr(s), b->str_len + 1);
43 }
44
45 static inline void string_builder_assign(String_Builder *b, Const_String s)
46 {
47     free_mem(b->str_buf);
48     __string_builder_assign(b, s);
49 }
50
51 void string_builder_append(String_Builder *b, Const_String s);
52
53 static inline void string_builder_append_char(String_Builder *b, Int c)
54 {
55     c &= (Int)0xff;
56     string_builder_append(b, S((const char *)&c));
57 }
58
59 static inline void string_builder_append_int16(String_Builder *b, Int chars)
60 {
61     chars &= (Int)0xffff;
62     string_builder_append(b, S((const char *)&chars));
63 }
64
65 static inline void string_builder_set_last_char(String_Builder *b, char c)
66 {
67     b->str_buf[b->str_len - 1] = c;
68 }
69
70 /* Warning. Might return NULL if the string builder
71  * has not been assigned a value. */
72 static inline const char *string_builder_to_cstr(String_Builder *b)
73 {
74     return b->str_buf;
75 }
76
77 /* Warning. Might return NULL if the string builder
78  * has not been assigned a value. */
79 static inline Const_String string_builder_const_str(String_Builder *b)
80 {
81     if (b->str_buf)
82         return &b->string_struct;
83     else
84         return NULL;
85 }
86
87 #endif // STRING_BUILDER_H

```

:

A.10.32 src/test/test_hash_map.c

```

1  #include <parser.h>
2  #include <std_define.h>
3  #include <std_include.h>
4  #include <stdio.h>
5
6  #include <time.h>
7  #include <hash_map.h>
8  #include <stdlib.h>
9
10 typedef struct Integer_Slot {
11     Hash_Map_Slot slot;
12     Uns val;

```

628

```

87
88 #define NUM_INSERTIONS2 (100000 * 10)
89 Uns *insertions2 = alloc_mem(sizeof(Uns) * NUM_INSERTIONS2);
90 for (int i = 0; i < NUM_INSERTIONS2; i++) {
91     Integer_Slot *s = ALLOC_NEW(Integer_Slot);
92     s->val = rand() % (NUM_INSERTIONS2 >> 1);
93     insertions2[i] = s->val;
94     hash_map_insert(&map, &s->slot, s->val);
95     if (map.num_slots > hash_map_size(&map) * (1 / HASH_MAP_MAX_LOAD_FACTOR) * 3.05 &&
96         map.num_slots > HASH_MAP_PRIME_LIST[HASH_MAP_DEFAULT_SIZE]) {
97         print_message(S("1. Unexpected number of slots size %U\n"), map.num_slots);
98         exit_failure();
99     }
100 }
101
102 if (hash_map_size(&map) != NUM_INSERTIONS2 + INSERT_SIZE - NUM_REMOVED) {
103     print_message(S("3. Unexpected map size %U\n"), hash_map_size(&map));
104     exit_failure();
105 }
106
107 for (Uns i = 0; i < NUM_INSERTIONS2; i++) {
108     if (!hash_map_contains(&map, INT_TO_PTR(insertions2[i]),
109         insertions2[i])) {
110         print_message(S("2. Could not find expected value: %U\n"),
111             insertions2[i]);
112         exit_failure();
113     }
114 }
115
116 for (int i = 0; i < NUM_INSERTIONS2 - NUM_INSERTIONS2 / 10; i++) {
117     free_mem(hash_map_remove(&map, INT_TO_PTR(insertions2[i]),
118         insertions2[i]));
119     if (map.num_slots > hash_map_size(&map) * 6) {
120         print_message(S("2. Unexpected number of slots size %U\n"), map.num_slots);
121         exit_failure();
122     }
123 }
124
125 if (hash_map_size(&map) !=
126     NUM_INSERTIONS2 / 10 + INSERT_SIZE - NUM_REMOVED) {
127     print_message(S("4. Unexpected map size %U\n"), hash_map_size(&map));
128     exit_failure();
129 }
130
131 for (Uns i = NUM_INSERTIONS2 - NUM_INSERTIONS2 / 10; i < NUM_INSERTIONS2; i++) {
132     if (!hash_map_contains(&map, INT_TO_PTR(insertions2[i]),
133         insertions2[i])) {
134         print_message(S("3. Could not find expected value: %U\n"),
135             insertions2[i]);
136         exit_failure();
137     }
138 }
139
140 for (int i = NUM_INSERTIONS2 - NUM_INSERTIONS2 / 10; i < NUM_INSERTIONS2; i++)
141     free_mem(hash_map_remove(&map, INT_TO_PTR(insertions2[i]),
142         insertions2[i]));
143
144 for (Uns i = 0; i < INSERT_SIZE; i++)
145     free_mem(hash_map_remove(&map, INT_TO_PTR(insert_values[i]),
146         insert_values[i]));
147
148 if (hash_map_size(&map) != 0) {
149     print_message(S("5. Unexpected map size %U\n"), hash_map_size(&map));
150     exit_failure();
151 }
152
153 free_mem(insertions2);
154
155 #define NUM_INSERTIONS3 10000
156 for (int i = 0; i < NUM_INSERTIONS3; i++) {
157     Integer_Slot *s = ALLOC_NEW(Integer_Slot);
158     s->val = rand() % 5;
159     hash_map_insert(&map, &s->slot, s->val);
160 }

```

```

161
162     if (hash_map_size(&map) != NUM_INSERTIONS3) {
163         print_message(S("6. Unexpected map size %U\n"), hash_map_size(&map));
164         exit_failure();
165     }
166
167     hash_map_for_each_destroy(&map, hash_map_destructor);
168
169     if (hash_map_size(&map) != 0) {
170         print_message(S("7. Unexpected map size %U\n"), hash_map_size(&map));
171         exit_failure();
172     }
173
174     print_message(S("Hash map test success.\n"));
175 }

```

:

A.10.33 src/test/test_include.h

```

1  #ifndef TEST_INCLUDE_H
2  #define TEST_INCLUDE_H
3
4  void test_hash_map();
5  void test_rb_tree();
6  void test_lists();
7  void test_import_handler();
8
9  #endif // TEST_INCLUDE_H

```

:

A.10.34 src/test/test_lists.c

```

1  #include <std_define.h>
2  #include <std_include.h>
3  #include <timer.h>
4  #include <rb_tree.h>
5  #include <vector.h>
6  #include <double_list.h>
7  #include <single_list.h>
8
9  struct number {
10     Single_List_Node n;
11     int i;
12 };
13
14 struct db_number {
15     Double_List_Node n;
16     int i;
17 };
18
19 struct rb_number {
20     Rb_Tree_Node n;
21     int i;
22 };
23
24 void single_destructor(Single_List_Node *node)
25 {
26     free_mem(CONTAINER_OF(node, struct number, n));
27 }
28
29 void double_destructor(Double_List_Node *node)
30 {
31     free_mem(CONTAINER_OF(node, struct db_number, n));
32 }
33

```

```

34 void tree_destructor(Rb_Tree_Node *node)
35 {
36     free_mem(CONTAINER_OF(node, struct rb_number, n));
37 }
38
39 bool rb_compare(Rb_Tree_Node *ins_node, Rb_Tree_Node *tree_node)
40 {
41     return RB_TREE_ENTRY(ins_node, struct rb_number, n)->i <=
42         RB_TREE_ENTRY(tree_node, struct rb_number, n)->i;
43 }
44
45 #define COUNT 1000000
46
47 void test_lists()
48 {
49     SINGLE_LIST(list);
50     DOUBLE_LIST(d);
51     VECTOR(v);
52     RB_TREE(tree, rb_compare);
53
54     Timer t = TIMER_INIT;
55
56     for (int i = 0; i < COUNT; i++) {
57         struct db_number *num = ALLOC_NEW(struct db_number);
58         num->i = i;
59         double_list_prepend(&d, &num->n);
60     }
61     double_list_for_each_destroy(&d, double_destructor);
62
63     timer_restart(&t);
64     for (int i = 0; i < COUNT; i++) {
65         struct number *num = ALLOC_NEW(struct number);
66         num->i = i;
67         single_list_prepend(&list, &num->n);
68     }
69     timer_stop(&t);
70     print_message(S("Insert %D into single list time: %.2fms\n"),
71         COUNT, timer_get_time(&t));
72     single_list_for_each_destroy(&list, single_destructor);
73
74     timer_restart(&t);
75     for (int i = 0; i < COUNT; i++) {
76         struct db_number *num = ALLOC_NEW(struct db_number);
77         num->i = i;
78         double_list_prepend(&d, &num->n);
79     }
80     timer_stop(&t);
81     print_message(S("Insert %D into double list time: %.2fms\n"),
82         COUNT, timer_get_time(&t));
83     double_list_for_each_destroy(&d, double_destructor);
84
85     timer_restart(&t);
86     for (int i = 0; i < COUNT; i++)
87         vector_append(&v, INT_TO_PTR(i));
88     timer_stop(&t);
89     print_message(S("Insert %D into vector time: %.2fms\n"),
90         COUNT, timer_get_time(&t));
91     vector_clear(&v);
92
93     timer_restart(&t);
94     for (int i = 0; i < COUNT; i++) {
95         struct rb_number *num = ALLOC_NEW(struct rb_number);
96         num->i = i;
97         rb_tree_insert(&tree, &num->n);
98     }
99     timer_stop(&t);
100    print_message(S("Insert %D into rb tree time: %.2fms\n"),
101        COUNT, timer_get_time(&t));
102    rb_tree_for_each_destroy(&tree, tree_destructor);
103 }

```

:

A.10.35 src/test/test_rb_tree.c

```

1  #include <parser.h>
2  #include <std_define.h>
3  #include <std_include.h>
4  #include <stdio.h>
5  #include <double_list.h>
6
7  #include <time.h>
8  #include <stdlib.h>
9  #include <rb_tree.h>
10
11 static Uns black_count;
12
13 struct RB_Data {
14     Rb_Tree_Node n;
15     int i;
16 };
17
18 Rb_Tree_Color test_tree(Rb_Tree *tree, Rb_Tree_Node *node, Uns bc)
19 {
20     if (node->color == RB_TREE_BLACK)
21         ++bc;
22
23     if (node == RB_TREE_NULL(tree)) {
24         if (!black_count)
25             black_count = bc;
26         else if (bc != black_count)
27             print_message(S("WRONG BLACK COUNT! Black count = %U\n"), bc);
28         return node->color;
29     }
30
31     if (test_tree(tree, node->left, bc) == RB_TREE_RED &&
32         node->color == RB_TREE_RED)
33         print_message(S("TWO RED NODES IN A ROW!\n"));
34     if (test_tree(tree, node->right, bc) == RB_TREE_RED &&
35         node->color == RB_TREE_RED)
36         print_message(S("TWO RED NODES IN A ROW!\n"));
37
38     return node->color;
39 }
40
41 void test_tree_balance(Rb_Tree *tree)
42 {
43     black_count = 0;
44     if (tree->root->color == RB_TREE_RED)
45         print_message(S("RED ROOT MAN!\n"));
46     test_tree(tree, tree->root, 0);
47 }
48
49 static Uns print_tree_indent;
50
51 void print_node_inorder(Rb_Tree *tree, Rb_Tree_Node *n)
52 {
53     struct RB_Data *data = RB_TREE_ENTRY(n, struct RB_Data, n);
54     for (Uns i = 0; i < print_tree_indent; i++)
55         print_message(S(" "));
56     print_message(S("<node color=%S value=%D>\n"),
57         n->color == RB_TREE_BLACK ? S("black") : S("red"),
58         data->i);
59     print_tree_indent += 4;
60     for (Uns i = 0; i < print_tree_indent - 4; i++)
61         print_message(S(" "));
62     print_message(S("LEFT:\n"));
63     if (n->left != RB_TREE_NULL(tree))
64         print_node_inorder(tree, n->left);
65     for (Uns i = 0; i < print_tree_indent - 4; i++)
66         print_message(S(" "));
67     print_message(S("RIGHT:\n"));
68     if (n->right != RB_TREE_NULL(tree))
69         print_node_inorder(tree, n->right);
70     print_tree_indent -= 4;
71 #if 0

```



```

72     for (Uns i = 0; i < print_tree_indent; i++)
73         print_message(S(" "));
74     print_message(S("</node color=\"%S\" value=\"%ld\">\n"),
75         n->color == RB_TREE_BLACK ? S("black") : S("red"),
76         PTR_TO_INT(n->data));
77 #endif
78 }
79
80 void print_tree(Rb_Tree *tree)
81 {
82     print_tree_indent = 0;
83     if (tree->root != RB_TREE_NULL(tree))
84         print_node_inorder(tree, tree->root);
85 }
86
87 static int prev_value;
88
89 void callback(Rb_Tree_Node *node)
90 {
91     struct RB_Data *data = RB_TREE_ENTRY(node, struct RB_Data, n);
92     if (prev_value > data->i)
93         print_message(S("TREE CORRUPTED!\n"));
94     prev_value = data->i;
95 }
96
97 bool compare(Rb_Tree_Node *search_node, Rb_Tree_Node *tree_node)
98 {
99     struct RB_Data *lhs = RB_TREE_ENTRY(search_node, struct RB_Data, n);
100    struct RB_Data *rhs = RB_TREE_ENTRY(tree_node, struct RB_Data, n);
101    return lhs->i <= rhs->i;
102 }
103
104 Int search_compare(void *arg, Rb_Tree_Node *tree_node)
105 {
106     struct RB_Data *d = RB_TREE_ENTRY(tree_node, struct RB_Data, n);
107     if (PTR_TO_INT(arg) < d->i)
108         return -1;
109     if (PTR_TO_INT(arg) > d->i)
110         return 1;
111     return 0;
112 }
113
114 void destructor(Rb_Tree_Node *node)
115 {
116     free_mem(RB_TREE_ENTRY(node, struct RB_Data, n));
117 }
118
119 void test_rb_tree()
120 {
121     srand(time(NULL));
122
123     RB_TREE(t, compare);
124
125 #define OUTER_ITERS 100
126 #define LOOP_ITERS 10000
127 #define VALUES_SIZE (LOOP_ITERS - 1)
128
129     int *values = alloc_mem(sizeof(int) * VALUES_SIZE);
130
131     for (Uns u = 0; u < OUTER_ITERS; u++) {
132         for (Uns i = 0; i < LOOP_ITERS; i++) {
133             struct RB_Data *d = ALLOC_NEW(struct RB_Data);
134             d->i = rand();
135             rb_tree_insert(&t, &d->n);
136             if (i < VALUES_SIZE)
137                 values[i] = d->i;
138         }
139
140         prev_value = 0;
141         rb_tree_for_each(&t, callback);
142         test_tree_balance(&t);
143
144         for (Uns i = 0; i < VALUES_SIZE; i++) {
145             Rb_Tree_Node *removed_node = rb_tree_search_remove(&t,

```

```

146         search_compare, INT_TO_PTR(values[i]));
147         free_mem(removed_node);
148     }
149
150     prev_value = 0;
151     rb_tree_for_each(&t, callback);
152     test_tree_balance(&t);
153 }
154
155 free_mem(values);
156 // print_tree(&t);
157 print_message(S("Tree size: %U\n"), rb_tree_size(&t));
158
159 rb_tree_for_each_destroy(&t, destructor);
160 }

```

:

A.10.36 src/timer.c

```

1  #include <timer.h>
2  #include <sys/time.h>
3  #include <stddef.h>
4
5  void timer_start(Timer *t)
6  {
7      struct timeval tv;
8      if (!t->start_time) {
9          /* Should never fail here. */
10         gettimeofday(&tv, NULL);
11         t->start_time = tv.tv_sec * 1000.0 + tv.tv_usec / 1000.0;
12     }
13     t->started = 1;
14 }
15
16 void timer_restart(Timer *t)
17 {
18     t->start_time = 0.0;
19     timer_start(t);
20 }
21
22 void timer_stop(Timer *t)
23 {
24     struct timeval tv;
25     if (t->started) {
26         /* Should never fail here. */
27         gettimeofday(&tv, NULL);
28         t->stop_time = tv.tv_sec * 1000.0 + tv.tv_usec / 1000.0;
29     }
30 }
31
32 double timer_get_time(Timer *t)
33 {
34     return t->stop_time - t->start_time;
35 }

```

:

A.10.37 src/timer.h

```

1  #ifndef TIMER_H
2  #define TIMER_H
3
4  typedef struct Timer {
5      double start_time;
6      double stop_time;
7      int started;

```

```

8  } Timer;
9
10 #define TIMER_STATIC_INIT {0.0, 0.0, 0}
11 #define TIMER_INIT ((Timer)TIMER_STATIC_INIT)
12
13 void timer_start(Timer *t);
14
15 void timer_restart(Timer *t);
16
17 void timer_stop(Timer *t);
18
19 double timer_get_time(Timer *t);
20
21 #endif // TIMER_H

```

:

A.10.38 src/vector.c

```

1  #include <vector.h>
2
3  void *vector_pop_last(Vector *v)
4  {
5      Uns next_idx;
6      void *data;
7      next_idx = v->next_idx;
8
9      assert(next_idx > 0);
10
11      data = v->array[--next_idx];
12      if (next_idx <= v->curr_size >> 2 && next_idx >= v->min_size) {
13          v->next_size = v->curr_size;
14          v->curr_size >>= 1;
15          v->array = realloc_mem(v->array, v->curr_size * sizeof(void *));
16      }
17      v->next_idx = next_idx;
18      return data;
19  }
20
21 void vector_sort(Vector *v, Vector_Comparator comparator)
22 {
23     int vector_comparator_bounce(const void *lhs, const void *rhs)
24     {
25         return comparator(*(const void **)lhs, *(const void **)rhs);
26     }
27     qsort(v->array, vector_size(v), sizeof(void *), vector_comparator_bounce);
28 }

```

:

A.10.39 src/vector.h

```

1  #ifndef VECTOR_H
2  #define VECTOR_H
3
4  #include <std_define.h>
5  #include <alloc.h>
6
7  /* XXX Don't use debug macros to debug vector implementation since the
8   * debug implementation uses Vector. */
9
10 typedef struct Vector {
11     void **array;
12     Uns min_size;
13     Uns next_size;
14     Uns curr_size;
15     Uns next_idx;

```

```

16 } Vector;
17
18 #define VECTOR_DEFAULT_SIZE 8
19
20 /* Note that init_size must be > 0 */
21 #define VECTOR_STATIC_INIT_SIZE(init_size) { \
22     .array = NULL, \
23     .min_size = init_size, \
24     .next_size = init_size, \
25     .curr_size = 0, \
26     .next_idx = 0 \
27 }
28
29 #define VECTOR_INIT_SIZE(init_size) \
30     ((Vector)VECTOR_STATIC_INIT_SIZE(init_size))
31
32 #define VECTOR_STATIC_INIT() VECTOR_STATIC_INIT_SIZE(VECTOR_DEFAULT_SIZE)
33
34 #define VECTOR_INIT() ((Vector)VECTOR_STATIC_INIT())
35
36 #define VECTOR(vec) Vector vec = VECTOR_STATIC_INIT()
37
38 /* Note that init_size must be > 0 */
39 #define VECTOR_SIZE(vec, size) Vector vec = VECTOR_STATIC_INIT_SIZE(size)
40
41 /* Allocate and initialize vector with initial size: 'size'. */
42 static inline Vector *vector_alloc_size(Uns size)
43 {
44     Vector *ret = ALLOC_NEW(Vector);
45     *ret = VECTOR_INIT_SIZE(size);
46     return ret;
47 }
48
49 /* Allocate and initialize vector with initial default size. */
50 static inline Vector *vector_alloc()
51 {
52     return vector_alloc_size(VECTOR_DEFAULT_SIZE);
53 }
54
55 /* Warning. The function does not check whether the Uns integer containing
56  * the size of the vector overflows. So make sure the size of the vector
57  * is <= UNSIGNED_MAX before calling this function. */
58 static inline void vector_append(Vector *v, void *data)
59 {
60     assert(v->next_idx <= UNSIGNED_MAX);
61     if (v->next_idx >= v->curr_size) {
62         v->array = realloc_mem(v->array, v->next_size * sizeof(void *));
63         v->curr_size = v->next_size;
64         v->next_size <= 1;
65     }
66     v->array[v->next_idx++] = data;
67 }
68
69 /* Warning, no bounds checking. Make sure size of vector > 0. */
70 void *vector_pop_last(Vector *v);
71
72 static inline Uns vector_size(Vector *v)
73 {
74     return v->next_idx;
75 }
76
77 static inline void *vector_peek_last(Vector *v)
78 {
79     assert(vector_size(v) > 0);
80     return v->array[vector_size(v) - 1];
81 }
82
83 typedef void (*Vector_Destructor)(void *data);
84
85 /* Should return -1 when search_obj < vec_obj,
86  * and return 0 when search_obj == vec_obj,
87  * and return 1 when search_obj > vec_obj. */
88 typedef int (*Vector_Comparator)(const void *search_obj, const void *vec_obj);
89

```

```

90 static inline void vector_set(Vector *v, Uns idx, void *data)
91 {
92     assert(idx <= v->next_idx);
93     if (idx < v->next_idx)
94         v->array[idx] = data;
95     else
96         vector_append(v, data);
97 }
98
99 /* Warning, no bounds checking. Make sure size of vector > 0. */
100 static inline void vector_set_last(Vector *v, void *data)
101 {
102     assert(vector_size(v) > 0);
103     v->array[vector_size(v) - 1] = data;
104 }
105
106 /* Warning, no bounds checking. Make sure idx < size of vector. */
107 static inline void *vector_get(Vector *v, Uns idx)
108 {
109     assert(idx < v->next_idx);
110     return v->array[idx];
111 }
112
113 /* Swap two vector elements.
114  * Warning, no bounds checking. Make sure idx1 < size of vector
115  * and idx2 < size of vector. */
116 static inline void vector_swap(Vector *v, Uns idx1, Uns idx2)
117 {
118     void *tmp = vector_get(v, idx1);
119     vector_set(v, idx1, vector_get(v, idx2));
120     vector_set(v, idx2, tmp);
121 }
122
123 /* idx must be <= vector size. */
124 static inline void vector_insert(Vector *v, Uns idx, void *data)
125 {
126     Uns size = vector_size(v);
127     assert(idx <= size);
128     if (idx < size) {
129         vector_append(v, vector_get(v, size - 1));
130         for (Uns i = idx; i < size - 1; i++)
131             vector_set(v, i + 1, vector_get(v, i));
132         vector_set(v, idx, data);
133     } else {
134         vector_append(v, data);
135     }
136 }
137
138 /* idx must be < vector size. */
139 static inline void *vector_remove(Vector *v, Uns idx)
140 {
141     Uns size = vector_size(v);
142     assert(idx < size);
143     void *ret = vector_get(v, idx);
144     for (Uns i = idx; i < size - 1; i++)
145         vector_set(v, i, vector_get(v, i + 1));
146     vector_pop_last(v);
147     return ret;
148 }
149
150 /* remove ptr from v.
151  * Returns true if ptr was in the vector.
152  * Otherwise returns false. */
153 static inline bool vector_remove_ptr(Vector *v, void *ptr)
154 {
155     Uns size = vector_size(v);
156     for (Uns i = 0; i < size; i++) {
157         if (vector_get(v, i) == ptr) {
158             vector_remove(v, i);
159             return true;
160         }
161     }
162     return false;
163 }

```

```

164
165 /* Warning, no bounds checking. Make sure idx < size of vector. */
166 static inline void *vector_peek_first(Vector *v)
167 {
168     return vector_get(v, 0);
169 }
170
171 static inline bool vector_is_empty(const Vector *v)
172 {
173     return !v->next_idx;
174 }
175
176 #define VECTOR_FOR_EACH_ENTRY(vec_ptr, data_ptr) \
177 for (Uns __i = 0; __i < (vec_ptr)->next_idx && \
178      ((data_ptr = (__typeof__(data_ptr))(vec_ptr)->array[__i]) || \
179       !data_ptr); __i++)
180
181 #define VECTOR_FOR_EACH_ENTRY_REVERSED(vec_ptr, data_ptr) \
182 if ((vec_ptr)->next_idx) \
183 for (Uns __i = (vec_ptr)->next_idx; __i > 0 && \
184      ((data_ptr = (__typeof__(data_ptr))(vec_ptr)->array[--__i]) || \
185       !data_ptr); )
186
187 static inline void vector_clear(Vector *v)
188 {
189     v->next_size = v->min_size;
190     v->curr_size = 0;
191     v->next_idx = 0;
192     free_mem(v->array);
193     v->array = NULL;
194 }
195
196 static inline void vector_for_each_destroy(Vector *v,
197     Vector_Destructor destructor)
198 {
199     void *data;
200     VECTOR_FOR_EACH_ENTRY(v, data)
201         destructor(data);
202     vector_clear(v);
203 }
204
205 static inline void vector_destroy(Vector *v, Vector_Destructor destructor)
206 {
207     if (!v)
208         return;
209
210     if (destructor)
211         vector_for_each_destroy(v, destructor);
212     vector_clear(v);
213     free_mem(v);
214 }
215
216 static inline bool vector_contains_ptr(Vector *v, const void *p)
217 {
218     void *oth;
219     VECTOR_FOR_EACH_ENTRY(v, oth) {
220         if (oth == p)
221             return true;
222     }
223     return false;
224 }
225
226 static inline bool vector_contains(Vector *v, Vector_Comparator comparator,
227     const void *comparator_arg)
228 {
229     void *data;
230     VECTOR_FOR_EACH_ENTRY(v, data) {
231         if (!comparator(comparator_arg, data))
232             return true;
233     }
234     return false;
235 }
236
237 void vector_sort(Vector *v, Vector_Comparator comparator);

```

```

238
239 #endif // VECTOR_H

```

A.11 Makefile System

```

:
```

A.11.1 src/aia/Makefile

```

1
2 ROOT_DIR := $(shell for i in `seq 0 9`; \
3   do if [ -n "`ls | grep Makefile.inc`" ]; \
4     then pwd; break; else cd ../; fi; done)
5
6 CSOURCE := $(shell ls *.c)
7
8 include $(ROOT_DIR)/Makefile.inc

```

```

:
```

A.11.2 src/ast/Makefile

```

1
2 ROOT_DIR := $(shell for i in `seq 0 9`; \
3   do if [ -n "`ls | grep Makefile.inc`" ]; \
4     then pwd; break; else cd ../; fi; done)
5
6 CSOURCE := $(shell ls *.c)
7
8 include $(ROOT_DIR)/Makefile.inc

```

```

:
```

A.11.3 src/Makefile

```

1
2 ROOT_DIR := $(shell for i in `seq 0 9`; \
3   do if [ -n "`ls | grep Makefile.inc`" ]; \
4     then pwd; break; else cd ../; fi; done)
5
6 BIN_NAME := vitality
7 AST_DIR := ast
8 AIA_DIR := aia
9 X86_32_DIR := x86_32
10 PARSE_DIR := parser
11 PARSE_GEN_DIR := $(PARSE_DIR)/gen
12 TEST_DIR := test
13 VIT_DIR := vit
14
15 EXTRA_BUILD_TARGETS := $(BIN_NAME) vitalitysrc
16 EXTRA_CLEAN_TARGETS := $(BIN_NAME)_clean vitalitysrc_clean
17 NEEDED_PROGRAMS := $(FLEX) $(BISON)
18 DEPDIRS := $(PARSE_DIR) $(PARSE_GEN_DIR) $(AIA_DIR) $(AST_DIR) \
19   $(X86_32_DIR) $(TEST_DIR)
20
21 SCAN_C := $(PARSE_GEN_DIR)/scanner.yy.c
22 PARSE_C := $(PARSE_GEN_DIR)/parser.tab.c
23
24 CSOURCE := $(shell ls *.c) $(SCAN_C) $(PARSE_C)
25
26 include $(ROOT_DIR)/Makefile.inc
27

```

```

28 LD = $(CC)
29 LIBS := -lfl
30 LFLAGS := -m$(ARCH)
31
32 BINARY_DIR := $(shell cd $(OUT_DIR)/.. 2> /dev/null; pwd)
33 BINARY := $(BINARY_DIR)/$(BIN_NAME)
34
35 $(BIN_NAME): $(BINARY)
36
37 $(BINARY): $(COBJ) $(OUT_DIR)/$(AST_DIR)/*.o $(OUT_DIR)/$(AIA_DIR)/*.o \
38   $(OUT_DIR)/$(TEST_DIR)/*.o $(OUT_DIR)/$(X86_32_DIR)/*.o
39 ifeq ($(VERBOSE), y)
40   $(LD) $(LFLAGS) -o $$@ $$^ $(LIBS)
41 else
42   @echo LD $$@
43   @$(LD) $(LFLAGS) -o $$@ $$^ $(LIBS)
44 endif
45
46 $(BIN_NAME)_clean:
47 ifeq ($(VERBOSE), y)
48   $(RM) $(BINARY)
49 else
50   @echo clean $(BINARY_DIR)
51   @$(RM) $(BINARY)
52 endif
53
54 vitalysrc:
55   @$(MAKE) -C $(VIT_DIR)
56
57 vitalysrc_clean:
58   @$(MAKE) -C $(VIT_DIR) clean

```

:

A.11.4 src/parser/gen/Makefile

```

1
2 ROOT_DIR := $(shell for i in `seq 0 9`; \
3   do if [ -n "`ls | grep Makefile.inc`" ]; \
4     then pwd; break; else cd ..; fi; done)
5
6 include $(ROOT_DIR)/Makefile.inc

```

:

A.11.5 src/parser/Makefile

```

1
2 ROOT_DIR := $(shell for i in `seq 0 9`; \
3   do if [ -n "`ls | grep Makefile.inc`" ]; \
4     then pwd; break; else cd ..; fi; done)
5
6 LEX = flex
7 YACC = bison
8
9 EXTRA_BUILD_TARGETS := parser_source
10 EXTRA_CLEAN_TARGETS := scan_parse_clean
11 NEEDED_PROGRAMS := $(LEX) $(YACC)
12 DEPDIRS := gen
13
14 include $(ROOT_DIR)/Makefile.inc
15
16 GEN_DIR := $(WORKING_DIR)/gen
17 SCAN_C := $(GEN_DIR)/scanner.yy.c
18 PARSE_C := $(GEN_DIR)/parser.tab.c
19 PARSE_H := $(GEN_DIR)/parser.tab.h
20 PARSE_OUT := $(GEN_DIR)/parser.output

```



```

21
22 parser_source: $(PARSE_C) $(SCAN_C)
23
24 $(SCAN_C): scanner.l
25 ifeq ($(VERBOSE), y)
26     $(LEX) -o $(SCAN_C) scanner.l
27 else
28     @echo LEX $(SCAN_C)
29     @$ (LEX) -o $(SCAN_C) scanner.l
30 endif
31
32 $(PARSE_C): parser.y
33 ifeq ($(VERBOSE), y)
34     $(YACC) -vd -o $(PARSE_C) --defines=$(PARSE_H) \
35         --report-file=$(PARSE_OUT) parser.y
36 else
37     @echo YACC $(PARSE_C)
38     @$ (YACC) -vd -o $(PARSE_C) --defines=$(PARSE_H) \
39         --report-file=$(PARSE_OUT) parser.y
40 endif
41
42 scan_parse_clean:
43 ifeq ($(VERBOSE), y)
44     $(RM) $(SCAN_C)
45     $(RM) $(PARSE_C)
46     $(RM) $(PARSE_H)
47     $(RM) $(PARSE_OUT)
48 else
49     @echo clean $(GEN_DIR)
50     @$ (RM) $(SCAN_C)
51     @$ (RM) $(PARSE_C)
52     @$ (RM) $(PARSE_H)
53     @$ (RM) $(PARSE_OUT)
54 endif

```

:

A.11.6 src/test/Makefile

```

1
2 ROOT_DIR := $(shell for i in `seq 0 9`; \
3     do if [ -n "`ls | grep Makefile.inc`" ]; \
4     then pwd; break; else cd ..; fi; done)
5
6 CSOURCE := $(shell ls *.c)
7
8 include $(ROOT_DIR)/Makefile.inc

```

:

A.11.7 src/vit/Makefile

```

1
2 ROOT_DIR := $(shell for i in `seq 0 9`; \
3     do if [ -n "`ls | grep Makefile.inc`" ]; \
4     then pwd; break; else cd ..; fi; done)
5
6 LIBVIT_DIR := std
7 LIBVIT_THREAD_DIR := _vit_thread
8
9 EXTRA_BUILD_TARGETS := startfiles libvitaly
10 EXTRA_CLEAN_TARGETS := startfiles_clean libvitaly_clean
11
12 include $(ROOT_DIR)/Makefile.inc
13
14 CSOURCE := lib.c
15 COBJ := $(CSOURCE:%.c=$(OUT_DIR)/%.o)

```

```

16
17 override CFLAGS := -O4 -Wall -Wextra -c -m32 -std=c99 -g
18
19 SSOURCE := ini.s vitmain.s retmain.s end.s # lib.s
20 SOBJ := $(SSOURCE:%.s=$(OUT_DIR)/%.o)
21
22 startfiles: $(SOBJ) $(COBJ)
23
24 $(SOBJ): $(OUT_DIR)/%.o: %.s
25 ifeq ($(VERBOSE), y)
26     $(AS) $(SFLAGS) -o $@ $(@:$(OUT_DIR)/%.o=%.s)
27 else
28     @echo AS $(@:$(OUT_DIR)/%.o=$(shell pwd)/%.s)
29     @$(AS) $(SFLAGS) -o $@ $(@:$(OUT_DIR)/%.o=%.s)
30 endif
31
32 $(COBJ): $(OUT_DIR)/%.o: %.c
33 ifeq ($(VERBOSE), y)
34     $(CC) $(CFLAGS) -o $@ $(@:$(OUT_DIR)/%.o=%.c)
35 else
36     @echo CC $(@:$(OUT_DIR)/%.o=$(shell pwd)/%.c)
37     @$(CC) $(CFLAGS) -o $@ $(@:$(OUT_DIR)/%.o=%.c)
38 endif
39
40 startfiles_clean:
41 ifeq ($(VERBOSE), y)
42     $(RM) $(SOBJ)
43 else
44     @echo clean $(OUT_DIR)
45     @$(RM) $(SOBJ)
46 endif
47
48 libvitaly:
49     @$(MAKE) -C $(LIBVIT_DIR)
50     @$(MAKE) -C $(LIBVIT_THREAD_DIR)
51
52 libvitaly_clean:
53     @$(MAKE) -C $(LIBVIT_DIR) clean
54     @$(MAKE) -C $(LIBVIT_THREAD_DIR) clean

```

:

A.11.8 src/vit/std/c/Makefile

```

1
2 ROOT_DIR := $(shell for i in `seq 0 9`; \
3 do if [ -n "`ls | grep Makefile.inc`" ]; \
4 then pwd; break; else cd ..; fi; done)
5
6 include $(ROOT_DIR)/Makefile.inc

```

:

A.11.9 src/vit/std/Makefile

```

1
2 ROOT_DIR := $(shell for i in `seq 0 9`; \
3 do if [ -n "`ls | grep Makefile.inc`" ]; \
4 then pwd; break; else cd ..; fi; done)
5
6 EXTRA_BUILD_TARGETS := libvitaly
7 EXTRA_CLEAN_TARGETS := libvitaly_clean
8 DEPDIRS := c
9
10 include $(ROOT_DIR)/Makefile.inc
11
12 override CSOURCE := cerrno.c c/cstdio.c

```

```

13 override COBJ := $(CSOURCE:%.c=$(OUT_DIR)/%.o)
14 override CFLAGS := -Wall -g -c -m32
15
16 VITSOURCE := string.vit c/string.vit object.vit indexable.vit \
17             array.vit comparator.vit sort.vit vector.vit errno.vit math.vit \
18             stdio.vit c/stdio.vit c/ctype.vit stdlib.vit
19 VITOBJ := $(VITSOURCE:%.vit=$(OUT_DIR)/%.o)
20 VITI := $(VITSOURCE:%.vit=$(OUT_DIR)/%.viti)
21 VITFLAGS := -c -I --lib-init
22 VITALY := $(ROOT_DIR)/bin/vitaly
23 LIBVIT := $(OUT_DIR)/../libvitaly.a
24
25 libvitaly: $(LIBVIT) viti_mv
26
27 $(LIBVIT): $(COBJ) $(VITOBJ)
28 ifeq ($(VERBOSE), y)
29     $(AR) ruv $@ $^
30     ranlib $@
31 else
32     @echo AR $@
33     @$ (AR) ruv $@ $^ > /dev/null
34     @ranlib $@
35 endif
36
37 .PHONY: $(VITOBJ)
38 $(VITOBJ):
39 ifeq ($(VERBOSE), y)
40     $(VITALY) $(VITFLAGS) -o $@ $(@:$(OUT_DIR)/%.o=%.vit)
41 else
42     @echo VITALY $(@:$(OUT_DIR)/%.o=$(shell pwd)/%.vit)
43     @$ (VITALY) $(VITFLAGS) -o $@ $(@:$(OUT_DIR)/%.o=%.vit)
44 endif
45
46 .PHONY: $(COBJ)
47 $(COBJ):
48 ifeq ($(VERBOSE), y)
49     $(CC) $(CFLAGS) -o $@ $(@:$(OUT_DIR)/%.o=%.c)
50 else
51     @echo CC $(@:$(OUT_DIR)/%.o=$(shell pwd)/%.c)
52     @$ (CC) $(CFLAGS) -o $@ $(@:$(OUT_DIR)/%.o=%.c)
53 endif
54
55 viti_mv:
56 ifeq ($(VERBOSE), y)
57     mv *.viti $(OUT_DIR)
58     mv c/*.viti $(OUT_DIR)/c
59 else
60     @mv *.viti $(OUT_DIR)
61     @mv c/*.viti $(OUT_DIR)/c
62 endif
63
64 libvitaly_clean:
65 ifeq ($(VERBOSE), y)
66     $(RM) $(VITI)
67     $(RM) $(VITOBJ)
68     $(RM) $(LIBVIT)
69 else
70     @echo clean $(OUT_DIR)
71     @$ (RM) $(VITI)
72     @$ (RM) $(VITOBJ)
73     @$ (RM) $(LIBVIT)
74 endif

```

:

A.11.10 src/vit/_vit_thread/Makefile

```

1
2 ROOT_DIR := $(shell for i in `seq 0 9`; \
3   do if [ -n "`ls | grep Makefile.inc`" ]; \
4     then pwd; break; else cd ../; fi; done)

```

```

5
6 DEPDIRS = std
7
8 include $(ROOT_DIR)/Makefile.inc

```

```

:
```

A.11.11 src/vit/_vit_thread/std/Makefile

```

1
2 ROOT_DIR := $(shell for i in `seq 0 9`; \
3   do if [ -n "`ls | grep Makefile.inc`" ]; \
4     then pwd; break; else cd ../; fi; done)
5
6 EXTRA_BUILD_TARGETS := libvitaly_thread
7 EXTRA_CLEAN_TARGETS := libvitaly_thread_clean
8
9 include $(ROOT_DIR)/Makefile.inc
10
11 override CSOURCE := cthread.c clock.c
12 override COBJ := $(CSOURCE:%.c=$(OUT_DIR)/%.o)
13 override CFLAGS := -Wall -g -c -m32 -pthread -D_GNU_SOURCE
14
15 VITSOURCE := thread.vit lock.vit
16 VITOBJ := $(VITSOURCE:%.vit=$(OUT_DIR)/%.o)
17 VITI := $(VITSOURCE:%.vit=$(OUT_DIR)/%.viti)
18 VITFLAGS := -c -I --lib-init --thread
19 VITALY := $(ROOT_DIR)/bin/vitaly
20 LIBVIT_THREAD := $(OUT_DIR)/../../libvitaly-thread.a
21
22 $(OUT_DIR)/thread.o: VITFLAGS += --dump=c-header
23 $(OUT_DIR)/lock.o: VITFLAGS += --dump=c-header
24
25 libvitaly_thread: $(LIBVIT_THREAD) viti_mv
26
27 $(LIBVIT_THREAD): $(COBJ) $(VITOBJ)
28 ifeq ($(VERBOSE), y)
29   $(AR) ruv $@ $^
30   ranlib $@
31 else
32   @echo AR $@
33   @$(AR) ruv $@ $^ > /dev/null
34   @ranlib $@
35 endif
36
37 .PHONY: $(VITOBJ)
38 $(VITOBJ):
39 ifeq ($(VERBOSE), y)
40   $(VITALY) $(VITFLAGS) -o $@ $(@:$(OUT_DIR)/%.o=%.vit)
41 else
42   @echo VITALY $(@:$(OUT_DIR)/%.o=$(shell pwd)/%.vit)
43   @$(VITALY) $(VITFLAGS) -o $@ $(@:$(OUT_DIR)/%.o=%.vit)
44 endif
45
46 .PHONY: $(COBJ)
47 $(COBJ): $(VITOBJ)
48 ifeq ($(VERBOSE), y)
49   $(CC) $(CFLAGS) -o $@ $(@:$(OUT_DIR)/%.o=%.c)
50 else
51   @echo CC $(@:$(OUT_DIR)/%.o=$(shell pwd)/%.c)
52   @$(CC) $(CFLAGS) -o $@ $(@:$(OUT_DIR)/%.o=%.c)
53 endif
54
55 viti_mv:
56 ifeq ($(VERBOSE), y)
57   mv *.viti $(OUT_DIR)
58 else
59   @mv *.viti $(OUT_DIR)
60 endif
61
62 libvitaly_thread_clean:

```

```

63 ifeq ($(VERBOSE), y)
64   $(RM) $(VITI)
65   $(RM) $(VITOBJ)
66   $(RM) $(LIBVIT)
67   $(RM) *.vitaly.*
68 else
69   @echo clean $(OUT_DIR)
70   @echo clean $(shell pwd)
71   @$(RM) $(VITI)
72   @$(RM) $(VITOBJ)
73   @$(RM) $(LIBVIT)
74   @$(RM) *.vitaly.*
75 endif

```

:

A.11.12 src/x86_32/Makefile

```

1
2 ROOT_DIR := $(shell for i in `seq 0 9`; \
3   do if [ -n "`ls | grep Makefile.inc`" ]; \
4     then pwd; break; else cd ../; fi; done)
5
6 CSOURCE := $(shell ls *.c)
7
8 include $(ROOT_DIR)/Makefile.inc

```

A.12 Vitaly Standard Library

:

A.12.1 src/vit/std/array.vit

```

1 package std;
2
3 import std.indexable;
4 import std.object;
5 import std.errno;
6
7 type Array = record of Indexable {
8   _ary:array of Object;
9
10   func record (size:int):void
11     if (size < 0) then {
12       errno.set(einval());
13       _ary = null;
14       return;
15     }
16     allocate _ary of length size;
17   end record
18
19   func get(i:int):Object
20     if (!record[]._verifyIdx(i)) then
21       return null;
22     return _ary[i];
23   end get
24
25   func set(i:int, obj:Object):bool
26     if (!record[]._verifyIdx(i)) then
27       return false;
28     _ary[i] = obj;
29     return true;
30   end set
31
32   func _verifyIdx(i:int):bool

```

```

33     if (_ary == null) then {
34         errno.set(efault());
35         return false;
36     }
37
38     if (i < 0 || i >= size()) then {
39         errno.set(einval());
40         return false;
41     }
42
43     return true;
44 end _verifyIdx
45
46 func size():int
47     if (_ary == null) then {
48         errno.set(efault());
49         return 0 - 1;
50     }
51     return |_ary|;
52 end size
53
54 func destroy():void
55     i:int = 0;
56     while (i < size()) do {
57         delete get(i);
58         i = i + 1;
59     }
60     delete record;
61 end destroy
62
63 func finalize():void
64     delete _ary;
65 end finalize
66 };

```

:

A.12.2 src/vit/std/cerrno.c

```

1  #include <errno.h>
2
3  int _Vit_get_errno()
4  {
5      return errno;
6  }
7
8  void _Vit_set_errno(int no)
9  {
10     errno = no;
11 }

```

:

A.12.3 src/vit/std/c/cstdio.c

```

1  #include <stdio.h>
2  #include <stdbool.h>
3
4  int _Vit_gets(void *dest, int size)
5  {
6      size_t n = fread(dest, 1, (size_t)size, stdin);
7      return n;
8  }
9
10 bool _Vit_feof()
11 {
12     return feof(stdin);

```

```

13 }
14
15 bool _Vit_ferror()
16 {
17     return ferror(stdin);
18 }

```

:

A.12.4 src/vit/std/c/ctype.vit

```

1 package std.c;
2
3 extern(C) func isspace(i:int):int;
4
5 extern(C) func isblank(i:int):int;

```

:

A.12.5 src/vit/std/c/stdio.vit

```

1 package std.c;
2
3 extern(C) func printf(fmt:string):int;
4 extern(C) func puts(s:string):int;
5 extern(C) func getchar():int;

```

:

A.12.6 src/vit/std/c/string.vit

```

1 package std.c;
2
3 import std.object;
4
5 extern(C) func strcmp(lhs:string, rhs:string):int;
6 extern(C) func strlen(s:string):int;
7
8 extern(C) func strlen(s:string, maxLen:int):int;
9
10 extern(C) func memcpy(dest:array of char, src:string, size:int):void;
11 extern(C) func memcpy(dest:array of char, src:array of char, size:int):void;
12
13 func memcpy(dest:array of Object, src:array of Object, size:int):void
14     extern(C) func memcpy(dest:array of Object, src:array of Object, size:int):void
15     ;
16     memcpy(dest, src, size * 4);
17 end memcpy

```

:

A.12.7 src/vit/std/comparator.vit

```

1 package std;
2
3 import object;
4
5 type Comparator = record of {
6     func compare(obj:Object, obj2:Object):int
7     return 0;
8 end compare

```

```
9 };
```

```
    :
```

A.12.8 src/vit/std/errno.vit

```
1 package std;
2
3 import std.string;
4 import std.object;
5
6 extern(C) func strerror(no:int):string;
7
8 func eperm():int
9     return 1;
10 end eperm
11 func enoent():int
12     return 2;
13 end enoent
14 func esrch():int
15     return 3;
16 end esrch
17 func eintr():int
18     return 4;
19 end eintr
20 func eio():int
21     return 5;
22 end eio
23 func enxio():int
24     return 6;
25 end enxio
26 func e2big():int
27     return 7;
28 end e2big
29 func enoexec():int
30     return 8;
31 end enoexec
32 func ebadf():int
33     return 9;
34 end ebadf
35 func echild():int
36     return 10;
37 end echild
38 func eagain():int
39     return 11;
40 end eagain
41 func enomem():int
42     return 12;
43 end enomem
44 func eacces():int
45     return 13;
46 end eacces
47 func efault():int
48     return 14;
49 end efault
50 func enotblk():int
51     return 15;
52 end enotblk
53 func ebusy():int
54     return 16;
55 end ebusy
56 func eexist():int
57     return 17;
58 end eexist
59 func exdev():int
60     return 18;
61 end exdev
62 func enodev():int
63     return 19;
64 end enodev
65 func enotdir():int
```



```

66     return 20;
67 end enotdir
68 func eisdir():int
69     return 21;
70 end eisdir
71 func einval():int
72     return 22;
73 end einval
74 func enfile():int
75     return 23;
76 end enfile
77 func emfile():int
78     return 24;
79 end emfile
80 func enotty():int
81     return 25;
82 end enotty
83 func etxtbsy():int
84     return 26;
85 end etxtbsy
86 func efbig():int
87     return 27;
88 end efbig
89 func enospc():int
90     return 28;
91 end enospc
92 func espipe():int
93     return 29;
94 end espipe
95 func erofs():int
96     return 30;
97 end erofs
98 func emlink():int
99     return 31;
100 end emlink
101 func epipe():int
102     return 32;
103 end epipe
104 func edom():int
105     return 33;
106 end edom
107 func erange():int
108     return 34;
109 end erange
110
111 func errno():int
112     extern(C) func _Vit_get_errno():int;
113     return _Vit_get_errno();
114 end errno
115
116 func errno(no:int):void
117     extern(C) func _Vit_set_errno(no:int):void;
118     _Vit_set_errno(no);
119 end errno
120
121 type Errno = record of Object {
122
123     func set(no:int):int
124         old:int = get();
125         errno(no);
126         return old;
127     end set
128
129     func clear():int
130         return set(0);
131     end clear
132
133     func get():int
134         return errno();
135     end get
136
137     func appendTo(dest:String):void
138         dest.append(strerror(get()));
139     end appendTo

```

```

140
141     func assignTo(dest:String):void
142         dest.assign(strerror(get()));
143     end assignTo
144
145     func finalize():void
146     end finalize
147 };
148
149 errno:Errno;
150 allocate errno;
151 finalize delete errno;

```

:

A.12.9 src/vit/std/indexable.vit

```

1 package std;
2
3 import object;
4 import string;
5 import stdio;
6
7 type Indexable = record of Object{
8     _str:String;
9
10    func get(i:int):Object
11        return null;
12    end get
13
14    func set(i:int, o:Object):bool
15        return false;
16    end set
17
18    func size():int
19        return 0;
20    end size
21
22    func finalize():void
23    end finalize
24 };

```

:

A.12.10 src/vit/std/math.vit

```

1 package std;
2
3 func mod(nom:int, denom:int):int
4     return nom - (nom / denom) * denom;
5 end mod

```

:

A.12.11 src/vit/std/object.vit

```

1 package std;
2
3 import string;
4 import stdio;
5
6 type Object = record of {
7     func finalize():void
8     end finalize

```

```
9 };
```

```
    :
```

A.12.12 src/vit/std/sort.vit

```
1 package std;
2
3 import object;
4 import indexable;
5 import comparator;
6
7 func sort(ary:Indexable, c:Comparator):void
8     func exchange(i:int, j:int):void
9         if (i == j) then
10             return;
11         tmp:Object = ary.get(i);
12         ary.set(i, ary.get(j));
13         ary.set(j, tmp);
14     end exchange
15
16     func partition(f:int, l:int):int
17         i:int = f-1;
18         j:int = f;
19         while (j <= l - 1) do {
20             if (c.compare(ary.get(j), ary.get(l)) < 0) then {
21                 i = i+1;
22                 exchange(i, j);
23             }
24             j = j+1;
25         }
26         exchange(i + 1, l);
27         return i + 1;
28     end partition
29
30     func quicksort(f:int, l:int):void
31         if (f < l) then {
32             q:int = partition(f, l);
33             quicksort(f, q - 1);
34             quicksort(q + 1, l);
35         }
36     end quicksort
37
38     quicksort(0, ary.size() - 1);
39 end sort
```

```
    :
```

A.12.13 src/vit/std/stdio.vit

```
1 package std;
2
3 import errno;
4 import std.c.stdio;
5 import object;
6 import string;
7 import std.c.ctype;
8
9 type StdOstream = record of {
10
11     func put(str:String):StdOstream
12         if (str != null) then {
13             put(str.str());
14         } else {
15             printf("null");
16         }
17     return record;
```

```
18     end put
19
20     func put(e:Errno):StdOstream
21         s:String;
22         allocate s;
23         e.assignTo(s);
24         put(s);
25         delete s;
26         return record;
27     end put
28
29     func put(i:int):StdOstream
30         s:String;
31         allocate s of record(i);
32         put(s);
33         delete s;
34         return record;
35     end put
36
37     func put(c:char):StdOstream
38         s:String;
39         allocate s of record(c);
40         put(s);
41         delete s;
42         return record;
43     end put
44
45     func put(b:bool):StdOstream
46         s:String;
47         allocate s of record(b);
48         put(s);
49         delete s;
50         return record;
51     end put
52
53     func put(s:string):StdOstream
54         printf(s);
55         return record;
56     end put
57
58     func put(a:array of char):StdOstream
59         s:String;
60         allocate s of record(a);
61         put(s);
62         delete s;
63         return record;
64     end put
65
66     func putln(obj:String):StdOstream
67         put(obj);
68         return ln();
69     end putln
70
71     func putln(e:Errno):StdOstream
72         put(e);
73         return ln();
74     end putln
75
76     func putln(i:int):StdOstream
77         put(i);
78         return ln();
79     end putln
80
81     func putln(c:char):StdOstream
82         put(c);
83         return ln();
84     end putln
85
86     func putln(b:bool):StdOstream
87         put(b);
88         return ln();
89     end putln
90
91     func putln(s:string):StdOstream
```

```

92     puts(s);
93     return record;
94 end putln
95
96 func putln(a:array of char):StdOstream
97     put(a);
98     return ln();
99 end putln
100
101 func ln():StdOstream
102     printf("\n");
103     return record;
104 end ln
105 };
106
107 stdo:StdOstream;
108 allocate stdo;
109 finalize delete stdo;
110
111 extern(C) func _Vit_gets(dest:array of char, size:int):int;
112 extern(C) func _Vit_feof():bool;
113 extern(C) func _Vit_ferror():bool;
114
115 type StdIstream = record of {
116     _next:char;
117
118     func _skipSpace():void
119         _next = getchar();
120         while (isspace(_next) != 0) do
121             _next = getchar();
122         end _skipSpace
123
124     func get(dest:String):StdIstream
125         _skipSpace();
126         while (isspace(_next) == 0 && _next != 0 - 1) do {
127             dest.append(_next);
128             _next = getchar();
129         }
130         dest.append(cast(char) 0);
131         return record;
132     end get
133
134     func getln(dest:String):StdIstream
135         _next = getchar();
136         while (_next != 10 && _next != 0 - 1) do {
137             dest.append(_next);
138             _next = getchar();
139         }
140         return record;
141     end getln
142 };
143
144 stdi:StdIstream;
145 allocate stdi;
146 finalize delete stdi;

```

:

A.12.14 src/vit/std/stdlib.vit

```

1 package std;
2
3 extern(C) func exit(status:int):void;

```

:

A.12.15 src/vit/std/string.vit

```

1  package std;
2
3  import c.string;
4  import object;
5  import math;
6
7  func intToString(i:int):String
8      ret:String;
9      allocate ret;
10     orig:int = i;
11     if (i == 0) then {
12         ret.append("0");
13         return ret;
14     }
15
16     if (i < 0) then
17         i = 0 - i;
18
19     while (i > 0) do {
20         m:int = mod(i, 10);
21         ret.append(cast(char) (m + 48));
22         i = i / 10;
23     }
24
25     if (orig < 0) then
26         ret.append("-");
27
28     i = 0;
29     j:int = ret.len() - 1;
30     while i < j do {
31         tmp:char = ret._str[j];
32         ret._str[j] = ret._str[i];
33         ret._str[i] = tmp;
34         i = i + 1;
35         j = j - 1;
36     }
37
38     return ret;
39 end intToString
40
41 func charToString(c:char):String
42     a:array of char;
43     allocate a of length 1;
44     a[0] = c;
45
46     ret:String;
47     allocate ret of record(a);
48
49     delete a;
50     return ret;
51 end charToString
52
53 type String = record of Object {
54     _str:array of char;
55     _len:int;
56
57     func record():void
58         record[]._init("", 0);
59     end record
60
61     func record(s:string):void
62         record[]._init(s, strlen(s));
63     end record
64
65     func record(oth:String):void
66         if oth == null then
67             record[]._init("null", 4);
68         else
69             record[]._init(oth.str(), oth.len());
70         end record
71
72     func record(a:array of char):void
73         len:int = strlen(cast(string) a, |a|);

```

```

74     record[]._init(cast(string) a, len);
75 end record
76
77 func record(b:bool):void
78     if (b) then
79         record[]._init("true", 4);
80     else
81         record[]._init("false", 5);
82     end record
83
84 func record(c:char):void
85     tmp:String = charToString(c);
86     record[]._init(tmp.str(), tmp.len());
87     delete tmp;
88 end record
89
90 func record(i:int):void
91     s:String = intToString(i);
92     record[]._init(s.str(), s.len());
93     delete s;
94 end record
95
96 func _init(s:string, len:int):void
97     if len < 0 then
98         len = 0;
99
100     _len = len;
101     allocate _str of length len + 1;
102     memcpy(_str, s, len);
103     _str[len] = 0;
104 end _init
105
106 func append(s:string, len:int):String
107     if len <= 0 then
108         return record;
109
110     offset:int = _len;
111     newLen:int = offset + len;
112
113     if newLen >= |_str| then
114         _expand(newLen);
115
116     i:int = 0;
117     while offset < newLen do {
118         _str[offset] = s[i];
119         i = i + 1;
120         offset = offset + 1;
121     }
122     _str[offset] = 0;
123     _len = newLen;
124
125     return record;
126 end append
127
128 func append(i:int):String
129     s:String = intToString(i);
130     append(s);
131     delete s;
132     return record;
133 end append
134
135 func append(c:char):String
136     s:String = charToString(c);
137     append(s);
138     delete s;
139     return record;
140 end append
141
142 func append(b:bool):String
143     if (b) then
144         return append("true", 4);
145     else
146         return append("false", 5);
147 end append

```

```

148
149     func append(s:string):String
150         return append(s, strlen(s));
151     end append
152
153     func append(oth:String):String
154         return append(oth.str(), oth.len());
155     end append
156
157     func append(a:array of char):String
158         len:int = strlen(cast(string) a, |a|);
159         return append(cast(string) a, len);
160     end append
161
162     func assign(s:string, len:int):String
163         if len < 0 then
164             len = 0;
165
166             size:int;
167             if len < |_str| then
168                 size = |_str|;
169             else
170                 size = len + 1;
171
172             nstr:array of char;
173             allocate nstr of length size;
174
175             memcpy(nstr, s, len);
176             nstr[len] = 0;
177
178             delete _str;
179             _str = nstr;
180             _len = len;
181             return record;
182         end assign
183
184     func assign(s:string):String
185         return assign(s, strlen(s));
186     end assign
187
188     func assign(a:array of char):String
189         len:int = strlen(cast(string) a, |a|);
190         return assign(cast(string) a, len);
191     end assign
192
193     func assign(oth:String):String
194         return assign(oth.str(), oth.len());
195     end assign
196
197     func assign(i:int):String
198         s:String = intToString(i);
199         assign(s);
200         delete s;
201         return record;
202     end assign
203
204     func assign(c:char):String
205         s:String = charToString(c);
206         assign(s);
207         delete s;
208         return record;
209     end assign
210
211     func assign(b:bool):String
212         if (b) then
213             return assign("true", 4);
214         else
215             return assign("false", 5);
216         end assign
217
218     func _expand(newLen:int):void
219         if newLen <= _len then
220             return;
221

```



```

222     tmp:array of char;
223     allocate tmp of length 2*newLen + 1;
224
225     i:int = 0;
226     while i < |_str| do {
227         tmp[i] = _str[i];
228         i = i + 1;
229     }
230
231     delete _str;
232     _str = tmp;
233 end _expand
234
235 func getLast():char
236     return _str[len() - 1];
237 end getLast
238
239 func len():int
240     return _len;
241 end len
242
243 func str():string
244     return cast(string) _str;
245 end str
246
247 func copy():String
248     ret:String;
249     allocate ret of record(record);
250     return ret;
251 end copy
252
253 func compare(oth:String):int
254     return strcmp(cast(string) _str, oth.str());
255 end compare
256
257 func finalize():void
258     delete _str;
259 end finalize
260 };

```

:

A.12.16 src/vit/std/vector.vit

```

1 package std;
2
3 import indexable;
4 import object;
5 import c.string;
6 import array;
7
8 type Vector = record of Array {
9     curIdx:int;
10
11     func record():void
12         record[] (15);
13     end record
14
15     func record(size:int):void
16         record[Array] (size);
17         curIdx = 0;
18     end record
19
20     func size():int
21         return curIdx;
22     end size
23
24     func append(obj:Object):bool
25         size:int = record[Array].size();
26         if (curIdx >= size) then
27             if (!_expand()) then

```

```

28         return false;
29         set(_incIdx(), obj);
30         return true;
31     end append
32
33     func _incIdx():int
34         curIdx = curIdx + 1;
35         return curIdx - 1;
36     end _incIdx
37
38     func _expand():bool
39         tmp:array of Object;
40         allocate tmp of length size() * 2;
41         if (tmp == null) then
42             return false;
43
44         size:int = record[Array].size();
45         if (size < 0) then
46             return false;
47
48         memcpy(tmp, _ary, size);
49
50         delete _ary;
51         _ary = tmp;
52         return true;
53     end _expand
54 };

```

:

A.12.17 src/vit/_vit_thread/std/clock.c

```

1  #include "lock.vit.vitaly.h"
2  #include <pthread.h>
3  #include <stdlib.h>
4
5  bool _Vit_lock_init(struct RT_std_lock_Lock *r)
6  {
7      r->_pthreadMutex = malloc(sizeof(pthread_mutex_t));
8      if (!r->_pthreadMutex)
9          return false;
10     return !pthread_mutex_init(r->_pthreadMutex, NULL);
11 }
12
13 bool _Vit_lock_do(struct RT_std_lock_Lock *r)
14 {
15     if (pthread_mutex_lock(r->_pthreadMutex) == -1)
16         return false;
17     return true;
18 }
19
20 bool _Vit_lock_trydo(struct RT_std_lock_Lock *r)
21 {
22     if (pthread_mutex_trylock(r->_pthreadMutex) == -1)
23         return false;
24     return true;
25 }
26
27 bool _Vit_lock_undo(struct RT_std_lock_Lock *r)
28 {
29     if (pthread_mutex_unlock(r->_pthreadMutex) == -1)
30         return false;
31     return true;
32 }
33
34 bool _Vit_lock_finalize(struct RT_std_lock_Lock *r)
35 {
36     bool ret = !pthread_mutex_destroy(r->_pthreadMutex);
37     free(r->_pthreadMutex);
38     return ret;
39 }

```

:

A.12.18 src/vit/_vit_thread/std/cthread.c

```

1  #include "thread.vit.vitaly.h"
2  #include <pthread.h>
3  #include <stdlib.h>
4  #include <errno.h>
5  #include <stdbool.h>
6  #include <stdio.h>
7  #include <string.h>
8
9  bool _Vit_thread_init(struct RT_std_thread_Thread *t)
10 {
11     t->_pthread = malloc(sizeof(pthread_t));
12     if (!t->_pthread)
13         return false;
14     return true;
15 }
16
17 void _Vit_thread_finalize(struct RT_std_thread_Thread *t)
18 {
19     if (!t->_isStarted)
20         goto out;
21
22     if (pthread_tryjoin_np(*(pthread_t *)t->_pthread), NULL) == -1) {
23         fprintf(stderr, "delete of live thread, aborting\n");
24         abort();
25     }
26
27 out:
28     free(t->_pthread);
29 }
30
31 static void *_Vit_thread_run(void *vit_thread)
32 {
33     struct RT_std_thread_Thread *t = vit_thread;
34     struct RT_std_thread_Thread_vmt *vmt = t->_vmt;
35     vmt->FRT_std_thread_Thread_run(vit_thread);
36     return NULL;
37 }
38
39 bool _Vit_thread_start(struct RT_std_thread_Thread *t)
40 {
41     int err = pthread_create(t->_pthread, NULL, _Vit_thread_run, t);
42     if (err) {
43         errno = err;
44         return false;
45     }
46     return true;
47 }
48
49 bool _Vit_thread_join(struct RT_std_thread_Thread *t)
50 {
51     int err = pthread_join(*(pthread_t *)t->_pthread), NULL);
52     if (err) {
53         errno = err;
54         return false;
55     }
56     return true;
57 }

```

:

A.12.19 src/vit/_vit_thread/std/lock.vit

```

1 package std;
2
3 type Lock = record of {
4   _pthreadMutex:record of {};
5   _error:bool;
6
7   func record():void
8     extern(C) func _Vit_lock_init(r:Lock):bool;
9     _error = !_Vit_lock_init(record);
10  end record
11
12  func lock():bool
13    extern(C) func _Vit_lock_do(r:Lock):bool;
14    return _Vit_lock_do(record);
15  end lock
16
17  func trylock():bool
18    extern(C) func _Vit_lock_trydo(r:Lock):bool;
19    return _Vit_lock_trydo(record);
20  end trylock
21
22  func unlock():bool
23    extern(C) func _Vit_lock_undo(r:Lock):bool;
24    return _Vit_lock_undo(record);
25  end unlock
26
27  func error():bool
28    return _error;
29  end error
30
31  func finalize():void
32    extern(C) func _Vit_lock_finalize(r:Lock):bool;
33    _Vit_lock_finalize(record);
34  end finalize
35 };

```

:

A.12.20 src/vit/_vit_thread/std/thread.vit

```

1 package std;
2
3
4 import std.errno;
5
6 type Thread = record of {
7   _pthread:record of {};
8   _isStarted:bool;
9   _error:bool;
10
11  func record():void
12    extern(C) func _Vit_thread_init(t:Thread):bool;
13    _isStarted = false;
14    _error = !_Vit_thread_init(record);
15  end record
16
17  func run():void
18  end run
19
20  func error():bool
21    return _error;
22  end error
23
24  func start():bool
25    extern(C) func _Vit_thread_start(t:Thread):bool;
26    if error() || started() then {
27      errno.set(eperm());
28      return false;
29    }
30    if _Vit_thread_start(record) then {

```

```
31         _isStarted = true;
32         return true;
33     }
34     if errno.get() != eagain() then
35         _error = true;
36         return false;
37     end start
38
39     func join():bool
40         extern(C) func _Vit_thread_join(t:Thread):bool;
41
42         if error() || !started() then {
43             errno.set(eperm());
44             return false;
45         }
46         return _Vit_thread_join(record);
47     end join
48
49     func started():bool
50         return _isStarted;
51     end started
52
53     func finalize():void
54         extern(C) func _Vit_thread_finalize(t:Thread):void;
55         _Vit_thread_finalize(record);
56     end finalize
57 };
```
