

Garbage Collection With LLVM

Andreas Lynge

January 25, 2016

Contents

1	Introduction	3
2	Motivation	3
3	LLVM	4
3.1	Description	4
3.2	Intermediate Representation	4
4	Implemented SML Subset	5
4.1	Grammar	5
4.2	Summary of Language	8
5	Parsing	11
6	Type Checking	12
6.1	Semantic Error Examples	12
7	Code Generation	13
7.1	Functions	13
7.2	Data Types	16
7.3	Generated LLVM Assembly	17
8	Testing Code Generation	23
8.1	Scope Rules	23
8.2	Common Functions	24
8.3	Binary Search Tree	25
8.4	Foldl with Ref	27
8.5	Foldl for Array	27
9	Garbage Collection	28
10	GC Strategy Implementation	29
10.1	Shadow-stack Strategy	29
10.2	Custom Strategy	30
11	Mark and Sweep Implementation	33
11.1	Buddy Memory Allocator	33
11.2	Marking and Sweeping	39
12	Copying Collector Implementation	41
12.1	Bump-a-pointer Allocator	41
12.2	Copying, Marking and Sweeping	44
13	Generational Collector	47
13.1	Bump-a-pointer Allocator	47
13.2	Minor and Major Garbage Collections	47
13.3	Full LLVM Assembly Listing	49
14	Testing GC Algorithms	53
14.1	Small Allocations	53

14.2 Small and Big Allocations	54
15 GC Algorithm Benchmarks	55
15.1 Recursive Subset Sum	56
15.2 Combination Subset Sum	58
15.3 Iterative Subset Sum	60
16 Conclusion	61
17 Source Code Appendix	62

1 Introduction

In this report I will present a compiler with garbage collection (GC) I have implemented. The compiler is implemented using LLVM. It supports a subset of SML which is useful for testing and benchmarking GC algorithms. In the first part of the report I will explain the implementation of the compiler. In the next part of the report I will explain implementation, test, benchmark and analysis of the GC algorithms supported by the compiler.

This report will discuss:

- what LLVM and LLVM intermediate representation is
- the subset of SML supported by the compiler
- the tools used for parsing
- how the compiler handles infix operators
- how type checking handles pattern matching
- how the code generator generates code using LLVM
- examples of test programs used for testing code generation
- how a GC strategy is used to find GC roots
- implementation of mark and sweep, copying, and generational GC algorithms
- how the GC algorithms are tested for not leaking memory
- benchmarks of the GC algorithms

My report is addressed to readers familiar with compiler construction. The reader is expected to know about parsing, parser generators, abstract syntax trees, and code generation. The reader is supposed acquainted with some of the widely used garbage collection algorithms: mark and sweep, copying and generational. Also, the C and SML programming languages are assumed known.

2 Motivation

Most new programming languages support garbage collection in one way or another. Even C++ has introduced smart-pointers we can use for doing garbage collection with reference counting. When that is not enough, then there are conservative garbage collectors like Boehm's garbage collector which provides tracing garbage collection support for C/C++ programs. Thus, before implementing a serious production compiler; a good understanding of garbage collection will be advantageous.

By implementing different garbage collection algorithms I hope to get an understanding of what it takes to implement garbage collection for a real language.

By benchmarking the GC algorithms I hope to get an understanding of what the benefits and drawbacks of the various algorithms are, this understanding is valuable for future implementations of GC algorithms.

The `clang` compiler, which is now the preferred C/C++/Objective-C compiler on Apple systems, is implemented using LLVM. LLVM is also used by Mozilla for implementing the `rustc` compiler for the Rust programming language. Using LLVM for implementing my compiler I expect to gain valuable insights into the popular compiler infrastructure that LLVM is.

3 LLVM

3.1 Description

LLVM is a collection of C++ libraries which are mainly used for performing compile time and link time optimizations on programs. Although LLVM is mostly used for compiling programs statically, LLVM is also a just-in-time (JIT) compiler.

The normal workflow when using LLVM, is to feed it with LLVM intermediate representation (IR). Given the IR, LLVM performs various optimizations on it and outputs an optimized bytecode file. By further passing a number of bytecode files to LLVM; it can link them together into one executable while performing link-time optimization, or it can execute the bytecode using JIT compilation.

LLVM supports generating binaries for a number of platforms including `x86`, `x86-64`, `arm*` and `mips*`. By generating LLVM IR a compiler can support generating code for all of the supported targets.

In short, LLVM is often used for what we would normally refer to as a compiler's middle-end and back-end.

3.2 Intermediate Representation

The LLVM Intermediate Representation (IR) is designed to be used in three different forms: as an in-memory compiler IR, as an on-disk bytecode representation, and as a human readable assembly language representation.

The LLVM IR is in static single assignment form (SSA) and it has an infinite number of registers/variables. In the human readable assembly language the register names start with `%`. The following example shows how to multiply the register `%x` by 8 and store the result in the register `%res`:

```
1 %res = mul i32 %x, 8
```

Notice that the LLVM IR is typed, in the example above we specify the type of the multiplication operands. If `%x` is not a 32 bit integer, then LLVM will report an error.

We can define a simple function `@fib` to compute the n th Fibonacci number as follows, note that semicolon `;` starts line comments:

```
1  define i32 @fib(i32 %n) {
2      ; Signed compare whether %n is less than 2, and
3      ; store boolean (1 bit integer) result in %1:
4      %1 = icmp slt i32 %n, 2
5      ; Branch to label Done if %n < 2, otherwise branch to Recurse:
6      br i1 %1, label %Done, label %Recurse
7      ; Label Recurse:
8  Recurse:
9      ; Subtract 1 from %n and store result in %3:
10     %3 = sub i32 %n, 1
11     ; Recursive call fib(%n - 1) and store result in %4:
12     %4 = tail call i32 @fib(i32 %3)
13     ; Store %n-2 in %5:
14     %5 = sub i32 %n, 2
15     ; Store call fib(%n - 2) in %6:
16     %6 = tail call i32 @fib(i32 %5)
17     ; Store result of fib(%n-1) + fib(%n-2) in %7:
18     %7 = add i32 %4, %6
19     ; Return result in %7:
20     ret i32 %7
21 ; Label Done:
22 Done:
23     ; Base case, returns 0 or 1:
24     ret i32 %n
25 }
```

Names in the global scope must be prefixed with `@`. When defining functions; we specify the return type of the function, and we specify the parameter types. Notice that the parameters are registers/variables. In the above example we define `@fib` to be a function returning an `i32` (32 bit integer) and taking one `i32` argument.

If we would like to allocate memory on the stack we can use the `alloca` instruction. Later in the report we will see examples of how to use the `alloca` instruction.

4 Implemented SML Subset

The goal of the compiler is to test and benchmark different GC algorithms. The compiler implements a subset of SML which I think is big enough to write some non-trivial SML programs to test GC algorithms, and small enough, such that I do not spend too much time implementing details not related to GC algorithms. Nevertheless, the supported subset is big enough to include some interesting implementation challenges which would not be present if only a "minimal" kind of functional language was supported.

4.1 Grammar

This section contains the grammar of the subset of SML supported by the compiler. Notice that highlighted bar (`|`) is a terminal which is distinct from

the non-highlighted bar used for separating production rules.

$\langle start \rangle$	$\rightarrow \langle decl-seq \rangle$
$\langle decl-seq \rangle$	$\rightarrow \langle decl-seq \rangle \langle colon-opt \rangle \langle decl \rangle$ $\langle decl \rangle$
$\langle decl \rangle$	$\rightarrow \mathbf{val} \langle val-decl \rangle$ $\mathbf{fun} \langle fun-decl \rangle$ $\mathbf{datatype} \langle datatype-decl \rangle$ $\mathbf{nonfix} \langle id-seq \rangle$ $\mathbf{infix} \langle int-lit-opt \rangle \langle id-seq \rangle$ $\mathbf{infixr} \langle int-lit-opt \rangle \langle id-seq \rangle$
$\langle val-decl \rangle$	$\rightarrow \langle pattern \rangle = \langle expr \rangle$
$\langle fun-decl \rangle$	$\rightarrow \langle fun-decl \rangle \mid \langle val-decl \rangle$ $\langle val-decl \rangle$
$\langle datatype-decl \rangle$	$\rightarrow \langle type-var-tuple-opt \rangle \langle type-id \rangle = \langle datatype-def \rangle$
$\langle datatype-def \rangle$	$\rightarrow \langle datatype-def \rangle \mid \langle id \rangle \langle of-type-opt \rangle$ $\langle id \rangle \langle of-type-opt \rangle$
$\langle of-type-opt \rangle$	$\rightarrow \mathbf{of} \langle type \rangle$ ϵ
$\langle type-var-tuple-opt \rangle$	$\rightarrow \langle type-var-tuple \rangle$ ϵ
$\langle type-var-tuple \rangle$	$\rightarrow \mathbf{typevar}$ $(\langle type-var-comma-seq \rangle)$
$\langle type-var-comma-seq \rangle$	$\rightarrow \langle type-var-comma-seq \rangle , \mathbf{typevar}$ $\mathbf{typevar}$
$\langle pattern \rangle$	$\rightarrow \langle pattern \rangle : \langle type \rangle$ $\langle apply-pattern \rangle$
$\langle apply-pattern \rangle$	$\rightarrow \langle apply-pattern \rangle \langle root-pattern \rangle$ $\langle root-pattern \rangle$
$\langle root-pattern \rangle$	$\rightarrow \mathbf{intlit}$ $\overline{\langle long-id \rangle}$ $\mathbf{op} \langle long-id \rangle$ $(\langle pattern-comma-seq-opt \rangle)$ $[\langle pattern-comma-seq-opt \rangle]$
$\langle pattern-comma-seq-opt \rangle$	$\rightarrow \langle pattern-comma-seq \rangle$ ϵ
$\langle pattern-comma-seq \rangle$	$\rightarrow \langle pattern-comma-seq \rangle , \langle pattern \rangle$ $\langle pattern \rangle$
$\langle expr \rangle$	$\rightarrow \mathbf{let} \langle decl-seq \rangle \mathbf{in} \langle expr-colon-seq \rangle \mathbf{end}$ $\mathbf{if} \langle expr \rangle \mathbf{then} \langle expr \rangle \mathbf{else} \langle expr \rangle$

	\mid while $\langle expr \rangle$ do $\langle expr \rangle$ \mid fn $\langle match-seq \rangle$ \mid $\langle expr \rangle$ andalso $\langle expr \rangle$ \mid $\langle expr \rangle$ orelse $\langle expr \rangle$ \mid $\langle apply-expr \rangle$
$\langle apply-expr \rangle$	$\rightarrow \langle apply-expr \rangle \langle root-expr \rangle$ $\mid \langle root-expr \rangle$
$\langle root-expr \rangle$	\rightarrow intlitt $\mid \langle long-id \rangle$ \mid op $\langle long-id \rangle$ \mid ($\langle expr-comma-seq-opt \rangle$) \mid [$\langle expr-comma-seq-opt \rangle$]
$\langle expr-scolon-seq \rangle$	$\rightarrow \langle expr-scolon-seq \rangle ; \langle expr \rangle$ $\mid \langle expr \rangle$
$\langle expr-comma-seq-opt \rangle$	$\rightarrow \langle expr-comma-seq \rangle$ $\mid \epsilon$
$\langle expr-comma-seq \rangle$	$\rightarrow \langle expr-comma-seq \rangle , \langle expr \rangle$ $\mid \langle expr \rangle$
$\langle match-seq \rangle$	$\rightarrow \langle match-seq \rangle \mid \langle match \rangle$ $\mid \langle match \rangle$
$\langle match \rangle$	$\rightarrow \langle pattern \rangle \Rightarrow \langle expr \rangle$
$\langle type \rangle$	$\rightarrow \langle fun-type-2 \rangle$ $\mid \langle prod-type-2 \rangle$ $\mid \langle apply-type \rangle$
$\langle fun-type-2 \rangle$	$\rightarrow \langle fun-type \rangle \rightarrow \langle apply-type \rangle$ $\mid \langle fun-type \rangle \rightarrow \langle prod-type-2 \rangle$
$\langle fun-type \rangle$	$\rightarrow \langle fun-type \rangle \rightarrow \langle apply-type \rangle$ $\mid \langle fun-type \rangle \rightarrow \langle prod-type-2 \rangle$ $\mid \langle apply-type \rangle$ $\mid \langle prod-type-2 \rangle$
$\langle prod-type-2 \rangle$	$\rightarrow \langle prod-type \rangle * \langle apply-type \rangle$
$\langle prod-type \rangle$	$\rightarrow \langle prod-type \rangle * \langle apply-type \rangle$ $\mid \langle apply-type \rangle$
$\langle apply-type \rangle$	$\rightarrow \langle apply-type \rangle \langle root-type \rangle$ $\mid \langle root-type \rangle$
$\langle root-type \rangle$	$\rightarrow \langle long-type-id \rangle$ \mid typevar \mid ($\langle type-comma-seq \rangle$)
$\langle type-comma-seq \rangle$	$\rightarrow \langle type-comma-seq \rangle , \langle type \rangle$ $\mid \langle type \rangle$

$\langle \text{scolon-opt} \rangle$	$\rightarrow ;$ ϵ
$\langle \text{id-seq} \rangle$	$\rightarrow \langle \text{id-seq} \rangle \langle \text{id} \rangle$ $\langle \text{id} \rangle$
$\langle \text{long-id} \rangle$	$\rightarrow \langle \text{long-id} \rangle . \langle \text{id} \rangle$ $\langle \text{id} \rangle$
$\langle \text{id} \rangle$	$\rightarrow \langle \text{type-id} \rangle$ $*$
$\langle \text{long-type-id} \rangle$	$\rightarrow \langle \text{long-type-id} \rangle . \langle \text{type-id} \rangle$ $\langle \text{type-id} \rangle$
$\langle \text{type-id} \rangle$	$\rightarrow \mathbf{identifier}$ $\mathbf{operator}$ $=$
$\langle \text{int-lit-opt} \rangle$	$\rightarrow \mathbf{intlit}$ ϵ

Where:

- **intlit** is any integer literal
- **identifier** is any identifier which is not a keyword and not starting with single quote (').
- **operator** is any operator excluding key operators. Some of the key operators are `*` `->` `.` `=>`
- **typevar** is any identifier starting with one single quote ('). Note that an identifier starting with two single quotes is not a **typevar**
- Other highlighted words and operators are keywords and key operators.

This grammar describes the subset of the language which is implemented by the compiler. In reality, the parser implemented for the compiler is almost capable of parsing the whole SML language, but the type checker and code generator does not support type checking and code generating more than what is described in this grammar.

4.2 Summary of Language

The following subsection summarizes the subset of SML implemented by the compiler:

Values defined with the **val** keyword. Any serious subset of SML should support defining values. Recursive values using **rec** is not supported. It supports recursive function definitions:

Curried functions (optionally recursive) defined with **fun**. Any functional programming language should support functions in some way. And these functions should support recursion. The compiler supports this, but the compiler does not support using the **and** keyword to define mutually recursive functions. If

the user wants to use mutual recursion he must make use of a nested function definition. For example:

```
1 fun factorial 0      : int = 1
2   | factorial(n:int) : int = let
3     fun run() : int = n * factorial(n-1)
4   in
5     run()
6   end
```

If wanted, the user can define operators to be **nonfix**, **infix** or **infixr**.

The compiler supports the expressions **if-then-else**, **let-in-end**, **orelse**, **andalso**, function values using **fn** and the **while-do** loop. It does not support the **case-of** expression.

In order for the **while-do** loop to be useful the compiler also should support mutable values using the **ref** and **array** data types. For the GC algorithms, this means they must be able to handle circular data structures; making reference counting algorithms less attractive.

Other built-in data types supported by the compiler are **int**, **bool**, **unit** and **list**.

The compiler also fully supports defining parameterized and unparameterized data types with the **datatype** keyword. The compiler supports directly recursive data types, but it does not support mutually recursive data types, using the **and** keyword.

Except from the **as**-patterns, there is full pattern matching support, but no support for type inference. Thus when the user writes a pattern, e.g. a function argument, the user must make sure the compiler knows the complete type of the pattern. For example:

```
1 fun id x = x
```

The previous example has two problems. The compiler does not infer **x** to have some polymorphic type **'a** and the compiler does not infer the return type of the **id** function to be **'a**. Thus the correct implementation of **id** is:

```
1 fun id(x: 'a): 'a = x
```

In the following example the compiler knows that the type of the pattern to the left hand side of the equal sign is **int list**:

```
1 val [1,2,3] = [1,2,3]
```

And in the following case the compiler does not:

```
1 val [x,y,z] = [1,2,3]
```

A type specification is needed since the type of **x,y** and **z** in **[x,y,z]** will not be inferred:

```
1 val [x,y,z]: int list = [1,2,3]
```

Or equivalently:

```
1 val [x:int, y:int, z:int] = [1,2,3]
```

As long as type inference is not needed the compiler has full support for patterns. Patterns are used in the `val` statement, the `fn` expression and when defining functions with the `fun` keyword.

When defining a function the user must make sure the compiler will not need to infer types of parameters. The following is an error:

```
1 fun len [] : int = 0
2   | len(_::xs : 'a list): int = 1 + len xs
```

Although one case fully specifies the type of the function parameter and return type, a type specification for the function parameter in the other case is needed. The following modification makes the function definition valid:

```
1 fun len([] : 'a list): int = 0
2   | len(_::xs : 'a list): int = 1 + len xs
```

In order to obey the value restriction the compiler refuses to compile expressions involving function application returning polymorphic types. For example, the program:

```
1 val xs: 'a list ref = ref []
```

is not allowed since `ref` is implemented as a function; the expression `ref []` is a function application returning a polymorphic type.

In order to solve the problem we must give `xs` a concrete type:

```
1 val xs: int list ref = ref []
```

This is not necessary when the polymorphic type is defined by a function parameter. For example, the following is fine:

```
1 fun f(xs: 'a list): 'a list ref =
2   let
3     val ys: 'a list ref = ref xs
4   in
5     ys
6   end
```

Although the function application `ref xs` returns `'a list ref` it is fine because the type `'a` is defined by the function parameter.

Another contrived example, also to show usage of function values, is:

```

1 | val myReverse: 'a list -> 'a list = fn []: 'a list =>
2 |                                     []
3 |                                     | xs: 'a list =>
4 |                                     let
5 |                                     val ret: 'a list = rev xs
6 |                                     in
7 |                                     ret
8 |                                     end

```

In this example, where `rev` is the function which reverses a list, in line 5 we are allowed to return `'a list` because the type of `'a` is defined by the function parameter.

The most significant features of SML missing from this subset of SML are records, exceptions, structures, signatures, functors and type inference. Without these features it is still possible to write some non-trivial GC test programs.

5 Parsing

Parsing is done using the `flex` and `bison` tools. If the parser finds a syntax error it will simply exit with an error message telling where the syntax error occurred. After the parsing phase the compiler has generated an Abstract Syntax Tree (AST).

Afterwards, the AST is passed to the AST Fixup Visitor which is used to convert expressions of the form:

```

1 | 1 + 2*3

```

into the form:

```

1 | op+(1, op*(2, 3))

```

Assuming standard associativity and precedence of the plus and multiplication operators.

When the AST Fixup Visitor receives the AST, then expressions are assumed to be `nonfix` operators. Hence, the expression `1+x` is initially parsed like the literal `1` is a function taking `+` as argument returning a function taking `x` as argument. The job of the Fixup Visitor is to spot that the `+` is an infix operator and convert the expression into `op+(1, x)`.

Obviously, not all expressions are that simple. Thus the Fixup Visitor implements a recursive descent parser which uses a map to store the precedence and associativity of operators. The user can define precedence and associativity of operators using the `infix`, `infixr` and `nonfix` statements.

If the Fixup Visitor finds an error with some function application expression the compiler will terminate with a proper error message.

6 Type Checking

The compiler implements an AST Type Check Visitor. The concern of the Type Check Visitor is to find semantic errors in the source code.

This primarily consists of detecting the type of an expression and detecting the type of a pattern, and afterwards verify that the expression's type is compatible with the pattern's type.

Whenever the Type Check Visitor locates a pattern it builds a Pattern object. Such a Pattern object can have one of several classes. For example the pattern `1` is a literal pattern and the pattern `(x,y)` is a tuple pattern containing two identifier patterns.

The idea behind the pattern object is to be able to, for example, ask the object about which type it has or ask the object about which identifiers are defined by the pattern.

Consider the following example:

```
1 val (x:int,y:int) = (0,0)
```

In this example the Type Check Visitor will build a tuple pattern object containing two identifier pattern objects, each of type `int`. Afterwards it will go through the expression `(0,0)` to determine that its type is `int*int`. Next the Type Check Visitor can query the pattern object for its type to find out if the expression's type can be assigned to the pattern's type, which it can since both types are `int*int`. Next the Type Check Visitor can ask the pattern object about which identifiers it defines and which type these identifiers have. It will put the identifiers and corresponding types in a map. In this example the Type Check Visitor will put `x` and `y` both of type `int` in the map.

Now, if a later expression uses `x` or `y` the Type Check Visitor can lookup in the map and find that they are `ints`.

Note that saying that the Type Check Visitor puts the identifiers in a map is a simplification. In reality the Type Check Visitor puts the identifiers in a map located in the front of a list of maps. This list of maps grows in size when entering a scope and shrinks in size when leaving a scope.

If the Type Check Visitor finds a semantic error, for example if the user uses an undeclared identifier, the compiler will terminate with an error message.

6.1 Semantic Error Examples

This subsection contains two examples of error messages given by the compiler when a semantic error is found.

Given the following program on standard input:

```
1 fun append(x: 'a)(xs: 'a list): 'a list = x::xs
2 val xs: int list = append true [0]
```

the compiler produces the error message:

```
1 <stdin>:2:32: error: argument has unexpected type
```

The first argument passed to `append` has type `bool`, hence `'a` is deduced to be of type `bool`. Now, the second argument should be of type `bool list`, but since the second argument is an `int list` the compiler produces the error message. The error message tells us that the function argument on line 2 column 32 from the file which was read from standard input has unexpected type.

And the following program:

```
1 fun append(x: 'a)(xs: 'a list): 'a list = x::xs
2 val xs: bool list = append 1 [0]
```

gives the error message:

```
1 <stdin>:2:5: error: pattern and expression have conflicting types
```

On line 2 column 5 there is a semantic error. The return type is deduced to be `int list`, but the pattern has type `bool list`.

7 Code Generation

The Code Generation Visitor's job is to generate LLVM intermediate representation. When visiting an expression the Code Generation Visitor generates code for executing the expression and storing the result in memory. By loading that result from memory the result may be used by other expressions or may be used to do pattern matching

7.1 Functions

All user defined functions internally have 2 parameters. The first parameter is the containing/parent function's frame and the second parameter is the user defined function parameter.

In the entry of each function; code is generated to dynamically allocate memory for the frame. The frame is used for storing the local function values and also for storing the parent function's frame. The parent function's frame is used when the current function wants to access a value declared in the parent function's frame. Also, since the parent function's frame contains its parent function's frame, this can be used to access values declared in the grandparent function's frame, etc.

So generating code for a function definition, whether the function is defined with `fun` or `fn`, consists of allocating memory for a pointer to the parent function's frame and a function pointer. In C we can declare a `struct` to represent a function value as follows:

```

1 struct function_value = {
2     void **parent_frame;
3     void *(*value)(void **frame, void *arg);
4 };

```

Calling a user defined function consists of obtaining the argument (**arg**) and afterwards calling the function passing the parent function's frame as first argument and passing **arg** as second argument. An example of generated code for calling a user defined function located at index **X** in the current function's frame using C code is:

```

1 struct function_value *fun = frame[X];
2 void *arg = ...;
3 void *result = (*fun->value)(fun->parent_frame, arg);

```

Now **result** is the return value of calling the function.

In general, except from **int** and **bool**, all data types are pointers. In this way it is easy to store values in function frames and pass values as arguments; they can all be represented as 32/64 bit words on 32/64 bit machines.

If at some point, while executing the program, a pattern does not match an expression the program will terminate with the error message: "pattern match failed".

An example of a failing pattern match is:

```

1 val 1 = 2

```

Another example of a failing pattern match is:

```

1 val () = (fn 1 => ()) 2

```

When generating code for a curried function like:

```

1 fun test 1 2 3 : int = 1
2   | test 2 3 4 : int = 3

```

What happens is that 3 functions are defined. In the following examples we will refer to these functions as **test_1**, **test_2** and **test_3**. The first function (**test_1**) is used to obtain the parent function's frame, allocate memory for the shared frame which is shared by all 3 functions and to store the argument and the parent function's frame in the shared frame. Function **test_1** returns a function value containing the shared frame and a pointer to the second function (**test_2**). A simplified example showing generated code using C syntax is:

```

1 struct function_value *test_1(void **parent_frame, void *arg) {
2     void **shared_frame = malloc(...);
3     shared_frame[0] = parent_frame;
4     shared_frame[1] = arg;
5     struct function_value *fun = malloc(...);
6     fun->parent_frame = shared_frame;
7     fun->value = &test_2;

```

```
8   return fun;
9 }
```

When the second function is called it receives the shared frame as first argument and the user defined argument as second argument. It stores the second argument in the shared frame and returns a function value containing the shared frame and a pointer to the third function. A simplified example of the generated code for the second function using C code is:

```
1 struct function_value *test_2(void **shared_frame, void *arg) {
2     shared_frame[2] = arg;
3     struct function_value *fun = malloc(...);
4     fun->parent_frame = shared_frame;
5     fun->value = &test_3;
6     return fun;
7 }
```

When the third function is called it will receive the shared frame as first argument and receive the user defined argument as second argument. Since this function has the shared frame, this function has access to the parent function's frame and it has access to the arguments passed to `test_1` and `test_2`. The body of this function will consist of pattern matching the arguments with the patterns the user defined when defining `test`. If the arguments match the pattern 1 2 3 the function will return 1, if the argument match the pattern 2 3 4 the function will return 3, otherwise the program execution will terminate with the "pattern match failed" error message. Example continued; showing the generated code for the third function using C code:

```
1 void *test_3(void **shared_frame, void *arg) {
2     if ((long int)shared_frame[1] == 1 &&
3         (long int)shared_frame[2] == 2 &&
4         (long int)arg == 3)
5         return (void *)1;
6     if ((long int)shared_frame[1] == 2 &&
7         (long int)shared_frame[2] == 3 &&
8         (long int)arg == 4)
9         return (void *)3;
10    fprintf(stderr, "pattern match failed");
11    exit(1);
12 }
```

And that is almost exactly how code generation for curried functions is done. We should notice that, if needed, the last function `test_3` can access the parent function's frame using `shared_frame[0]`.

We can see that we can generalize the previous example to using arbitrary argument types and arbitrary number of parameters.

In order to optimize some of the mostly used functions the code generator will locate when expressions like `x+y` and `x*y` occur. If the user has not redefined the `+` or `*` operator the code generator will emit a single instruction like `add` or `mul`, instead of invoking a function when encountering expressions like `x+y` or `x*y`, etc. Note that it is still possible for the user to pass `+` or `*` as a function value if wanted:

```
1 val a: int*int -> int = op+
2 val m: int*int -> int = op*
```

Then `a` is assigned a function value containing a pointer to a runtime function used for adding two integers and `m` is assigned a function value containing a pointer to a runtime function for multiplying two integers.

7.2 Data Types

The user has the option to create his own types using the `datatype` keyword. Such a type is, for simplicity, always represented as a pointer. For example:

```
1 datatype ordering = LT | EQ | GT
```

In the following example:

```
1 val a:ordering = LT
2 val b:ordering = EQ
3 val c:ordering = GT
```

A single word of memory will be dynamically allocated for `a`, `b` and `c`. The word allocated for `a` will be initialized with 0, the word allocated for `b` will be initialized with 1, the word allocated for `c` will be initialized with 2.

In this way we can do pattern matching on `ordering` types by comparing with the value it has been initialized with. For example generating code for the following `val` statement, where `c` is an `ordering` type:

```
1 val EQ = c
```

consists of adding pattern matching code for testing whether the memory word pointed to equals 1. Example of the generated code using C syntax, assuming `c` is the `ordering` type:

```
1 if (*c != 1) {
2     fprintf(stderr, "pattern match failed");
3     exit(1);
4 }
```

User defined type instances can also be parameterized. For example:

```
1 datatype intopt = NoneInt | SomeInt of int
```

In this case when defining the values:

```
1 val n:intopt = NoneInt
2 val s:intopt = SomeInt 7
```

A single memory word for the `n` value is allocated and initialized with 0, and two memory words for the `s` value is allocated; the first word is initialized with 1 and the second word is initialized with the integer 7 as defined by the user.

The generated code for the following `val` statement where `s` has `intopt` type:

```
1 val SomeInt 10 = s
```

can be illustrated using C syntax as follows, assuming `s` is the `intopt` type:

```
1 if (s[0] != 1) {  
2     fprintf(stderr, "pattern match failed");  
3     exit(1);  
4 }  
5 if (s[1] != 10) {  
6     fprintf(stderr, "pattern match failed");  
7     exit(1);  
8 }
```

And for the following `val` statement:

```
1 val SomeInt (x:int) = s
```

code like the following will be generated:

```
1 if (s[0] != 1) {  
2     fprintf(stderr, "pattern match failed");  
3     exit(1);  
4 }  
5 x = s[1];
```

It is also possible to create parameterized types:

```
1 datatype 'a option = NONE | SOME of 'a
```

Generating code for initializing an `option` type and generating pattern matching code for an `option` type is similar with how it was previously done for the `intopt` type.

The `list` type is internally implemented as a cons list as follows:

```
1 datatype 'a list = :: of 'a * 'a list | nil
```

And for convenience, the `[]` square brackets can be used for initializing a list where `[]` is equivalent with `nil`, `[1]` is equivalent with `1::nil`, etc.

7.3 Generated LLVM Assembly

This section contains an example LLVM program which is compiled into the human readable LLVM assembly format such that we can inspect the code the compiler generates. Some parts of the generated LLVM assembly will be left

out as these parts contain code which is generated only to support GC. After having talked about GC we will take a look at a whole LLVM assembly program generated by the compiler. When parts of the generated code has been left out; it will be marked by an ellipsis (...). I have also added comments to the LLVM assembly such that it is easier to understand.

Consider the following program which does not do much when executed, but contains a couple of functions and a `datatype` declaration:

```

1  datatype nat = Zero | Succ of nat
2
3  fun maxImpl Zero      (_:nat)      (_:nat,y:nat) : nat = y
4    | maxImpl(_:nat)    Zero        (x:nat,_:nat) : nat = x
5    | maxImpl(Succ n: nat)(Succ m: nat)(xy: nat*nat) : nat =
6      maxImpl n m xy
7
8  fun max(x:nat)(y:nat): nat = maxImpl x y (x,y)

```

The generated LLVM IR contains 6 functions. The entry function which executes the code in the global scope is called `@entry`, the `maxImpl` is called `@_1maxImpl` and `max` is called `@_4max`. The entry function will allocate memory for storing function values for the `@_1maxImpl` and `@_4max` functions in `@entry`'s frame.

```

1  define i64 @entry() ... {
2    ...
3    ; Allocate stack memory for pointer to @entry's frame:
4    %locals = alloca i64*, align 8
5    ...
6    ; Dynamically allocate memory for @entry's frame:
7    %1 = tail call i64* @allocate(i64 24)
8    ; Store pointer to frame on the stack:
9    store i64* %1, i64** %locals, align 8
10   ...
11   ; Allocate memory for function value for @_1maxImpl:
12   %2 = tail call i64* @allocate(i64 16)
13   ; Pointer to function value converted to i64:
14   %3 = ptrtoint i64* %2 to i64
15   ...
16   ; Store pointer to @_maxImpl at index 0 in the function value:
17   store i64 ptrtoint (i64 (i64, i64)* @_1maxImpl to i64), i64* %2,
      align 8
18   ; Get pointer to index 1 of the function value:
19   %4 = getelementptr i64, i64* %2, i64 1
20   ; Get @entry's frame from the stack and store it in %frame11:
21   %5 = bitcast i64** %locals to i64*
22   %frame11 = load i64, i64* %5, align 8
23   ; Store @entry's frame at index 1 in the function value:
24   store i64 %frame11, i64* %4, align 8
25   ; Get @entry's frame:
26   %frame2 = load i64*, i64** %locals, align 8
27   ; Get pointer to index 1 of @entry's frame:
28   %6 = getelementptr i64, i64* %frame2, i64 1
29   ; Store function value in @entry's frame:
30   store i64 %3, i64* %6, align 8
31   ; Allocate memory for function value for @_4max:
32   %7 = tail call i64* @allocate(i64 16)
33   ; Initialize and store function value for @_4max is analogous
34   ; to how it was done for @_1maxImpl, thus it is left out.

```

```

35     ...
36     ; Return 0 to indicate execution was a success:
37     ret i64 0
38 }

```

Where `i64` is an 64 bit integer type and `i64*` is a pointer to an `i64`.

The LLVM assembly for `maxImpl` is 3 functions called `@_1maxImpl`, `@_2maxImpl`, `@_3maxImpl`:

```

1  define i64 @_1maxImpl(i64, i64) ... {
2      ; Allocate stack memory for temporary storage:
3      %temppptr1 = alloca i64*, align 8
4      ; Convert it from i64** to i64* for a technical reason
5      ; related to GC:
6      %temp2 = bitcast i64** %temppptr1 to i64*
7      ...
8      ; Allocate stack memory for storing frame shared among
9      ; the maxImpl functions (shared frame):
10     %locals = alloca i64*, align 8
11     ; Store second function argument. This is the argument
12     ; which is used for partially applying maxImpl:
13     store i64 %1, i64* %temp2, align 8
14     ...
15     ; Dynamically allocate shared frame:
16     %4 = tail call i64* @allocate(i64 24)
17     ; Store shared frame on the stack:
18     store i64* %4, i64** %locals, align 8
19     ; Store first function argument (parent
20     ; function's frame) in shared frame:
21     store i64 %0, i64* %4, align 8
22     ; Get second function argument from stack:
23     %5 = load i64, i64* %temp2, align 8
24     ; Get shared frame from stack:
25     %frame3 = load i64*, i64** %locals, align 8
26     ; Get pointer to index 1 of shared frame:
27     %6 = getelementptr i64, i64* %frame3, i64 1
28     ; Store second function argument at index 1 of shared frame:
29     store i64 %5, i64* %6, align 8
30     ; Allocate memory for function value for maxImpl
31     ; partially applied 1 argument:
32     %7 = tail call i64* @allocate(i64 16)
33     ; Store pointer to @_2maxImpl (partially applied maxImpl)
34     ; at index 0 of function value:
35     store i64 ptrtoint (i64 (i64, i64)* @_2maxImpl to i64), i64* %7,
        align 8
36     ; Get pointer to index 1 of function value:
37     %8 = getelementptr i64, i64* %7, i64 1
38     ; Get shared frame from stack and store it in %frame41:
39     %9 = bitcast i64** %locals to i64*
40     %frame41 = load i64, i64* %9, align 8
41     ; Store shared frame in function value:
42     store i64 %frame41, i64* %8, align 8
43     ; Convert to function value to i64:
44     %10 = ptrtoint i64* %7 to i64
45     ; Return function value of partially applied maxImpl
46     ; (@_2maxImpl):
47     ret i64 %10
48 }
49
50 define i64 @_2maxImpl(i64, i64) ... {

```

```

51     ...
52     ; Allocate stack memory for shared frame:
53     %locals = alloca i64*, align 8
54     ...
55     ; Get shared frame (first argument):
56     %.cast = inttoptr i64 %0 to i64*
57     ; Store shared frame on stack:
58     store i64* %.cast, i64** %locals, align 8
59     ; Get pointer to index 2 of shared frame:
60     %3 = getelementptr i64, i64* %.cast, i64 2
61     ; Store second argument at index 2 in shared frame:
62     store i64 %1, i64* %3, align 8
63     ; Allocate memory for function value for maxImpl
64     ; partially applied 2 arguments:
65     %4 = tail call i64* @allocate(i64 16)
66     ; Store pointer to @_3maxImpl at index 0 of function value:
67     store i64 ptrtoint (i64 (i64, i64)* @_3maxImpl to i64), i64* %4,
        align 8
68     ; Get pointer to index 1 of function value:
69     %5 = getelementptr i64, i64* %4, i64 1
70     ; Get shared frame from stack and store it in %frame31:
71     %6 = bitcast i64** %locals to i64*
72     %frame31 = load i64, i64* %6, align 8
73     ; Store shared frame at index 1 in function value:
74     store i64 %frame31, i64* %5, align 8
75     %7 = ptrtoint i64* %4 to i64
76     ; Return function value:
77     ret i64 %7
78 }
79
80 define i64 @_3maxImpl(i64, i64) ... {
81     ; Allocate stack memory:
82     %tempptr1 = alloca i64*, align 8
83     ...
84     %temp2 = bitcast i64** %tempptr1 to i64*
85     ...
86     ; Allocate stack memory for frame:
87     %locals = alloca i64*, align 8
88     ...
89     ; Store second argument passed to this function
90     ; on the stack:
91     store i64 %1, i64* %temp2, align 8
92     ...
93     ; Dynamically allocate memory for frame:
94     %11 = tail call i64* @allocate(i64 48)
95     ; Store pointer to frame on the stack:
96     store i64* %11, i64** %locals, align 8
97     ; Store shared frame at index 0 in frame:
98     store i64 %0, i64* %11, align 8
99     ; Get frame:
100    %frame3 = load i64*, i64** %locals, align 8
101    %12 = bitcast i64* %frame3 to i64**
102    ; Get shared frame:
103    %13 = load i64*, i64** %12, align 8
104    ; Get pointer to index 1 of shared frame:
105    %14 = getelementptr i64, i64* %13, i64 1
106    %15 = bitcast i64* %14 to i64**
107    ; Get the argument which was passed to @_1maxImpl from
108    ; index 1 of shared frame:
109    %16 = load i64*, i64** %15, align 8
110    ; Get first word of that argument:
111    %17 = load i64, i64* %16, align 8

```

```

112 ...
113 ; Compare whether the first word of the argument is Zero or Succ:
114 %19 = icmp eq i64 %18, 0
115 ; Branch to label %20 if the argument which was passed
116 ; to @_1maxImpl is Zero, otherwise branch to label %27:
117 br i1 %19, label %20, label %27
118
119 ; <label>:20
120 ; Argument passed to @_1maxImpl was Zero.
121 ; Thus, in this branch the index 1 from the tuple
122 ; which was passed to this function is returned.
123 ; Get tuple which was passed to this function (as
124 ; second argument):
125 %21 = load i64*, i64** %tempPtr1, align 8
126 %22 = getelementptr i64, i64* %21, i64 1
127 %23 = load i64, i64* %22, align 8
128 %24 = getelementptr i64, i64* %frame3, i64 1
129 store i64 %23, i64* %24, align 8
130 %frame6 = load i64*, i64** %locals, align 8
131 %25 = getelementptr i64, i64* %frame6, i64 1
132 %26 = load i64, i64* %25, align 8
133 ret i64 %26
134
135 ; <label>:27
136 ; Argument passed to @_1maxImpl was Succ.
137 ; Get pointer to index 2 of shared frame:
138 %28 = getelementptr i64, i64* %13, i64 2
139 %29 = bitcast i64* %28 to i64**
140 ; Get argument which was passed to @_2maxImpl:
141 %30 = load i64*, i64** %29, align 8
142 ...
143 ; Branch to %33 if argument which was passed to @_2maxImpl
144 ; is Zero, else it is Succ, then branch to %39:
145 br i1 %32, label %33, label %39
146
147 ; <label>:33
148 ; Argument passed to @_2maxImpl was Zero.
149 ; Get tuple which was passed to @_2maxImpl function
150 ; and return index 0 from that tuple:
151 %34 = load i64*, i64** %tempPtr1, align 8
152 %35 = load i64, i64* %34, align 8
153 %36 = getelementptr i64, i64* %frame3, i64 2
154 store i64 %35, i64* %36, align 8
155 %frame10 = load i64*, i64** %locals, align 8
156 %37 = getelementptr i64, i64* %frame10, i64 2
157 %38 = load i64, i64* %37, align 8
158 ret i64 %38
159
160 ; <label>:39
161 ; Arguments passed to @_1maxImpl and @_2maxImpl were both Succ.
162 ; Verify the first word of the argument passed to
163 ; @_1maxImpl is 1:
164 %40 = icmp eq i64 %18, 1
165 ; If it is 1 then branch to %41, otherwise crash program:
166 br i1 %40, label %41, label %fun_match_fail
167
168 ; <label>:41
169 ; Get index 1 of the argument which was passed to @_1maxImpl:
170 %42 = getelementptr i64, i64* %16, i64 1
171 %43 = load i64, i64* %42, align 8
172 %44 = getelementptr i64, i64* %frame3, i64 3
173 ; Store index 1 from the argument passed to @_1maxImpl in frame:

```

```

174 store i64 %43, i64* %44, align 8
175 %frame13 = load i64*, i64** %locals, align 8
176 %45 = bitcast i64* %frame13 to i64**
177 ; Get shared frame from frame:
178 %46 = load i64*, i64** %45, align 8
179 %47 = getelementptr i64, i64* %46, i64 2
180 %48 = bitcast i64* %47 to i64**
181 ; Get word at index 2 from shared frame, that is the argument
182 ; which was passed to @_2maxImpl:
183 %49 = load i64*, i64** %48, align 8
184 ; Get first word of that:
185 %50 = load i64, i64* %49, align 8
186 ...
187 ; Branch to %52 if the argument which was passed to
188 ; @_2maxImpl was a Succ, otherwise crash the program:
189 br i1 %51, label %52, label %fun_match_fail
190
191 ; <label>:52
192 ; Arguments passed to @_1maxImpl and @_2maxImpl are both Succ.
193 %53 = getelementptr i64, i64* %49, i64 1
194 %54 = load i64, i64* %53, align 8
195 %55 = getelementptr i64, i64* %frame13, i64 4
196 ; Store index 1 from the argument passed to @_2maxImpl in frame
197 store i64 %54, i64* %55, align 8
198 %56 = load i64, i64* %temp2, align 8
199 %frame15 = load i64*, i64** %locals, align 8
200 %57 = getelementptr i64, i64* %frame15, i64 5
201 store i64 %56, i64* %57, align 8
202 ; Allocate memory for function value for recursive call:
203 %58 = tail call i64* @allocate(i64 16)
204 %59 = ptrtoint i64* %58 to i64
205 store i64 %59, i64* %temp17, align 8
206 %60 = bitcast i64** %locals to i64***
207 %frame1812345 = load i64**, i64*** %60, align 8
208 %61 = load i64*, i64** %frame1812345, align 8
209 %62 = load i64, i64* %61, align 8
210 ; Store function pointer at index 0 in function value:
211 store i64 ptrtoint (i64 (i64, i64)* @_1maxImpl to i64), i64*
    %58, align 8
212 %63 = getelementptr i64, i64* %58, i64 1
213 ; Store shared frame at index 1 in function value:
214 store i64 %62, i64* %63, align 8
215 %frame19 = load i64*, i64** %locals, align 8
216 %64 = getelementptr i64, i64* %frame19, i64 3
217 %65 = load i64, i64* %64, align 8
218 store i64 %65, i64* %temp21, align 8
219 ; Get function frame:
220 %66 = load i64*, i64** %tempptr16, align 8
221 %67 = bitcast i64* %66 to i64 (i64, i64)**
222 %68 = load i64 (i64, i64)*, i64 (i64, i64)** %67, align 8
223 %69 = getelementptr i64, i64* %66, i64 1
224 %70 = load i64, i64* %69, align 8
225 ; Call @_1maxImpl:
226 %71 = tail call i64 %68(i64 %70, i64 %65)
227 ; Returns function value, which we store on the stack:
228 store i64 %71, i64* %temp23, align 8
229 %72 = getelementptr i64, i64* %frame19, i64 4
230 %73 = load i64, i64* %72, align 8
231 store i64 %73, i64* %temp26, align 8
232 %.cast = inttoptr i64 %71 to i64*
233 %74 = inttoptr i64 %71 to i64 (i64, i64)**
234 %75 = load i64 (i64, i64)*, i64 (i64, i64)** %74, align 8

```

```

235 %76 = getelementptr i64, i64* %.cast, i64 1
236 %77 = load i64, i64* %76, align 8
237 ; Call @_2maxImpl:
238 %78 = tail call i64 @75(i64 %77, i64 %73)
239 ; Store returned function value on stack:
240 store i64 %78, i64* %temp28, align 8
241 %79 = getelementptr i64, i64* %frame19, i64 5
242 %80 = load i64, i64* %79, align 8
243 store i64 %80, i64* %temp31, align 8
244 %.cast6 = inttoptr i64 %78 to i64*
245 %81 = inttoptr i64 %78 to i64 (i64, i64)**
246 %82 = load i64 (i64, i64)*, i64 (i64, i64)** %81, align 8
247 %83 = getelementptr i64, i64* %.cast6, i64 1
248 %84 = load i64, i64* %83, align 8
249 ; Recursively call @_3maxImpl:
250 %85 = tail call i64 @82(i64 %84, i64 %80)
251 ret i64 %85
252
253 fun_match_fail:
254 ; Should never get here.
255 ; This function call will crash the program with the
256 ; "pattern match failed" error message:
257 tail call void @matchError()
258 unreachable
259 }

```

The LLVM Assembly generated for the `max` function consists of 2 functions. I will not show the code generated for `max`, since it does not contain anything interesting that we have not seen in the previous LLVM assembly listings.

8 Testing Code Generation

In this section I will show some of test programs which have been used to test correctness of the generated code. The test programs I have included in this section serve as good test programs as well as programs demonstrating the compiler.

Note that I have added the `print` function to the compiler. Since the compiler does not support `strings` the `print` function has the type signature `'a -> unit`. It takes any argument and prints the arguments integer value, for a pointer that is the address and for a `bool` that is the value 0 or 1. Later in the report I will describe how the compiler distinguishes between pointers and the scalar types `int` and `bool`.

8.1 Scope Rules

In the first program we test some scope rules of the language:

```

1 val shadow: int = 1
2
3 fun first(f1:int)(f2:int): unit =
4   let
5     val f: int = 20
6     fun second(s1:int)(s2:int): unit =

```



```

7         if s1 = f1 andalso s2 = f2
8         then print(f)
9         else print(~1)
10
11     val f: int = 30
12     fun third(t:int): unit =
13         if t = f
14         then print(shadow)
15         else print(~1)
16
17     val shadow: int = 2
18 in
19     second f1 f2;
20     third f;
21     print shadow
22 end
23
24 val () = (first 3 4;
25           print shadow)

```

Nested function `second` is called with `f=20`, `s1=f1` and `s2=f2`, thus it prints 20. Function `third` is called with `f` redeclared such that `f=30` and `t=f`, thus it prints 1 (the value of global `shadow`). Value of `shadow` is later shadowed by function `first` such that printing `shadow` in the end of `first` outputs 2. After `first` has returned the global `shadow` is printed, that is 1. Thus the output of executing that program is:

```

1 20
2 1
3 2
4 1

```

8.2 Common Functions

In the next test program we will implement some common functions:

```

1 (* List we will use as input for functions *)
2 val input: int list = [1,2,3,4,5]
3
4 (* Define function composition *)
5 infixr 3 o;
6 fun (f: 'b -> 'c) o (g: 'a -> 'b) : 'a -> 'c = fn (x:'a) => f (g x)
7
8 fun even(x:int) : bool = x mod 2 = 0
9 val odd: int -> bool = not o even
10
11 fun map(_: 'a -> 'b)([] : 'a list): 'b list = []
12   | map(f: 'a -> 'b)(x::xs : 'a list): 'b list = f x :: map f xs
13
14 fun filter(f: 'a -> bool)([] : 'a list): 'a list = []
15   | filter(f: 'a -> bool)(x::xs : 'a list): 'a list = let
16       val tail: 'a list = filter f xs
17   in
18       if f x then x::tail else tail
19   end
20
21 (* Map function partially applied with negation op~ *)

```

```

22 val negmap: int list -> int list = map op~
23
24 (* Results of using the functions *)
25 val duped: int list = map (fn x:int => x*2) input
26 val neged: int list = negmap input
27 val evens: int list = filter even input
28 val odds : int list = filter odd input
29
30 (* Print results *)
31 fun printList([] : 'a list): unit = ()
32   | printList(x::xs : 'a list): unit = (print x; printList xs)
33 val () = (printList duped;
34           printList neged;
35           printList evens;
36           printList odds)

```

The test program speaks for itself. The output of compiling and afterwards running the program is:

```

1 2
2 4
3 6
4 8
5 10
6 -1
7 -2
8 -3
9 -4
10 -5
11 2
12 4
13 1
14 3
15 5

```

The first 5 numbers are doubles of the input numbers, the next 5 numbers are negations of the input numbers, the next 2 numbers are the even input numbers and the last 3 numbers are the odd input numbers.

8.3 Binary Search Tree

The next program demonstrates the implementation of a simple binary search tree:

```

1 (* Ordering for comparing elements in a set *)
2 datatype ordering = LT | EQ | GT
3
4 (* A node in a set is a leaf or a node with 2 children *)
5 datatype 'a setnode = Leaf
6                     | Node of 'a * 'a setnode * 'a setnode
7
8 (* Set contains comparison function and root node. *)
9 datatype 'a set = Set of ('a*'a -> ordering) * 'a setnode
10
11 (* Get an empty set by supplying a comparison function. *)
12 fun empty(cmp: 'a*'a -> ordering): 'a set = Set (cmp, Leaf)
13

```

```

14 (* insert element in set. *)
15 fun insert(x:'a)(Set(cmp, node): 'a set): 'a set =
16   let
17     infix cmp
18     fun ins(Leaf: 'a setnode): 'a setnode =
19       Node (x, Leaf, Leaf)
20     | ins(Node(y, left, right): 'a setnode): 'a setnode =
21       (fn LT      => Node(y, ins left, right)
22        | EQ      => Node(y, left, right)
23        | (_ : ordering) => Node(y, left, ins right)) (x cmp y)
24   in
25     Set(op cmp, ins node)
26   end
27
28 (* Convert list into set. *)
29 fun fromList(cmp: 'a*'a -> ordering)([] : 'a list): 'a set =
30   empty cmp
31 | fromList(cmp: 'a*'a -> ordering)(x::xs: 'a list): 'a set =
32   insert x (fromList cmp xs)
33
34 (* Apply function f to each element in set (in-order). *)
35 fun app(f: 'a -> 'b)(Set(_, node): 'a set): unit =
36   let
37     fun inord(Leaf : 'a setnode): unit = ()
38     | inord(Node (x, left, right): 'a setnode): unit = (
39       inord left;
40       f x;
41       inord right)
42   in
43     inord node
44   end
45
46 (* An integer set for testing *)
47 val set: int set =
48   let
49     fun cmp(x:int,y:int): ordering = if x < y then LT
50                                     else if x = y then EQ
51                                     else GT
52   in
53     fromList cmp [5, 3, 1, 6, ~3, 1]
54   end
55
56 (* Print values of an in-order traversal. *)
57 val () = app print set

```

Note that the compiler does not support the **case-of** expression. Thus we use a function value like a **case-of** expression in the **ins** function nested in the **insert** function. When executed, the program correctly outputs:

```

1  -3
2  1
3  3
4  5
5  6

```

8.4 Foldl with Ref

The next program demonstrates using `ref` to implement `foldl`:

```
1 fun null([]: 'a list): bool = true
2   | null(_ : 'a list): bool = false
3
4 fun hd(x::_ : 'a list): 'a = x
5 fun tl(_::xs: 'a list): 'a list = xs
6
7 fun foldl(f: 'a * 'b -> 'b)(i: 'b)(xs: 'a list): 'b =
8   let
9     val acc : 'b ref = ref i
10    val rest: 'a list ref = ref xs
11  in
12    while not (null (!rest)) do
13      let
14        val head: 'a = hd (!rest)
15      in
16        rest := tl(!rest);
17        acc := f(head, !acc)
18      end;
19      !acc
20    end
21  end
22 val sum: int list -> int = foldl op+ 0
23
24 val () = print (sum [1,2,3])
```

Executing that program yields the output:

```
1 6
```

8.5 Foldl for Array

And the final test program I will demonstrate in this section shows how we can implement `foldl` for arrays:

```
1 fun foldl(f: 'a * 'b -> 'b)(i: 'b)(a: 'a array): 'b =
2   let
3     val n : int      = Array.length a
4     val idx: int ref = ref 0
5     val acc: 'b ref = ref i
6   in
7     while !idx < n do (
8       acc := f(Array.get(a, !idx), !acc);
9       idx := !idx + 1
10    );
11    !acc
12  end
13
14 val sum: int array -> int = foldl op+ 0
15
16 val a: int array = Array.array (3, 0)
17 val () = (Array.update (a, 0, 1);
18           Array.update (a, 1, 2);
19           Array.update (a, 2, 3))
```

```
20 | val () = print (sum a)
```

When executing the program it outputs the expected:

```
1 | 6
```

9 Garbage Collection

When generating LLVM intermediate representation supporting GC there is a number of things we need to do. Initially we must select a GC strategy. In reality, we have two options when selecting a GC strategy. We can select the **shadow-stack** strategy or we can implement our own custom GC strategy. My SML compiler supports both options.

A GC strategy defines how we detect GC roots at runtime. The **shadow-stack** strategy maintains a linked list with stack roots. Whenever a function is called the stack roots from that function are inserted in the linked list, and when the function returns; the stack roots are removed.

The advantages of using the **shadow-stack** GC strategy are:

- the **shadow-stack** GC strategy is build into LLVM; making it a simple solution
- this strategy is cross platform; we don't need to write complex platform dependent code for keeping track of GC roots

The disadvantages of this GC strategy are:

- since code for removing stack roots from the linked list is inserted right before returning from functions; LLVM is not able to perform tail call optimization
- there is a minor runtime overhead of maintaining the linked list with stack roots

When implementing a custom GC strategy, what we do is to write code for emitting a stack map statically into the resulting executable. The stack map should contain information about the size of each function's stack frame and where stack roots are located in the stack frame. Whenever a GC is triggered we use this information to locate stack roots.

The advantage of the custom GC strategy is:

- it is fast; LLVM is not injecting any additional code into user functions to help keeping track of GC roots

The disadvantages are:

- it is more complex to implement than the **shadow-stack** strategy
- we need to write platform dependent code for locating stack roots when a GC is triggered

When generating LLVM IR supporting GC we let LLVM know which stack variables are GC roots.

When the **shadow-stack** strategy is selected; LLVM uses this information to maintain the linked list with stack roots.

When the custom strategy is selected; LLVM uses that information to decide where roots are located on the stack. In turn, we use this information from LLVM when emitting a stack map into the resulting executable.

When generating code supporting GC there are a number of pitfalls we have to consider.

- On function entry, we typically need to initialize stack roots with null. Otherwise, if a GC is triggered before initializing some stack root; the stack root will contain an uninitialized random value.
- Before a GC may be triggered we need to make sure pointers are written to memory. If we keep some pointer in a callee save register or some random memory location, then the GC might be unable to find that pointer and mistakenly reclaim its memory.
- When a pointer value gets out of scope before the function returns; we want to assign it with null such that the GC can reclaim its memory.

Consider the following expression:

```
1 f(g(), h())
```

Say `g()` returns some pointer and that the call to `h()` triggers a GC. If we have not written the temporary result from `g()` to memory, then it might erroneously be reclaimed.

Thus, when generating code supporting GC we cannot simply put intermediate results in registers and spill them to random memory locations. Before a GC can happen we must make sure that local function values and temporary results are written to memory locations where the GC will look, e.g. stack roots.

In the case of a GC which moves objects around, like a copying collector: After GC might have been triggered we must make sure to read pointers from memory. Since pointer values might change after a GC, it is not safe to keep a pointer in a callee save register and use it afterwards.

10 GC Strategy Implementation

10.1 Shadow-stack Strategy

When using the **shadow-stack** GC strategy; LLVM creates a global variable called `llvm_gc_root_chain`. This variable is the head of the linked list containing stack roots. We use this variable to locate and, if needed, reassign stack roots.

Each node in the `llvm_gc_root_chain` linked list is a stack frame, where the most recently called function's stack frame is the head of the list. The first word of each stack frame is a pointer to the next stack frame in the linked list, the second word in each stack frame is a pointer to a stack frame descriptor which contains the number of GC roots on the stack frame and other metadata. If there are N roots then the next N words in the stack frame are GC roots. In this way we can use the head of the linked list `llvm_gc_root_chain` to crawl the stack and locate GC roots from the stack. The following C code illustrates how to do so, where `llvm_gc_root_chain` is an `int64_t` pointer:

```

1 // Get head of list:
2 int64_t *stack_frame = llvm_gc_root_chain;
3 while (stack_frame != 0) {
4     // Pointer to stack frame descriptor is located at index 1:
5     int32_t *descriptor = (int32_t *)stack_frame[1];
6     // Get number of GC roots in stack frame from descriptor:
7     int32_t number_roots = descriptor[0];
8     for (int32_t i = 0; i < number_roots; i++)
9         // Use GC root at index i+2 from the stack frame for something:
10        process_root(stack_frame[i+2]);
11    // Get pointer to previous function's stack frame from index 0:
12    stack_frame = (int64_t *)stack_frame[0];
13 }

```

At entry of each function LLVM generates code to maintain the linked list. When `stackptr` is a `int64_t` pointer; pointing to the top of the x86-64 stack, then we can illustrate the LLVM generated code using C code as follows:

```

1 stackptr[0] = (int64_t)llvm_gc_root_chain;
2 stackptr[1] = stack_frame_descriptor;
3 llvm_gc_root_chain = stackptr;

```

Where `stack_frame_descriptor` is a pointer to the stack frame descriptor of the function.

Right before the return instructions in each function LLVM generates code to remove the function's stack frame (the head of the linked list) from the linked list as follows:

```

1 llvm_gc_root_chain = (int64_t*)stackptr[0];

```

10.2 Custom Strategy

When using the custom GC strategy we need to do a bit more work to find stack roots. The stack map which is generated by my compiler is put in a section called `ssmkgcmap`. This section is an array containing stack information for each function in the program. The array looks as follows (using gas assembly syntax since it is not possible to define this data structure with standard C syntax):

```

1 ssmkgcmap_entry_1: # Entry of 1st function in program.
2     .short L        # Stack frame size.

```

```

3  .short M          # Number of stack roots.
4  .short I_1        # Stack index of 1st stack root.
5  .short I_2        # Stack index of 2nd stack root.
6  ...
7  .short I_M        # Stack index of Mth stack root.
8  ssmlgcmap_entry_2: # Entry for 2nd function in program.
9  ...
10 ...
11 ssmlgcmap_entry_N: # Entry for Nth function in program.
12 ...

```

In order to make use of the `ssmlgcmap` array I emit information to another section called `ssmlgclookup`. This section contains 2 constant variables. The first variable called `ssml_gclookup_size` contains the total number of functions in the program (excluding runtime library functions). The other variable called `ssml_gclookup_map` is an array of function addresses followed by the address of the corresponding stack information in the `ssmlgcmap` array. Using C syntax, the `ssmlgclookup` section looks like this:

```

1  struct lookup_entry {
2      void *fun_addr;
3      void *ssmlgc_entry;
4  }
5
6  const uint64_t ssml_gclookup_size = N; // Number of functions
7  const struct lookup_entry ssml_gclookup_map[N+1] = {
8      {&user_fun_1, &ssmlgcmap_entry_1},
9      {&user_fun_2, &ssmlgcmap_entry_2},
10     ...
11     {&user_fun_N, &ssmlgcmap_entry_N},
12     {&code_end, &0}
13 }

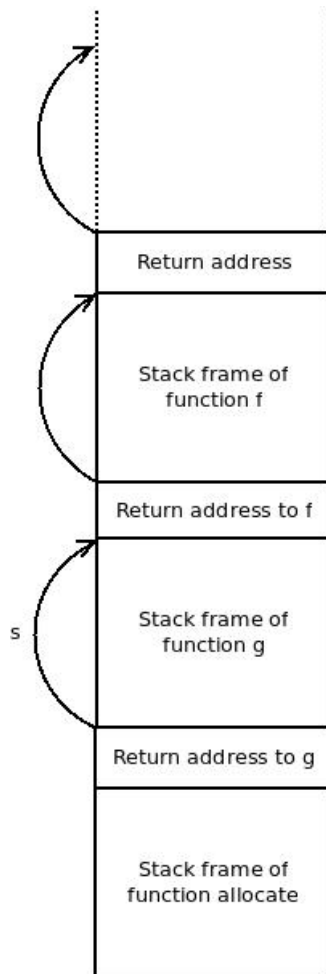
```

While executing, the user program allocates memory using a function called `allocate`. Say that the `allocate` function has just been called and the GC algorithm has decided that it is time to trigger a GC.

Now we need to locate stack roots. The way we do this is by using the information in the `ssmlgcmap` and `ssmlgclookup` sections to implement a stack crawling algorithm and locate stack roots while crawling the stack.

The first thing we do is to get the return address of the `allocate` function. This address `x` will be an address to some executable code in a user defined function `g` which called `allocate`. Next we can make a binary search in the `ssml_gclookup_map` array to locate which function the address belongs to. When we find `ssml_gclookup_map[i].fun_addr <= x` and `ssml_gclookup_map[i+1].fun_addr > x` then the entry to stack information of the function which called `allocate` is at index `i`, (`ssml_gclookup_map[i].ssmlgc_entry` corresponding to address of `ssmlgcmap_entry_i`).

Having address of the `ssmlgcmap_entry_i` entry of the function `g` which called `allocate`; We can use it to locate the stack roots in `g`'s stack frame. By further using `ssmlgcmap_entry_i` we can get the size `s` of `g`'s stack frame. With `s` we can locate the return address to the function `f` which called `g`. Consider the following figure of the call stack:



We know that the return address to function f is located right before g 's stack frame and that the stack frame of f is located before the return address.

Continuing by performing a binary search in the `ssml_gclookup_map` array searching for `ssml_gclookup_map[j].ssmlgc_entry` corresponding to the address of f 's `ssmlgcmap_entry_j` entry. Using `ssmlgcmap_entry_j` we find the stack roots on f 's stack frame, and the return address of f , etc.

Continuing using this procedure we can crawl the stack and locate all stack roots at the cost of doing a binary search in the `ssml_gclookup_map` for each stack frame.

By using this method for locating stack roots we avoid storing the addresses of the `ssmlgcmap_entry_k` entries on the stack.

The problem with doing a binary search in the `ssml_gclookup_map` array is: In big programs the array is unlikely to be in cache so it might cause some cache misses while doing the binary search.

The problem with storing address of the `ssmlgcmap_entry_*` entries on the stack is that LLVM does not provide a simple platform independent way to

specify at which index we want local variables to be on the stack frame. Without implementing complex platform dependent code, only given a stack frame, we cannot know where address of the `ssmlogcmap_entry_k` entry is stored. However, if implementing this, then given stack frame pointer `f` we know that address of the `ssmlogcmap_entry_k` entry is located at `f[c]` for some constant index `c`. This solution would be almost free, since address of `ssmlogcmap_entry_k` is constant; it has the cost of storing a constant on the stack frame for each function called.

There are also other workarounds for solving this problem. One of them includes passing the `ssmlogcmap_entry_k` entry as argument when calling a function such that the callee have the `ssmlogcmap_entry_k` entry of the caller.

I settled with the binary search solution since it works and it is simple, and it is unlikely to impose any significant overhead.

11 Mark and Sweep Implementation

I have implemented a mark and sweep GC algorithm for the compiler. The algorithm is specifically implemented for the x86-64 linux platforms.

11.1 Buddy Memory Allocator

The memory allocator used by the mark and sweep algorithm is the most complex part, it is a Buddy memory allocation variant. It has 17 free lists containing memory blocks. The free list with the smallest memory blocks (`free_lists[0]`) contains memory blocks of 2^5 bytes, the next free list (`free_lists[1]`) contains blocks of size 2^6 bytes, etc. The free list with the largest memory blocks is `free_lists[16]`, it contains memory blocks of size 2^{21} bytes, i.e. 2MB blocks. In general `free_lists[i]` contains memory blocks of size 2^{i+5} . The array of free lists can be declared using C code as follows:

```
1 int64_t *free_lists[17];
```

A memory block from `free_lists[i]` can "only" serve memory allocations of size $\leq 2^{i+5} - 8$ since one 8 byte machine word is used for holding the size of the memory block.

While a memory block is in a free list it contains a pointer to the previous and next memory blocks in the free list. When the memory block is located in the beginning of the free list then it does not have a previous memory block (represented with NULL) and when the memory block is located in the end of the free list then it does not have a next memory block (also represented with NULL). Say memory block beginning at address `m` is located in some free list, then `m[1]` is a pointer to the previous memory block in the free list (possibly NULL), `m[2]` is a pointer to the next memory block in the free list (possibly NULL) and `m[0]` contains the size of the memory block.

When a memory allocation of n bytes is requested, the allocator will locate the free list at index i containing the smallest memory blocks that can serve a n byte memory allocation. If this free list has an available memory block, then that memory block is removed from the free list and a pointer to that memory is returned. Assuming there is a free memory block in `free_lists[i]` then the following C code illustrates how we can remove that memory block from the free list with the `pop` function:

```

1  int64_t *pop(size_t i) {
2      // Get memory block from beginning of free list at index i:
3      int64_t *m = free_lists[i];
4      // Remove the memory block from the free list:
5      free_list[i] = (int64_t*)m[2];
6      // Now head of the free list is next memory block, possibly NULL.
7      if (free_list[i]) {
8          // If there was a next memory block:
9          // We remove its previous pointer:
10         free_list[i][1] = 0;
11         // And we remove the next pointer from m:
12         m[2] = 0;
13     }
14     return m;
15 }

```

If `free_lists[i]` does not have a free memory block; the algorithm will look for a free memory block in the free lists containing bigger memory blocks, starting from `free_lists[i+1]`. If a free memory block in `free_lists[i+j]` is found, then that memory block is removed from the free list and split into two evenly sized blocks. One of those blocks are put in `free_lists[i+j-1]`, and if $j = 1$ then a pointer to the other memory block is returned, otherwise the other memory block is further split into two evenly sized blocks, etc. Once the memory block which was removed from `free_lists[i+j]` has been split j times; all free lists i to $i+j-1$ will contain an extra memory block and a pointer to a memory block with size fitting `free_lists[i]` will be returned. Using C code, we can illustrate it with the following function, assuming `free_lists[i+j]` contains a free memory block:

```

1  int64_t *split(size_t i, size_t j) {
2      // Remove memory block from free_lists[i+j]:
3      int64_t *first = pop(i+j);
4      if (j == 0)
5          // Done. Return memory block from free_lists[i].
6          return first;
7      // Otherwise we need to split memory block first.
8      // Get size of the memory block:
9      int64_t old_size = first[0];
10     // When splitting the memory block we half the size of the
11     // memory block. Compute size of the two new memory blocks:
12     int64_t new_size = old_size / 2;
13     // Save size in first memory block:
14     first[0] = new_size;
15     // The size of the memory blocks are in bytes, but we index
16     // an int64_t array. Compute index of second new memory block:
17     int64_t second_index = new_size/sizeof(int64_t);
18     // Get pointer to second memory block:
19     int64_t *second = &first[second_index];
20     // Store size of second memory block:

```

```

21     second[0] = new_size;
22     // Insert second in free_lists[i+j-1]:
23     push(second, i+j-1);
24     // Insert first in free_lists[i+j-1]:
25     push(first, i+j-1);
26     // Continue until first has size fitting free_lists[i].
27     return split(i, j-1);
28 }

```

Where function `push(int64_t *b, size_t i)` inserts memory block `b` in the beginning of `free_lists[i]`, `push` is defined as follows:

```

1 void push(int64_t *b, size_t i) {
2     // Block b becomes previous pointer of current head.
3     free_lists[i][1] = (int64_t)b;
4     // Current head becomes b's next pointer.
5     b[2] = (int64_t)free_lists[i];
6     // Block b becomes the head.
7     free_lists[i] = (int64_t)b;
8 }

```

Note that the functions described here are not efficient implementations, but they are easier to understand than optimized implementations.

If `free_lists[i]` does not have a free memory block and none of the free lists containing larger memory blocks have a free block, then a new 2MB memory block is allocated from the operating system. This memory block is put in `free_lists[16]`. Now the algorithm can split this memory block as needed and return the properly sized memory block, for example using:

```

1 // Use malloc or some other function to get memory from OS:
2 int64_t *b = malloc(pow(2, 21));
3 push(b, 16);
4 return split(i, 16-i);

```

Initially all free lists are empty, thus all memory blocks with size $\leq 2\text{MB}$ originate from some 2MB memory block.

Note that the address returned to the user is not going to be the beginning of the memory block. When memory block pointed to by `int64_t* b`, which will be used to serve the users memory allocation, has been removed from its free then pointer `b+1` is returned to the user. This makes sure the user does not override (he doesn't even know about) the size word in the beginning of the block.

Memory allocations bigger than 2MB are handled by asking the operating system for the memory. The allocated memory block is put in a linked list containing all the memory blocks of size bigger than 2MB and a pointer to the memory block is returned.

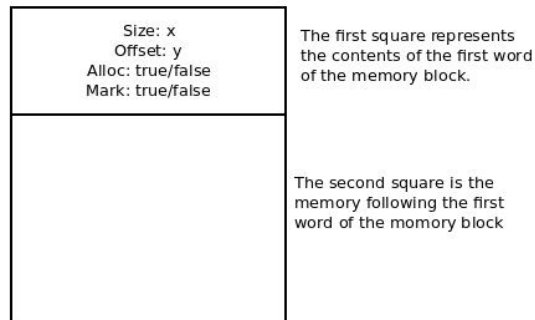
Say that we have a pointer to a memory block `b`. In the previous examples we assumed that we could get the size by indexing `b[0]`. In reality `b[0]` contains its size, and its byte offset to the 2MB block it originates from, and an allocated bit indicating whether the block is free or allocated, and a mark bit which is

used by the marking phase of the mark and sweep algorithm. Bit at index 63 is the mark bit, index 62 is the allocated bit, bit indices 23-61 are used for containing the offset into the 2MB block it originates from and bit indices 0-22 are used for containing the block size.

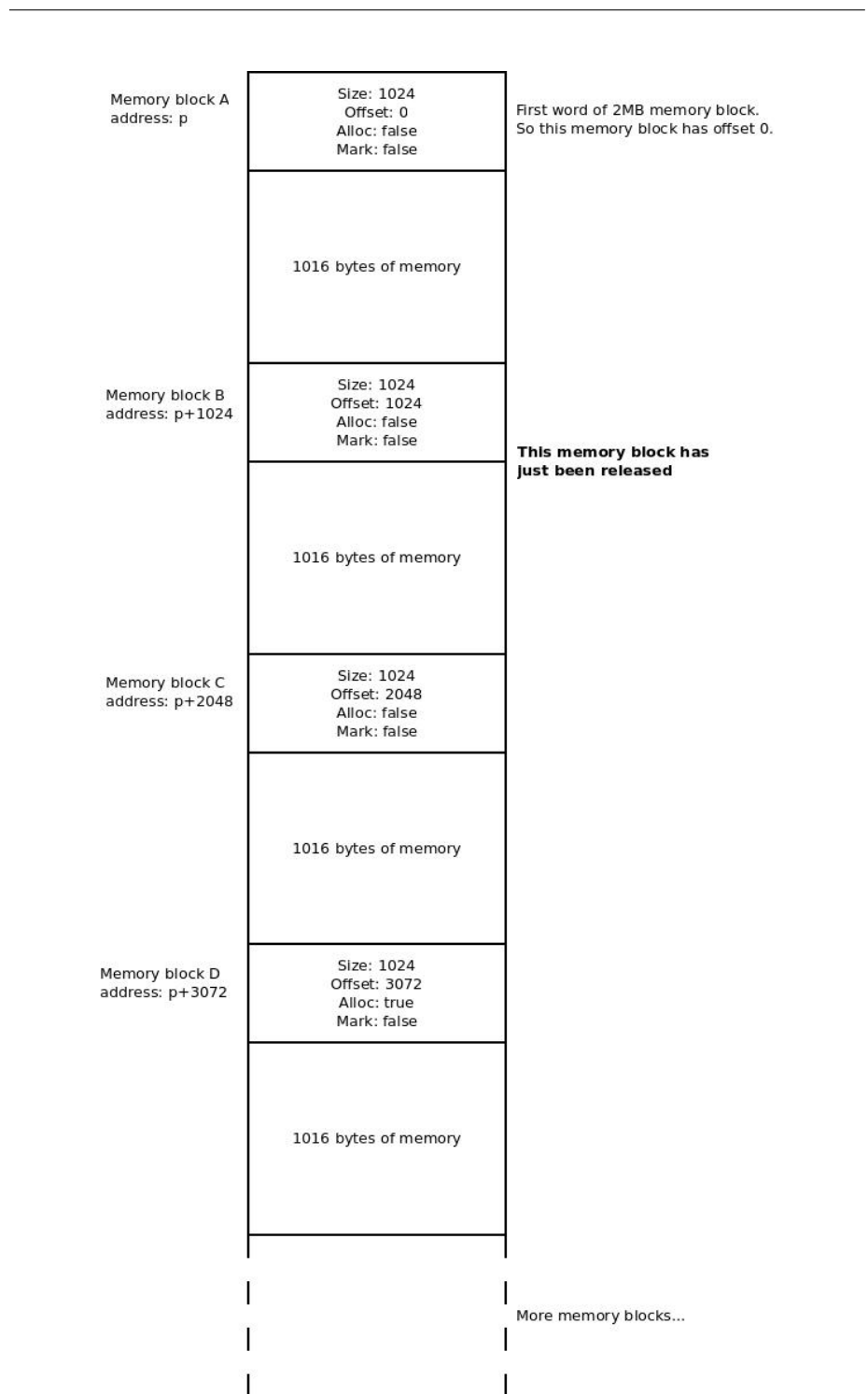
When a block is in a free list the mark and allocated bits are cleared but the offset into the 2MB block it originates from might not be 0 / cleared. Thus when we need to get the size of a memory block we really need to mask out the first 23 bits.

When a memory block starting at address z is released; first the memory block is filled with zeros (except from the first word of the memory block containing its size, etc.). Afterwards its size x in bytes and offset y into the 2MB memory block it originates from is used to find the memory block which it was split from (its buddy). If $(y \bmod (2 \cdot x) = 0)$ then its buddy is located at address $z + x$ otherwise its buddy is located at address $z - x$. If the buddy's size is also x and its allocated bit is cleared, then the buddy is removed/popped from its free list and the memory block and its buddy are merged into a memory block of size $2 \cdot x$. This process continues until the memory block cannot be merged with its buddy, in which case it is inserted into its free list. The following illustrations show how the merging of memory blocks works:

A memory block is represented like this:

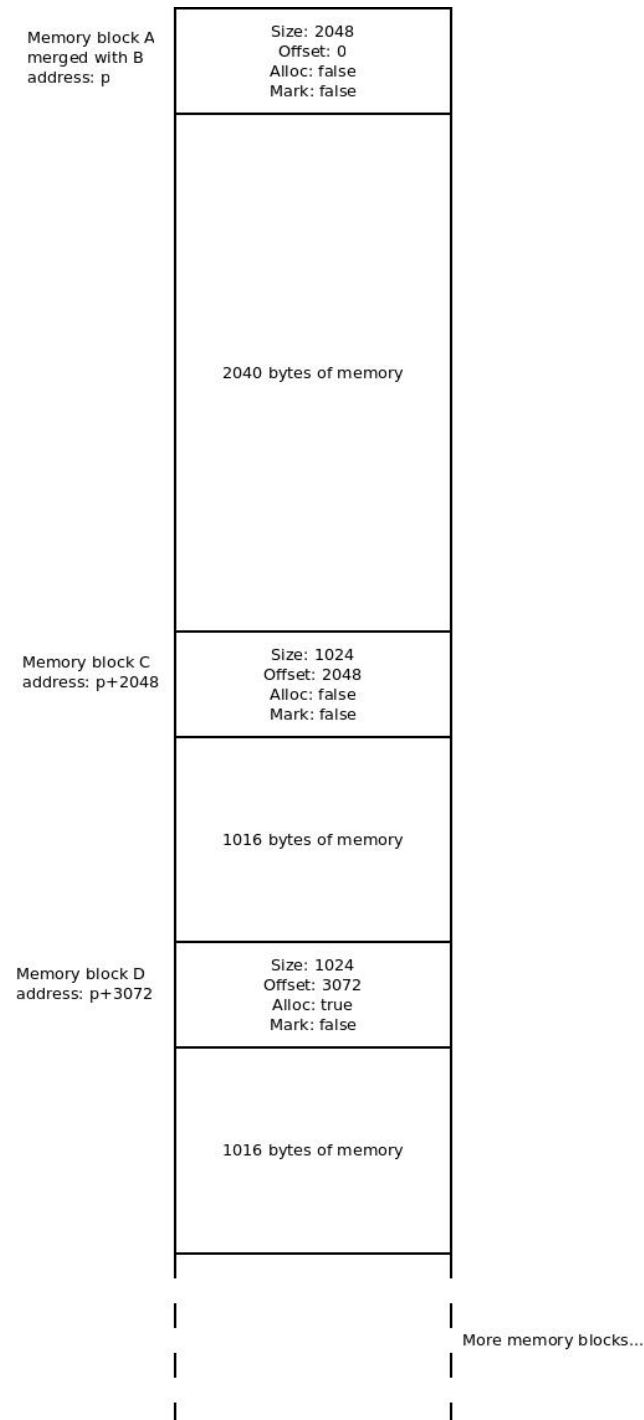


Say that a GC has been triggered and that memory block B in the following illustration has been released:



Using the formula to find the buddy of memory block B, we have $z = p + 1024$, $x = 1024$, $y = 1024$. Thus: $(y \bmod (2 \cdot x) = 1024 \bmod 2048 = 1024)$. Implying that the buddy of memory block B is located at address $(z - x = (p + 1024) - 1024 = p)$.

Since the size of the buddy (memory block A) is also 1024, and since the buddy's allocated bit is cleared; memory block A and memory block B are merged:



While merging memory block A with B memory block A is removed from its free list.

Using the formula to find the buddy of memory block A merged with B, we have $z = p$, $x = 2048$, $y = 0$. Thus: $(y \bmod (2 \cdot x) = 0 \bmod 4096 = 0)$. Implying that the buddy of memory block A merged with B is located at address $(z + x = p + 2048)$.

Although memory block C at address $p + 2048$ is not allocated, it does not have the same size as memory block A merged with B. Thus, the blocks are not merged and memory block A merged with B is inserted in the free list containing memory blocks of size 2048.

As we can see, memory block D is allocated since its allocated bit is not cleared. If memory block D is now released (not illustrated) then $z = p + 3072$, $x = 1024$, $y = 3072$, and we have: $(y \bmod (2 \cdot x) = 3072 \bmod 2048 = 1024)$ implying that memory block C located at address $(z - x = (p + 3072) - 1024 = p + 2048)$ is the buddy of memory block D. Now, C and D will merge, and afterwards further merge with memory block A merged with B...

Refer to appendix `runtime/MarkSweepAlloc.cpp` function `release` if you would like to see some C++ code taking care of merging released memory blocks.

When the allocation function `allocate` is called, where n bytes of memory is requested, and the amount of allocated memory plus n is bigger than a certain threshold then a garbage collection is triggered. If, after the garbage collection, the amount of allocated memory plus n is still bigger than the threshold, then the threshold is increased to allow allocation of more memory. If the size of the threshold becomes bigger than 2GB then the program will crash with an "out of memory" error message. In the beginning of the program the threshold is 128MB.

11.2 Marking and Sweeping

When a GC is triggered the mark and sweep algorithm will go through the stack roots one at a time; marking the roots and marking the allocated memory blocks reachable from them by setting the mark bit. In order to avoid stack overflow, for example when marking a large list, the marking function is not recursive. Instead the marking function uses a dynamically growing list from the C++ standard library to keep track of state. The C++ implementation of the marking function `markall` is understandable, also for people not knowing the C++ language:

```
1 // Function markall marks object Root and all objects
2 // reachable from Root. If Root is not dynamically allocated
3 // or if it is not a pointer, then the function does nothing.
4 static void markall(int64_t *Root) {
5     // If Root is not a dynamically allocated pointer or
6     // if Root is already marked, then return:
7     if (!istraced(Root) || ismarked(Root))
8         return;
9
10    // Linked list containing reachable unmarked objects:
11    std::forward_list<int64_t *> Objects;
```

```

12 // Initialliy Root is unmarked:
13 Objects.push_front(Root);
14
15 do {
16     // Get Obj from front of list:
17     int64_t *Obj = Objects.front();
18     // Remove front of list:
19     Objects.pop_front();
20
21     // If allocated bit is not set, then it is an error:
22     assert(isAllocated(Obj));
23
24     // Mark Obj, and return the word size of Obj:
25     int64_t Size = mark(Obj);
26     // For each word in Obj:
27     for (int64_t I = 0; I < Size; ++I) {
28         // Get Child from index I of Obj:
29         int64_t *Child = (int64_t *)Obj[I];
30         // If Child is a dynamically allocated object and
31         // it has not been marked, then put it in list with
32         // unmarked reachable objects:
33         if (istraced(Child) && !ismarked(Child))
34             Objects.push_front(Child);
35     }
36     // While there are reachable unmarked objects:
37 } while (!Objects.empty());
38 }

```

An optimization possibility with this marking function is to spot arrays. If an object has type `t array` where `t` is a scalar type, then scanning through the array searching for reachable objects is not necessary.

For the mark and sweep collector, to distinguish between pointers and scalar values the Code Generator makes sure all scalar values contain a 1 bit as first bit. Thus the integer value 0 is internally represented as 1, and the integer 1 is internally represented as 3, etc. By looking at the first bit of a word the marking phase can determine whether a word is pointer or scalar value. Using this; the collector will only mark reachable objects, and it will not accidentally mark a garbage object because an `int` happens to have a value equivalent to that object's address. This makes the mark and sweep collector precise.

If a precise GC algorithm is not a requirement, with a mark and sweep allocator, it is possible to treat any word aligned value as a pointer to a dynamically allocated object. If the value does not point to anything dynamically allocated then it is treated as a scalar type, otherwise it is treated as a pointer to the dynamically allocated object it points to. This might give some false positives, but since most objects are small it is unlikely to be a problem.

After the marking phase the mark and sweep algorithm will go through all allocated memory blocks: The memory blocks which have not been marked are put back in free lists (using the `release` function) and the memory blocks which have been marked will get unmarked by clearing the mark bit, this is referred to as the sweeping phase. In order to do so, a linked list containing all memory blocks of size 2MB is maintained. Given a 2MB memory block `m` we can loop over all memory blocks contained within `m` as follows:

```

1  int64_t *b = m;
2  // End address of 2MB block.
3  uint64_t end = (uint64_t)b + pow(2, 21);
4  // While memory block b is within the bounds of m:
5  while ((uint64_t)b < end) {
6      // Use memory block b:
7      ...
8      // Get size of b masking out mark, allocated and offset bits.
9      uint64_t size = b[0] & 0x7FFFFFFF;
10     // Get next memory block:
11     b = (uint64_t)b + size;
12 }

```

Using this method for looping over all memory blocks contained within a 2MB memory block and using the linked list with all the 2MB memory blocks, then the sweeping phase can find all memory blocks with size less than or equal 2MB. And since the memory blocks of sizes bigger than 2MB are also linked together in a linked list; the sweeping phase can also find all of those.

After a GC has been triggered, the threshold defining how much memory the user must allocate before the next GC is triggered might increase. In order to determine whether this should happen, I use the following expression:

$$(t - u) \cdot 4 < 3 \cdot 128 \cdot 2^{20}$$

where u is the amount of reachable memory and t is the threshold, such that $(t - u)$ is the amount of memory the user can allocate before next GC is triggered. If $(t - u) \cdot 4$ is less than 384MB then the threshold t is increased with 128MB.

Refer to appendix `runtime/MarkSweepAlloc.cpp` if you are interested in the low level C++ mark and sweep implementation.

12 Copying Collector Implementation

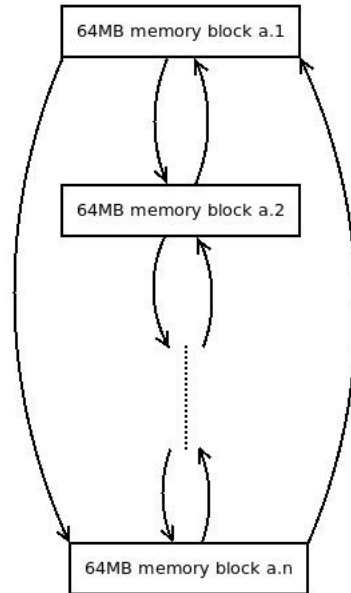
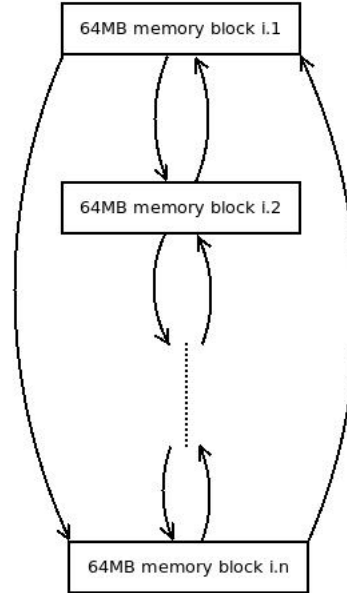
Like with the mark and sweep collector the copying collector is specific to the x86-64 architecture.

12.1 Bump-a-pointer Allocator

One huge benefit of a copying GC compared to a mark and sweep GC is that we do not need to write a complex memory allocator to avoid fragmentation.

My copying collector uses a bump-a-pointer memory allocator variant.

At all time there is an active memory space and an inactive memory space of the same size. Initially the active and inactive memory spaces are represented by one 64MB memory block each. If at any point more memory is necessary then two additional 64MB memory blocks are allocated, one linked with the active memory space, the other linked with the inactive memory space. We can illustrate the active and inactive memory spaces as circular doubly linked lists as follows:

Active memory space:**Inactive memory space:**

Thus for each 64MB memory block, 2 words are reserved, one for a pointer to the previous memory block and the other for a pointer to the next memory block.

The user allocates memory with the `allocate` function. When the user requests a memory chunk of n bytes then $n + 8$ bytes are allocated from the active space, the extra 8 bytes are allocated to store the size of the memory chunk. The previous and next pointers are at index 0 and 1 respectively, in the 64MB memory blocks. We can use the following C code to illustrate how to handle memory allocations with the `active_allocate` function. This might look like a lot of code, but it mostly consists of comments to make it easier to understand:

```

1 // Pointer to current active memory block:
2 int64_t *active_block;
3 // Pointer to head of active memory block linked list:
4 int64_t *active_head;
5 // Pointer to head of inactive memory block linked list:
6 int64_t *inactive_head;
7 // Index of next memory allocation from active_block starts
8 // from active_index:
9 uint64_t active_index;
10 // Allocate parameter n bytes of memory for the user.
11 // This function assumes that n is a multiple of sizeof(int64_t)
12 // and that n <= pow(2,16) - 8:
13 int64_t *active_allocate(uint64_t n) {
14     // Allocate an additional word for storing the size of the chunk.
15     uint64_t total = n + sizeof(int64_t);
16     if (active_offset_in_bounds(total))
17         // Then the active block can serve the allocation:
18         return active_allocate_curr_block(total);
19     // Otherwise we get the necessary memory by trying the
20     // next memory block pointed to by active_block, or
21     // by triggering a GC, or by asking the OS:
22     return active_allocate_next_block(total);

```

```

23 }
24 // Returns true if there is free memory in active_block to
25 // serve a memory allocation of parameter total bytes,
26 // otherwise returns false:
27 bool active_offset_in_bounds(uint64_t total) {
28     // Get address &active_block[active_index] after an allocation
29     // of total bytes.
30     uint64_t end = (uint64_t)(active_block + active_index) + total;
31     // Get address of active_block:
32     uint64_t start = (uint64_t)active_block;
33     // Return whether an allocation of total bytes is inside
34     // the bounds of active_block:
35     return (end-start) <= pow(2,26);
36 }
37 // Allocate parameter total bytes from active_block:
38 int64_t *active_allocate_curr_block(uint64_t total) {
39     // Get beginning of memory chunk which will be allocated:
40     int64_t *chunk = active_block;
41     // Store index where next allocation will begin:
42     active_index += total / sizeof(int64_t);
43     // Store size of chunk in first word:
44     chunk[0] = total;
45     // Return pointer to chunk the user can use (this does not
46     // include the first word of the chunk used for the size):
47     return chunk + 1;
48 }
49 // Try to allocate parameter total bytes of memory from the
50 // memory block after active_block. If that is not possible
51 // get the necessary memory by triggering a GC. If that does
52 // not work also, then ask OS for more memory.
53 int64_t *active_allocate_next_block(uint64_t total) {
54     // Make the active block be the next block pointed
55     // to by the current active block:
56     active_block = (int64_t*)active_block[1];
57     if (active_block != active_head)
58         // Then the "new" active block is unused.
59         // Allocate the memory from that block:
60         return active_allocate_curr_block(total);
61     // There are no unused memory blocks left and the last
62     // memory block was unable to serve the memory allocation.
63     // Now we trigger a GC to see if we can free memory
64     // such that we can serve the allocation without
65     // asking the OS for more memory:
66     triggerGC();
67     // After the garbage collection we check if we can
68     // get a memory block from the new active memory block
69     // (which before the GC was an inactive memory block):
70     if (active_offset_in_bounds(total))
71         return active_allocate_curr_block(total);
72     // Otherwise, there might be a free memory
73     // block after the new active block:
74     active_block = (int64_t*)active_block[1];
75     if (active_block != active_head) {
76         // After the GC there is an unused memory block.
77         // Allocate the memory from that block:
78         return active_allocate_curr_block(total);
79     }
80     // We were unable to free enough memory to serve
81     // the memory allocation. We ask the OS for 2 new
82     // memory blocks:
83     active_block = malloc(pow(2,26));
84     int64_t *new_inactive = malloc(pow(2,26));

```

```

85 // Insert the two new memory blocks in the lists:
86 insert(active_head, active_block);
87 insert(inactive_head, new_inactive);
88 // Now we got memory to serve the allocation:
89 return active_allocate_curr_block(total);
90 }
91 // Function to insert memory block parameter new_block
92 // before memory block parameter list_node in the doubly
93 // linked list of memory blocks:
94 void insert(int64_t *list_node, int64_t *new_block) {
95     // Get list_node's prev pointer:
96     int64_t *prev = (int64_t*)list_node[0];
97     // new_block's prev pointer will become list_node's prev pointer:
98     new_block[0] = (int64_t)prev;
99     // new_block's prev pointer's next pointer will become
100     // new_block instead of list_node:
101     prev[1] = (int64_t)new_block;
102     // new_block's next pointer will become list_node:
103     new_block[1] = (int64_t)list_node;
104     // list_node's prev pointer will become new_block:
105     list_node[0] = (int64_t)new_block;
106 }
107 // Function called when a GC is necessary. This function
108 // will copy reachable memory from active space to inactive space
109 // and afterwards swap which space is active and inactive:
110 void triggerGC() {
111     // For a C++ implementation, see appendix runtime/CopyingGC.cpp
112     // A lot of the workings of this function is also described later
113 }

```

Using the `active_allocate` function the copying collector will serve all memory allocations smaller than 16kB.

If the user wants to allocate a block of memory bigger than or equal to 16kB then the operating system is directly asked for the memory block. The memory block is put in a linked list with memory blocks of size bigger than or equal to 16kB, and a pointer to the memory block is given to the user.

12.2 Copying, Marking and Sweeping

The `triggerGC()` function is partly used for copying reachable objects from the active space to the inactive space. In order to do so, it iterates through the GC roots and copies the GC roots and the objects directly and indirectly reachable from the GC roots from active space to inactive space. Once all the reachable objects have been copied from active space to inactive space; active and inactive space is swapped such that the memory space which was active is now inactive and vice versa.

A detail when copying from active memory space to inactive memory space is: Even though there is an equal amount of memory blocks allocated for active and inactive space, if the allocated memory is packed differently in the inactive memory blocks; it is possible that there is not enough inactive memory. If this event occurs, then the algorithm will continue by allocating an additional 64MB memory block for both active and inactive space.

Note that reachable objects with size bigger than or equal to 16kB are not

copied; they are only being marked. Now, since they are located in the linked list with memory blocks of size bigger than or equal to 16kB. This linked list can be used by the sweep phase in the end of the `triggerGC()` function, where the sweep phase frees the unreachable "big" objects to the OS and unmarks the reachable "big" objects.

Right before returning, the `triggerGC()` function will fill what was the active memory space (which is now inactive memory space) with zeros. If the code generator makes sure to completely initialize objects right after allocating memory for them, before a garbage collection may be triggered, then it is not strictly necessary to fill the unused memory with zeros. For security reasons garbage collectors often zero memory after it has been released anyway.

Similar to the marking phase in the mark and sweep collector, to avoid stack overflows it is important that the copying phase inside the `triggerGC()` function is not recursive. This is achieved in a similar fashion, by using a linked list to keep track of objects which have not yet been copied. It is a little more involved than with the marking function from the mark and sweep algorithm, since the linked list also contains a destination address where the new address of copied objects is assigned to. Although the following C++ code might be complex; I include the implementation of the function used by `triggerGC` for copying GC roots (`memoryCopy`) here, since the way I avoid recursion using a linked list is quite interesting:

```
1 // Function memoryCopy will forward (copy) parameter Object and
2 // the objects directly and indirectly reachable from Object
3 // from inactive to active memory space.
4 // The function returns the new address of Object after
5 // it has been copied to inactive memory space.
6 static int64_t *memoryCopy(int64_t *Object) {
7     // Ret is destination memory location for Object.
8     int64_t Ret;
9
10    // List containing destination addresses and objects
11    // which will be copied and assigned to its destination
12    // memory address afterwards:
13    std::forward_list<std::pair<int64_t *, int64_t *>> MemObjStack;
14    // Push Object and its destination address:
15    MemObjStack.emplace_front(&Ret, Object);
16
17    do {
18        // Pop destination address and associated object from stack:
19        auto P = MemObjStack.front();
20        MemObjStack.pop_front();
21        int64_t *DestMem = P.first;
22        int64_t *Obj = P.second;
23
24        if (!istraced(Obj)) {
25            // Obj is not a heap pointer, so should not be forwarded.
26            // Just store its value to its new memory field.
27            *DestMem = (int64_t)Obj;
28            continue;
29        }
30
31        // Get the word size of Obj:
32        int64_t Size = getObjectSize(Obj);
33        // Word size of Obj must never be 0:
34        assert(Size);
```

```

35
36     if (!isSmallMemory(Obj)) {
37         // If Obj is a big memory object >= 16kB,
38         // then the object is an array. We do not forward the object,
39         // and we only forward its elements if they are non-scalar
40         // and if they have not yet been forwarded.
41
42         // Index 0 of arrays contain the array's size.
43         // Index 1 of arrays contain the first element.
44         // If first element is non-scalar, then so are the rest.
45         if (!isForwarded(Obj) && istraced((int64_t *)Obj[1])) {
46             // Array contains non-scalar elements which have not yet
47             // been forwarded. Add them to list so they can be copied:
48             for (int64_t I = 1; I < Size - 1; ++I) {
49                 int64_t *Child = (int64_t *)Obj[I];
50                 MemObjStack.emplace_front(&Obj[I], Child);
51             }
52         }
53         // Store Obj to its destination memory location:
54         *DestMem = (int64_t)Obj;
55         // Mark Obj as forwarded:
56         markForwarded(Obj);
57         continue;
58     }
59
60     if (isForwarded(Obj)) {
61         // Obj has already been forwarded.
62         // Store its forwarded address to its destination address:
63         *DestMem = *Obj;
64         continue;
65     }
66
67     // If we get here; Obj is a heap allocated pointer and has not
68     // yet been forwarded. So we forward it now and store its
69     // forwarded address to its destination address.
70
71     // Allocate memory from inactive heap. This is the new inactive
72     // memory address of Obj. This will be destination memory for
73     // fields from Obj:
74     int64_t *Forward = memoryForwardAlloc(Size * sizeof(int64_t));
75
76     // Put fields from Obj in list. Together with
77     // destination address:
78     for (int64_t I = 0; I < Size - 1; ++I) {
79         int64_t *Child = (int64_t *)Obj[I];
80         MemObjStack.emplace_front(&Forward[I], Child);
81     }
82
83     // Insert forwarded address of Obj at index 0 of Obj
84     // and mark Obj forwarded:
85     *Obj = (int64_t)Forward;
86     markForwarded(Obj);
87
88     // Store forwarded memory address of Obj to destination
89     // address:
90     *DestMem = (int64_t)Forward;
91     // While there are reachable objects not yet copied...
92     } while (!MemObjStack.empty());
93
94     // Return inactive memory address of copy of Object:
95     return (int64_t *)Ret;
96 }

```

Also for the copying collector to be precise; the Code Generator makes sure all scalar values contain a 1 bit as first bit.

After a GC has been triggered, if there is less than 32MB of free memory in the "new" active memory space, then the active and inactive memory spaces are both increased with a new 64MB memory block from the OS.

If at any point the program needs to use more than 2GB of memory then the program will terminate and print the "out of memory" error message.

If no memory allocations bigger than 16kB has been made, then half of the memory is used for active space and the other half is used for inactive space. In this case, effectively the copying collector has half as much memory available than the mark and sweep collector.

Please refer to appendix `runtime/CopyingGC.cpp` for the rest of the C++ implementation of the copying collector.

13 Generational Collector

The generational garbage collector I have implemented is also specific to the x86-64 platform.

13.1 Bump-a-pointer Allocator

My generational collector uses the fast bump-a-pointer memory allocation strategy.

The user program allocates memory chunks smaller than 16kB from a 4MB memory block, which we will refer to as the first generation.

Once all the memory from the first generation is exhausted a GC is triggered. After the GC, the reachable objects from the first generation will be copied to memory which is allocated using the same allocation strategy as the copying collector. We will refer to this memory as the second generation.

So my generational collector has two generations: first generation, which is a 4MB memory block; and second generation, which consists of an active and an inactive memory space, like the copying collector.

If the user wants to allocate a memory chunk bigger than or equal to 16kB then the memory is allocated the same way as the copying collector handles memory allocations of this size.

13.2 Minor and Major Garbage Collections

The generational garbage collector has two kinds of GCs: Minor and Major.

A minor collection happens when the first generation is exhausted and the active memory space from the second generation has more than 4MB of free memory.

The minor collection consists of copying all the reachable objects from the first generation to the active memory space of the second generation.

A major collection happens when the first generation is exhausted and there is less than 4MB of free memory in the active memory space of the second generation. The major collection is similar to the collection which is performed by the copying collector. It copies all reachable objects from the first generation and from the active memory space of the second generation to the inactive memory space of the second generation.

The thing that makes a generational collector attractive is that when "only" a minor collection is triggered, then it is not always necessary to scan through the whole stack while searching for GC roots. The pointers to the first generation will be located in the most recent part of the stack. Thus, by using a remembered set, we do not have to scan the whole stack when a minor GC is triggered. We only scan the last part which contains stack frames of functions which have been executing after the last GC was triggered.

In order to know which stack frames belong to functions which have been executing since the last GC was triggered, each stack frame contains a boolean field. In the entry of each function this boolean field is initialized to 0.

While a minor collection is scanning the stack looking for GC roots, it will set the boolean field of each scanned stack frame to 1. If it finds a stack frame where the boolean field is 1 (before setting it), then this is the last stack frame which will be scanned for GC roots. In this way a minor collection can avoid scanning the whole stack for GC roots.

The remembered set is used when a pointer to a first generation object is stored in a second generation object. The obvious cases where this may happen are when updating an array or when assigning a reference. The non-obvious case when this can happen is when a function is called, a GC is triggered while executing that function, and the function returns a pointer to a first generation object, which is afterwards stored in a local variable. Since memory for all local variables is dynamically allocated (as discussed in section 7.1), and since a GC was triggered, the local variable will be a pointer to a second generation object. When storing the returned pointer in the local variable, we are storing a first generation pointer in a second generation object.

So the remembered set is used for remembering which second generation objects contains pointers to first generation objects. Of course, when a minor collection is triggered we must also copy the first generation objects which are pointed to by the second generation objects in the remembered set.

After any GC, the remembered set is cleared. I use a `DenseSet` from the LLVM library as remembered set in my generational collector implementation.

Instead of having a boolean field in each stack frame, an alternative to find out which stack frames belong to functions which have been executing since the last GC is to use a GC strategy which dynamically allocates stack frames.

This GC strategy (which is not implemented by the compiler) allocates all stack frames on function entry from the first generation. Like with the shadow-stack GC strategy, this strategy maintains a linked list of all the stack frames. When

a minor collection is triggered; while scanning the linked list with stack frames for GC roots, once a stack frame from the second generation is reached, then that is the last stack frame which is scanned.

The source code for the generational collector is located in appendix `runtime/GenerationalGC.cpp`.

The generational collector is also distinguishing between scalar types and pointers by letting scalar have a 1 bit as first bit.

The drawback of having 1 as first bit of all scalar types is that applications doing many mathematical computations will not be quite as fast as if scalar types was represented by their true value. This is so since it will be necessary to convert back and fourth between having and not having 1 as first bit. In most applications it is unlikely to result in any noticeable difference.

An alternative to letting scalar types have 1 as first bit is to include a pointer to an object descriptor in each dynamically allocated object. The object descriptor contains information necessary to locate objects directly reachable from that object. One problem with that approach is that a function with type `'a -> t` does not know whether the argument passed to it is a scalar type or a pointer without further modifications.

13.3 Full LLVM Assembly Listing

This section contains the LLVM assembly code which is generated by passing the compiler the following program, while using the custom GC strategy and enabling the generational collector:

```
1 fun head(x::xs: 'a list): 'a = x
```

The generated LLVM assembly is as follows:

```
1 target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
2 target triple = "x86_64-unknown-linux-gnu"
3
4 @FirstGenBegin = external externally_initialized constant i64
5 @FirstGenEnd = external externally_initialized constant i64
6
7 ; Function Attrs: nounwind
8 declare void @llvm.gcroot(i8**, i8*) #0 gc "ssml"
9
10 declare i64* @allocate(i64) gc "ssml"
11
12 ; Function Attrs: noreturn
13 declare void @matchError() #1
14
15 declare void @rememberset_insert(i64*)
16
17 define i64 @entry() gc "ssml" {
18 entrylabel:
19   %tempptr = alloca i64*, align 8
20   %0 = bitcast i64** %tempptr to i8**
21   tail call void @llvm.gcroot(i8** %0, i8* null)
22   %temp = bitcast i64** %tempptr to i64*
23   %tempptr1 = alloca i64*, align 8
```

```

24 %1 = bitcast i64** %tempptr1 to i8**
25 tail call void @llvm.gcroot(i8** %1, i8* null)
26 %temp2 = bitcast i64** %tempptr1 to i64*
27 %locals = alloca i64*, align 8
28 store volatile i64 1, i64* %temp, align 8
29 %2 = bitcast i64** %locals to i8**
30 tail call void @llvm.gcroot(i8** %2, i8* null)
31 %3 = tail call i64* @allocate(i64 16)
32 store volatile i64* %3, i64** %locals, align 8
33 %frame = load volatile i64*, i64** %locals, align 8
34 %4 = ptrtoint i64* %frame to i64
35 %5 = load i64, i64* @FirstGenEnd, align 8
36 %6 = icmp uge i64 %4, %5
37 %7 = load i64, i64* @FirstGenBegin, align 8
38 %8 = icmp ult i64 %4, %7
39 %or.cond.i = or i1 %6, %8
40 br i1 %or.cond.i, label %9, label %ssmlMemorybarrier.exit
41
42 ; <label>:9 ; preds = %entrylabel
43 tail call void @rememberset_insert(i64* %frame)
44 br label %ssmlMemorybarrier.exit
45
46 ssmlMemorybarrier.exit: ; preds = %entrylabel, %9
47 store volatile i64 0, i64* %frame, align 8
48 %10 = tail call i64* @allocate(i64 16)
49 %11 = ptrtoint i64* %10 to i64
50 store volatile i64 %11, i64* %temp2, align 8
51 %12 = load i64, i64* @FirstGenEnd, align 8
52 %13 = icmp uge i64 %11, %12
53 %14 = load i64, i64* @FirstGenBegin, align 8
54 %15 = icmp ult i64 %11, %14
55 %or.cond.i.1 = or i1 %13, %15
56 br i1 %or.cond.i.1, label %16, label %ssmlMemorybarrier.exit2
57
58 ; <label>:16 ; preds = %ssmlMemorybarrier.exit
59 tail call void @rememberset_insert(i64* %10)
60 br label %ssmlMemorybarrier.exit2
61
62 ssmlMemorybarrier.exit2: ; preds = %ssmlMemorybarrier.exit, %16
63 store volatile i64 ptrtoint (i64 (i64, i64)* @_lhead to i64),
64 i64* %10, align 8
65 %17 = getelementptr i64, i64* %10, i64 1
66 %frame3 = load volatile i64*, i64** %locals, align 8
67 %18 = ptrtoint i64* %frame3 to i64
68 %19 = and i64 %18, 1
69 %20 = icmp eq i64 %19, 0
70 br i1 %20, label %21, label %ssmlMemorybarrier.exit4
71
72 ; <label>:21 ; preds = %ssmlMemorybarrier.exit2
73 %22 = load i64, i64* @FirstGenEnd, align 8
74 %23 = icmp uge i64 %11, %22
75 %24 = load i64, i64* @FirstGenBegin, align 8
76 %25 = icmp ult i64 %11, %24
77 %or.cond.i.3 = or i1 %23, %25
78 br i1 %or.cond.i.3, label %26, label %ssmlMemorybarrier.exit4
79
80 ; <label>:26 ; preds = %21
81 tail call void @rememberset_insert(i64* %10)
82 br label %ssmlMemorybarrier.exit4
83
84 ssmlMemorybarrier.exit4: ; preds = %ssmlMemorybarrier.exit2,
85 %21, %26

```

```

84     store volatile i64 %18, i64* %17, align 8
85     %frame4 = load volatile i64*, i64** %locals, align 8
86     %27 = getelementptr i64, i64* %frame4, i64 1
87     %28 = and i64 %11, 1
88     %29 = icmp eq i64 %28, 0
89     br i1 %29, label %30, label %ssmlMemorybarrier.exit6
90
91 ; <label>:30                                ; preds = %ssmlMemorybarrier.exit4
92     %31 = ptrtoint i64* %frame4 to i64
93     %32 = load i64, i64* @FirstGenEnd, align 8
94     %33 = icmp uge i64 %31, %32
95     %34 = load i64, i64* @FirstGenBegin, align 8
96     %35 = icmp ult i64 %31, %34
97     %or.cond.i.5 = or i1 %33, %35
98     br i1 %or.cond.i.5, label %36, label %ssmlMemorybarrier.exit6
99
100 ; <label>:36                                ; preds = %30
101     tail call void @rememberset_insert(i64* %frame4)
102     br label %ssmlMemorybarrier.exit6
103
104 ssmlMemorybarrier.exit6:                    ; preds = %ssmlMemorybarrier.exit4,
105     %30, %36
106     store volatile i64 %11, i64* %27, align 8
107     ret i64 0
108 }
109
110 define i64 @_lhead(i64, i64) gc "ssml" {
111     entrylabel:
112     %tempptr = alloca i8*, align 8
113     tail call void @llvm.gcroot(i8** nonnull %tempptr, i8* null)
114     %tempptr1 = alloca i64*, align 8
115     %2 = bitcast i64** %tempptr1 to i8**
116     tail call void @llvm.gcroot(i8** %2, i8* null)
117     %temp2 = bitcast i64** %tempptr1 to i64*
118     %tempptr3 = alloca i64*, align 8
119     %3 = bitcast i64** %tempptr3 to i8**
120     tail call void @llvm.gcroot(i8** %3, i8* null)
121     %temp4 = bitcast i64** %tempptr3 to i64*
122     %locals = alloca i64*, align 8
123     store volatile i64 %0, i64* %temp2, align 8
124     store volatile i64 %1, i64* %temp4, align 8
125     %4 = bitcast i64** %locals to i8**
126     tail call void @llvm.gcroot(i8** %4, i8* null)
127     %5 = tail call i64* @allocate(i64 24)
128     store volatile i64* %5, i64** %locals, align 8
129     %6 = load volatile i64, i64* %temp2, align 8
130     %frame = load volatile i64*, i64** %locals, align 8
131     %7 = and i64 %6, 1
132     %8 = icmp eq i64 %7, 0
133     br i1 %8, label %9, label %ssmlMemorybarrier.exit
134
135 ; <label>:9                                ; preds = %entrylabel
136     %10 = ptrtoint i64* %frame to i64
137     %11 = load i64, i64* @FirstGenEnd, align 8
138     %12 = icmp uge i64 %10, %11
139     %13 = load i64, i64* @FirstGenBegin, align 8
140     %14 = icmp ult i64 %10, %13
141     %or.cond.i = or i1 %12, %14
142     br i1 %or.cond.i, label %15, label %ssmlMemorybarrier.exit
143
144 ; <label>:15                                ; preds = %9
145     tail call void @rememberset_insert(i64* %frame)

```

```

145     br label %ssmlMemorybarrier.exit
146
147 ssmlMemorybarrier.exit:    ; preds = %entrylabel, %9, %15
148     store volatile i64 %6, i64* %frame, align 8
149     %16 = load volatile i64, i64* %temp4, align 8
150     %17 = inttoptr i64 %16 to i64*
151     %18 = load volatile i64, i64* %17, align 8
152     %19 = icmp ult i64 %18, 2
153     br i1 %19, label %20, label %fun_match_fail
154
155 ; <label>:20                ; preds = %ssmlMemorybarrier.exit
156     %21 = getelementptr i64, i64* %17, i64 1
157     %22 = load volatile i64, i64* %21, align 8
158     %23 = inttoptr i64 %22 to i64*
159     %24 = load volatile i64, i64* %23, align 8
160     %frame5 = load volatile i64*, i64** %locals, align 8
161     %25 = getelementptr i64, i64* %frame5, i64 1
162     %26 = and i64 %24, 1
163     %27 = icmp eq i64 %26, 0
164     br i1 %27, label %28, label %ssmlMemorybarrier.exit2
165
166 ; <label>:28                ; preds = %20
167     %29 = ptrtoint i64* %frame5 to i64
168     %30 = load i64, i64* @FirstGenEnd, align 8
169     %31 = icmp uge i64 %29, %30
170     %32 = load i64, i64* @FirstGenBegin, align 8
171     %33 = icmp ult i64 %29, %32
172     %or.cond.i.1 = or i1 %31, %33
173     br i1 %or.cond.i.1, label %34, label %ssmlMemorybarrier.exit2
174
175 ; <label>:34                ; preds = %28
176     tail call void @rememberset_insert(i64* %frame5)
177     br label %ssmlMemorybarrier.exit2
178
179 ssmlMemorybarrier.exit2:    ; preds = %20, %28, %34
180     store volatile i64 %24, i64* %25, align 8
181     %35 = getelementptr i64, i64* %23, i64 1
182     %36 = load volatile i64, i64* %35, align 8
183     %frame6 = load volatile i64*, i64** %locals, align 8
184     %37 = getelementptr i64, i64* %frame6, i64 2
185     %38 = and i64 %36, 1
186     %39 = icmp eq i64 %38, 0
187     br i1 %39, label %40, label %ssmlMemorybarrier.exit4
188
189 ; <label>:40                ; preds = %ssmlMemorybarrier.exit2
190     %41 = ptrtoint i64* %frame6 to i64
191     %42 = load i64, i64* @FirstGenEnd, align 8
192     %43 = icmp uge i64 %41, %42
193     %44 = load i64, i64* @FirstGenBegin, align 8
194     %45 = icmp ult i64 %41, %44
195     %or.cond.i.3 = or i1 %43, %45
196     br i1 %or.cond.i.3, label %46, label %ssmlMemorybarrier.exit4
197
198 ; <label>:46                ; preds = %40
199     tail call void @rememberset_insert(i64* %frame6)
200     br label %ssmlMemorybarrier.exit4
201
202 ssmlMemorybarrier.exit4:    ; preds = %ssmlMemorybarrier.exit2,
    %40, %46
203     store volatile i64 %36, i64* %37, align 8
204     %frame7 = load volatile i64*, i64** %locals, align 8
205     %47 = getelementptr i64, i64* %frame7, i64 1

```

```

206   %48 = load volatile i64, i64* %47, align 8
207   ret i64 %48
208
209 fun_match_fail:                ; preds = %ssmlMemorybarrier.exit
210   tail call void @matchError()
211   unreachable
212 }
213
214 attributes #0 = { nounwind }
215 attributes #1 = { noreturn }

```

The `target datalayout` and `target triple` statements in the top of the listing are used by LLVM to perform optimizations for particular platforms.

The `@FirstGenBegin` and `@FirstGenEnd` global variables are used for implementing the memory barrier when a value is stored to dynamically allocated memory. `@FirstGenBegin` contains the address of the beginning of the first generation memory block, and `@FirstGenEnd` contains the address of the end of the first generation memory block.

If a pointer is stored in a dynamically allocated object which is not located within the bounds of `@FirstGenBegin` and `@FirstGenEnd` then the pointer is being stored in a second generation object and the address of that object is put in the remembered set, by calling the `@rememberset_insert` function.

When defining a function we specify which GC strategy we would like to use. In the previous listing the `"ssml"` GC strategy is selected, that is the name of the custom GC strategy which is described in section 10.2.

Notice the calls to the `@llvm.gcroot` function. These are not really function calls, but it is used to mark which stack allocated objects are GC roots.

14 Testing GC Algorithms

In this section I will present two of the programs which have been used to test the GC Algorithms. These test programs are designed to allocate a lot of memory to verify that the GC algorithms do not leak memory.

14.1 Small Allocations

The following program makes a lot of small memory allocations by allocating linked list nodes:

```

1 fun genlist(size: int): int list =
2 let
3   fun gen(0: int): int list = []
4   | gen(n: int): int list = size::gen(n-1)
5 in
6   gen size
7 end
8
9 fun allocList(listSize: int): unit =
10 let

```

```

11   val _: int list = genlist listSize
12 in
13   ()
14 end
15
16 fun allocLists(_: int)(0: int): int = 0
17   | allocLists(listSize: int)(numberLists: int): int =
18     let
19       val () = allocList listSize
20     in
21       1 + allocLists listSize (numberLists-1)
22     end
23
24 val _:int = allocLists 3000 3000
25 val _:int = allocLists 3000 3000
26 val _:int = allocLists 3000 3000
27 val _:int = allocLists 3000 3000
28 val _:int = allocLists 3000 3000
29 val _:int = allocLists 3000 3000

```

14.2 Small and Big Allocations

The following test program makes a combination of many small and large memory allocations:

```

1 fun getList(size:int): int list =
2 let
3   val n: int ref = ref 0
4   val ret: int list ref = ref []
5 in
6   while (!n < size) do (
7     ret := !n :: !ret;
8     n := !n + 1
9   );
10  !ret
11 end
12
13 fun alloc1(): unit =
14 let
15   val a: int array =
16     let
17       val x: int array = Array.array(300000, 0)
18       val y: int array = Array.array(300000, 0)
19       val z: int array = Array.array(300000, 0)
20     in
21       x
22     end
23
24   val a: int array =
25     let
26       val x: int array = Array.array(300000, 0)
27       val y: int array = Array.array(300000, 0)
28       val z: int array = Array.array(300000, 0)
29     in
30       x
31     end
32
33   val size: int = 2100077
34   val list: int list = getList size

```

```

35 in
36   ()
37 end
38
39 fun alloc2(): unit =
40 let
41   val size: int = 1300089
42   val list: int list = getList size
43   val list: int list = getList size
44   val list: int list = getList size
45 in
46   ()
47 end
48
49 fun run(): unit =
50 let
51   val () = alloc1 ()
52   val () = alloc1 ()
53   val () = alloc1 ()
54
55   val x: int array = Array.array(300000, 0)
56
57   val () = alloc2 ()
58   val () = alloc2 ()
59   val () = alloc2 ()
60   val () = alloc2 ()
61   val () = alloc2 ()
62
63   val () = alloc1 ()
64   val () = alloc1 ()
65   val () = alloc1 ()
66   val x: int array = Array.array(300000, 0)
67   val () = alloc1 ()
68   val () = alloc1 ()
69   val () = alloc1 ()
70
71   val () = alloc2 ()
72   val () = alloc2 ()
73   val () = alloc2 ()
74   val () = alloc2 ()
75   val () = alloc2 ()
76 in
77   ()
78 end
79
80 val () = (run(); run(); run(); run())

```

15 GC Algorithm Benchmarks

The next subsections contain three programs which have been used for testing and benchmarking the GC algorithms; discussed in sections 11, 12 and 13. In order to shorten the source code listings in the following subsections, I have included a common source code listing which is used by the programs in the following subsections. The common source code is:

```

1 fun id(x:'a):'a = x
2

```



```

3 fun head(x::_: 'a list): 'a = x
4
5 fun tail(_::x: 'a list): 'a list = x
6
7 fun null([]: 'a list): bool = true
8   | null(_ : 'a list): bool = false
9
10 fun foldl(_: 'a*'b -> 'b)(acc: 'b)([]: 'a list): 'b =
11   acc
12   | foldl(f: 'a*'b -> 'b)(acc: 'b)(x::xs: 'a list): 'b =
13     foldl f (f(x,acc)) xs
14
15 fun map(f: 'a -> 'b): 'a list -> 'b list =
16   foldl (fn (x: 'a, acc: 'b list) => f x :: acc) []
17
18 fun filter(f: 'a -> bool): 'a list -> 'a list =
19   foldl (fn (x: 'a, acc: 'a list) => if f x then x::acc else acc)
20     []
21
22 infixr 5 @
23 fun (xs: 'a list) @ (ys: 'a list): 'a list = let
24   fun cat([] : 'a list,
25         ys : 'a list,
26         cont: 'a list -> 'a list): 'a list =
27     cont ys
28     | cat(x::xs: 'a list,
29         ys : 'a list,
30         cont : 'a list -> 'a list): 'a list =
31         cat(xs, ys, fn acc: 'a list => cont (x::acc))
32   in
33     cat(xs, ys, id)
34   end
35
36 fun concat(xs: 'a list list): 'a list = foldl op@ [] xs
37
38 fun range(x:int)(y:int): int list = let
39   fun rng(acc: int list)(x:int)(y:int): int list =
40     if x > y
41     then acc
42     else rng (y::acc) x (y-1)
43   in
44     rng [] x y
45   end
46
47 val sum: int list -> int = foldl op+ 0

```

15.1 Recursive Subset Sum

This subsection contains the implementation of a naive recursive solution to the subset sum problem. The implementation is as follows:

```

1 fun subsetSum(_: int)([]: int list):bool =
2   false
3   | subsetSum(result: int)(set: int list):bool = let
4     fun recurse(h: int list, []: int list):bool =
5       sum h = result
6       | recurse(h: int list, t::ts: int list):bool =
7         if subsetSum result (h@ts)
8         then true

```

```

9         else recurse(t::h, ts)
10     in
11         recurse([], set)
12     end
13
14 val true = subsetSum 0 (range (~10) 10)

```

The program is designed to generate a lot of garbage while at the same time producing a useful result. The runtimes of the different GC algorithms using the custom "ssml" GC strategy are:

Mark and Sweep	~22.7 seconds
Copying	~4.5 seconds
Generational	~5.0 seconds

And by using the shadow-stack GC strategy, because of the runtime overhead of this strategy, we get the times:

Mark and Sweep	~23.0 seconds
Copying	~4.7 seconds
Generational	~5.1 seconds

As we can see from the results in both tables, the copying collector performs best on this test program. The generational collector is almost as fast. The mark and sweep collector is performing quite bad.

Why is the mark and sweep collector so slow compared to the copying and generational collectors?

Every time a GC is triggered, in the sweep phase, the mark and sweep collector scans through all allocated objects. The copying and generational collectors are only scanning objects which are reachable. Since the biggest part of all the allocated objects are not reachable when a GC is triggered, the mark and sweep collector pays a big price for scanning through all objects compared to only scanning reachable objects. As we will discuss later, the mark and sweep collector is also slow because of the complex buddy memory allocation strategy it uses.

When enabling GC statistics the compiler will output how much time is spent doing garbage collection. Using the custom "ssml" GC strategy, the total time spent on "only" doing garbage collection by the different algorithms are:

Mark and Sweep	~15.6 seconds
Copying	~0.83 seconds
Generational	~0.9 seconds

Hence, with the custom GC strategy, the fraction of the total runtime spent on doing garbage collection is:

Mark and Sweep	65%
Copying	18.67%
Generational	18%

The fraction of the runtime spent by the generational collector doing GC is smaller than the copying collector. The reason the copying collector is faster

executing this program is because of the runtime penalty the generational collector pays by doing memory barriers (making sure the remembered set up to date, but not necessarily inserting anything in the remembered set).

The total runtime spent on doing other things than garbage collection is:

Mark and Sweep	~7.1 seconds
Copying	~3.67 seconds
Generational	~4.1 seconds

The generational collector is slower than the copying collector here because of the memory barriers. The generational collector spends approximately $4.1 - 3.67 = 0.43$ seconds on memory barriers. That is $0.43/4.1 = 10.48\%$ of the amount of time spent on other things than GC is being spent on memory barriers.

Why is the mark and sweep collector even slower?

The time spent on doing other things than garbage collection includes memory allocations, and as we have discussed earlier, the mark and sweep buddy allocator is quite complex and time consuming. We can get an estimate of how much time is spent on allocating memory by the mark and sweep collector by subtracting the previous table's mark and sweep runtime from the copying collector runtime: $7.1 - 3.87 = 3.23$ seconds. Compare this to the 0.83 seconds which was the total time spent doing garbage collection by the copying collector.

The code which is generated by my compiler is not very optimized. It "only" relies on the default optimization options provided by LLVM. Although these optimizations are good, LLVM cannot optimize away calls to `allocate`. In reality, most of the calls to `allocate` are not necessary, and in a real compiler most of these calls should get optimized away by optimization passes. This will improve on the time spent doing memory allocations by the mark and sweep collector, and it will improve the time spent doing garbage collection by all the algorithms.

15.2 Combination Subset Sum

The naive solution to the subset sum problem I present in this subsection allocates a big `int list list` in the beginning of the program, this big list stays reachable until the program terminates:

```

1 fun combine(xs: 'a list): 'a list list = let
2   fun sublists(acc: 'a list list)(_: 'a list)([]: 'a list): 'a
      list list =
3     acc
4     | sublists(acc: 'a list list)(hs: 'a list)(t::ts: 'a list): 'a
      list list =
5       sublists ((hs@ts)::acc) (t::hs) ts
6 in
7   xs :: concat (map combine (sublists [] [] xs))
8 end
9
10 fun subsetSum(result:int)(set: int list): bool = let
11   val cs : int list list = combine set

```

```

12   val res: int list list = filter (fn x: int list => sum x =
13       result) cs
14 in
15   (fn []: int list list => false | _: int list list => true) res
16 end
17 val true = subsetSum 0 (range (~4) 4)

```

With the custom GC strategy the runtimes are:

Mark and Sweep	~14.22 seconds
Copying	~11.73 seconds
Generational	~4.67 seconds

The generational collector is significantly the fastest in programs like this one. It is designed to perform better than the copying collector when there is a mix of objects which are reachable for a long time and objects which are only reachable for a short time.

Because of the lack of tail call elimination when the shadow-stack strategy is used, when running this program while using the shadow-stack strategy, we get a stack overflow when calling the `concat` function. For this reason the shadow-stack strategy is a relatively uninteresting GC strategy option when implementing a compiler for a functional language with LLVM.

When using the custom strategy, the amount of time spent on doing only garbage collection is:

Mark and Sweep	~10.0 seconds
Copying	~10.0 seconds
Generational	~2.9 seconds

As we can see from the table, the mark and sweep collector spends approximately the same amount of time doing garbage collection as the copying collector.

The program spends most of its time allocating the big `int list list`, hence this list is reachable throughout the program, the copying collector spends a lot of time copying this list back and fourth between memory spaces.

Now, the price the mark and sweep collector pays for scanning through all allocated objects is not as high, since many of them are reachable anyway.

The fraction of total execution time spent on garbage collection is shown in the following table:

Mark and Sweep	70.3%
Copying	85.25%
Generational	62.10%

On this program where there is a lot of reachable objects, the fraction of the total execution time spent on doing garbage collection is high.

The times spent doing other things than garbage collection, is shown in the following table:

Mark and Sweep	~4.22 seconds
Copying	~1.73 seconds
Generational	~1.77 seconds

Again, in this program the mark and sweep algorithm pays for using a complex buddy memory allocator, compared to a fast bump-a-pointer allocator which is used by the generational and copying collector.

15.3 Iterative Subset Sum

The solution to the subset sum problem presented in this subsection makes use of the `while-do` loop and `ref`. It is very much implemented like the program in section 15.1, but using `while-op` loop as inner loop instead of the tail recursive `recurse` function.

```

1 fun subsetSum(_: int)([]: int list):bool =
2   false
3   | subsetSum(result: int)(set: int list):bool = let
4     val ret: bool ref = ref(sum set = result)
5     val hs : int list ref = ref []
6     val ts : int list ref = ref set
7   in
8     while not(!ret) andalso not(null(!ts)) do let
9       val t: int = head(!ts)
10      in
11        ts := tail(!ts);
12        if subsetSum result (!hs @ !ts)
13        then ret := true
14        else (hs := t :: !hs)
15      end;
16      !ret
17    end
18
19 val false = subsetSum 0 (range 1 10)

```

This test program is intended to test the penalty involved when the generational collector inserts elements in the remembered set. In section 15.1, there was no explicit use of operations which might cause the generational collector to insert elements in the remembered set. In the program above, we make heavy use of `ref` updates which are likely to cause insertions in the remembered set of the generational collector.

Using the custom "ssml" GC strategy the runtimes are:

Mark and Sweep	~20.5 seconds
Copying	~4.0 seconds
Generational	~4.6 seconds

A large fraction of objects in the program are short lived, hence the mark and sweep collector does not perform well compared to the other collectors. For the same reason, the generational collector is slower than the copying collector.

The amount of runtime and fraction of runtime spent on only doing garbage collection are listed in the following table:

Mark and Sweep	~13.45 seconds	65.6%
Copying	~0.76 seconds	19.0%
Generational	~0.8 seconds	17.4%

The most interesting table in this section is the following table which contains the amount of runtime spent on other things than garbage collection:

Mark and Sweep	~7.05 seconds
Copying	~3.24 seconds
Generational	~3.8 seconds

The generational collector spends $3.8 - 3.24 = 0.66$ seconds on keeping the remembered set up to date. Thus, $0.66/3.8 = 17.37\%$ of the amount of time spent on other things than GC is spent on keeping the remembered set up to date. In section 15.1 that number was 10.48%. The program in this section is making a lot of `ref` updates which the program in section 15.1 did not. Since the `ref` updates are likely to cause the generational collector to insert elements in the remembered set there is a time penalty. Using a faster remembered set implementation which is less `malloc` intensive is likely to significantly reduce this time penalty.

16 Conclusion

Although, the compiler is missing a lot of optimizations, for example by eliminating calls to `allocate`, we can still make some conclusions from the benchmark results we got.

Based on the benchmark results we have found benefits and drawbacks of the GC algorithms:

- A functional language like SML which is doing a lot of memory allocations is benefiting from having a fast bump-a-pointer allocator to avoid spending a lot of time allocating memory.
- The generational GC is faster than the copying GC when there are many long lived objects, but when there are only short lived objects, then the copying collector is slightly faster.
- There is a significant runtime penalty involved with memory barriers for the generational collector.
- The mark and sweep collector is not performing well when there are only short lived objects.
- Overall, whether there are many long lived or short lived objects, the generational GC is performing well.
- When there are many long lived objects then the fraction of time spent on doing garbage collection by the copying collectors (the generational and copying) is high, compared with when there are only short lived objects.

In order to get results we are more confident with, we should add some optimization passes to the compiler. We need to reduce the number of calls to `allocate` to reduce the time spent on doing garbage collection.

Afterwards we should implement more programs for benchmarking the algorithms, and then implement a number of other garbage collection algorithms for comparison. Garbage collector algorithms which will be interesting to implement in the future are:

- A generational collector which uses a mark and sweep algorithm to manage the second generation.
- A mark compact collector supporting bump-a-pointer allocation.
- Some of the concurrent collector variants.
- A reference counting and mark and sweep hybrid.

Although there is still work to do before we can draw final conclusions, we have got interesting benchmarking results. Not in the sense that the compiler generates blazingly fast executables, but in the sense that the benchmark results were the kind of results we were theoretically expecting based on the GC algorithm implementations.

17 Source Code Appendix

Listing 1: runtime/Print.c

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <inttypes.h>
4
5  void *print(int64_t i) {
6      if (i & 1)
7          printf("%" PRIu64 "\n", i >> 1);
8      else
9          printf("%p\n", (void *)i);
10     return 0;
11 }
12
13 void *framePrint(void *Frame, int64_t i) {
14     (void)Frame;
15     return print(i);
16 }
17
18 int64_t framePrintVal[2];
19
20 void initFramePrintVal() {
21     framePrintVal[0] = (int64_t)framePrint;
22     framePrintVal[1] = 0;
23 }
```

Listing 2: runtime/Error.c

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  void error(const char *Msg) {
5      fprintf(stderr, "runtime error: %s\n", Msg);
6      exit(1);
7  }
```

Listing 3: runtime/MatchError.c

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 void error(const char *);
5
6 void matchError() {
7     error("pattern match failed");
8 }

```

Listing 4: runtime/StdLib.c

```

1
2 #include <stdint.h>
3 #include <stdio.h>
4 #include <stdlib.h>
5
6 #ifdef SSML_GENERATIONAL_GC
7 void rememberset_insert(int64_t *DestBase);
8 #endif
9
10 int64_t *allocateFrame(int64_t N, void *Frame);
11 void error(const char *);
12
13 static inline int64_t mark(int64_t V) { return (V << 1) + 1; }
14
15 static inline int64_t unmark(int64_t V) { return V >> 1; }
16
17 int64_t arrayempty[1] = {0};
18
19 int64_t doarrayupdate(int64_t *Ary, int64_t N, int64_t V) {
20     N = unmark(N);
21     if (N < 0 || N >= unmark(Ary[0]))
22         error("array index out of bounds");
23     Ary[N + 1] = V;
24 #ifdef SSML_GENERATIONAL_GC
25     if (!(V & 1))
26         rememberset_insert(Ary);
27 #endif
28     return 0;
29 }
30
31 int64_t arrayupdate(int64_t Env, int64_t A[3]) {
32     return doarrayupdate((int64_t *)A[0], A[1], A[2]);
33 }
34 int64_t frameArrayupdate[2];
35
36 int64_t plus(int64_t Env, int64_t A[2]) {
37     return mark(unmark(A[0]) + unmark(A[1]));
38 }
39 int64_t framePlus[2];
40
41 int64_t minus(int64_t Env, int64_t A[2]) {
42     return mark(unmark(A[0]) - unmark(A[1]));
43 }
44 int64_t frameMinus[2];
45
46 int64_t multiply(int64_t Env, int64_t A[2]) {
47     return mark(unmark(A[0]) * unmark(A[1]));
48 }
49 int64_t frameMultiply[2];

```

```

50
51 int64_t division(int64_t Env, int64_t A[2]) {
52     return mark(unmark(A[0]) / unmark(A[1]));
53 }
54 int64_t frameDivision[2];
55
56 int64_t modulo(int64_t Env, int64_t A[2]) {
57     return mark(unmark(A[0]) % unmark(A[1]));
58 }
59 int64_t frameModulo[2];
60
61 int64_t refassign(int64_t Env, int64_t A[2]) {
62     int64_t *Dst = (int64_t *)A[0];
63     int64_t Src = A[1];
64     *Dst = Src;
65 #ifdef SSML_GENERATIONAL_GC
66     if (!(Src & 1))
67         rememberset_insert(Dst);
68 #endif
69     return 0;
70 }
71 int64_t frameRefassign[2];
72
73 static int64_t doarrayFrame(int64_t N, int64_t V, void *Frame) {
74     int64_t I;
75     int64_t *Ret;
76
77     int64_t Un = unmark(N);
78
79     if (Un < 1)
80         error("array size must be greater than 0");
81
82     Ret = allocateFrame(Un * sizeof(int64_t) + sizeof(int64_t),
83                         Frame);
84     Ret[0] = N;
85     for (I = 1; I < Un; ++I)
86         Ret[I] = V;
87     return (int64_t)Ret;
88 }
89
90 int64_t doarray(int64_t N, int64_t V) {
91     return doarrayFrame(N, V, __builtin_frame_address(0));
92 }
93
94 int64_t array(int64_t Env, int64_t A[2]) {
95     return doarrayFrame(A[0], A[1], __builtin_frame_address(0));
96 }
97 int64_t frameArray[2];
98
99 int64_t doarrayget(int64_t *Ary, int64_t N) {
100     N = unmark(N);
101     if (N < 0 || N >= unmark(Ary[0]))
102         error("array index out of bounds");
103     return Ary[1 + N];
104 }
105
106 int64_t arrayget(int64_t Env, int64_t A[2]) {
107     return doarrayget((int64_t *)A[0], A[1]);
108 }
109 int64_t frameArrayget[2];
110
111 int64_t negate(int64_t Env, int64_t Val) { return

```

```

        mark(-unmark(Val)); }
111 int64_t frameNegate[2];
112
113 int64_t not(int64_t Env, int64_t Val) {
114     return Val == mark(1) ? mark(0) : mark(1);
115 }
116 int64_t frameNot[2];
117
118 int64_t ref(int64_t Env, int64_t Val) {
119     int64_t *Ret = allocateFrame(sizeof(int64_t),
120     __builtin_frame_address(0));
121     *Ret = Val;
122     return (int64_t)Ret;
123 }
124 int64_t frameRef[2];
125
126 int64_t arraylength(int64_t Env, int64_t Val) { return *(int64_t
127 *)Val; }
128 int64_t frameArraylength[2];
129
130 int64_t deref(int64_t Env, int64_t Val) { return *(int64_t *)Val; }
131 int64_t frameDeref[2];
132
133 int64_t equals(int64_t Env, int64_t A[2]) {
134     return mark((int64_t)(A[0] == A[1] ? 1 : 0));
135 }
136 int64_t frameEquals[2];
137
138 int64_t notEquals(int64_t Env, int64_t A[2]) {
139     return mark((int64_t)(A[0] != A[1] ? 1 : 0));
140 }
141 int64_t frameNotEquals[2];
142
143 int64_t less(int64_t Env, int64_t A[2]) {
144     return mark((int64_t)(A[0] < A[1] ? 1 : 0));
145 }
146 int64_t frameLess[2];
147
148 int64_t greater(int64_t Env, int64_t A[2]) {
149     return mark((int64_t)(A[0] > A[1] ? 1 : 0));
150 }
151 int64_t frameGreater[2];
152
153 int64_t lessEquals(int64_t Env, int64_t A[2]) {
154     return mark((int64_t)(A[0] <= A[1] ? 1 : 0));
155 }
156 int64_t frameLessEquals[2];
157
158 int64_t greaterEquals(int64_t Env, int64_t A[2]) {
159     return mark((int64_t)(A[0] >= A[1] ? 1 : 0));
160 }
161 int64_t frameGreaterEquals[2];
162
163 void initStdLib() {
164     frameArrayupdate[0] = (int64_t)arrayupdate;
165     frameArrayupdate[1] = 0;
166
167     framePlus[0] = (int64_t)plus;
168     framePlus[1] = 0;
169     frameMinus[0] = (int64_t)minus;
170     frameMinus[1] = 0;
171     frameMultiply[0] = (int64_t)multiply;

```

```

170     frameMultiply[1] = 0;
171     frameDivision[0] = (int64_t)division;
172     frameDivision[1] = 0;
173     frameModulo[0] = (int64_t)modulo;
174     frameModulo[1] = 0;
175     frameRefassign[0] = (int64_t)refassign;
176     frameRefassign[1] = 0;
177     frameArray[0] = (int64_t)array;
178     frameArray[1] = 0;
179     frameArrayget[0] = (int64_t)arrayget;
180     frameArrayget[1] = 0;
181
182     frameNegate[0] = (int64_t)negate;
183     frameNegate[1] = 0;
184     frameNot[0] = (int64_t) not;
185     frameNot[1] = 0;
186     frameRef[0] = (int64_t)ref;
187     frameRef[1] = 0;
188     frameDeref[0] = (int64_t)deref;
189     frameDeref[1] = 0;
190     frameArraylength[0] = (int64_t)arraylength;
191     frameArraylength[1] = 0;
192
193     frameEquals[0] = (int64_t>equals;
194     frameEquals[1] = 0;
195     frameNotEquals[0] = (int64_t)notEquals;
196     frameNotEquals[1] = 0;
197     frameLess[0] = (int64_t)less;
198     frameLess[1] = 0;
199     frameGreater[0] = (int64_t)greater;
200     frameGreater[1] = 0;
201     frameLessEquals[0] = (int64_t)lessEquals;
202     frameLessEquals[1] = 0;
203     frameGreaterEquals[0] = (int64_t)greaterEquals;
204     frameGreaterEquals[1] = 0;
205 }
206
207 void patternCheck(int64_t Val) {
208     if (Val > 1000) {
209         fprintf(stderr, "VAL %p\n", (void *)Val);
210         error("VAL TOO BIG!");
211     }
212 }

```

Listing 5: runtime/GenerationalGC.cpp

```

1  #ifdef SSML_GENERATIONAL_GC
2
3  #include "GC.h"
4
5  #include "ssml/Common/Timer.h"
6  #include "llvm/ADT/DenseSet.h"
7
8  #include <strings.h>
9  #include <iostream>
10 #include <stdint.h>
11 #include <stdlib.h>
12 #include <sys/mman.h>
13 #include <forward_list>
14
15 #define MMAP_ALLOC

```

```

16
17 using namespace ssml;
18
19 extern "C" void error(const char *);
20
21 static uint64_t TriggerGCCount;
22 static uint64_t TotalGCTime;
23 static uint64_t BestGCTime = UINT64_MAX;
24 static uint64_t WhorstGCTime;
25 static uint64_t TotalGCRoots;
26 static uint64_t GCMemoryUsage;
27 static uint64_t FullGCCount;
28 static uint64_t FirstGenGCCount;
29
30 extern "C" void setFirstGenBegin(uint64_t);
31 extern "C" void setFirstGenEnd(uint64_t);
32
33 extern "C" void ssmlGCPrintStatistics() {
34     std::cout << "**** Generational GC Statistics ****\n";
35 #ifdef SSML_SHADOW_STACK_GC
36     std::cout << "Used Shadow Stack GC strategy\n";
37 #else
38     std::cout << "Used Binary Map GC strategy\n";
39 #endif
40     std::cout << "Total GCs: " << TriggerGCCount << '\n';
41     if (!TriggerGCCount)
42         return;
43
44     double Tot = (double)TotalGCTime / 1000;
45     double Avg = ((double)TotalGCTime / TriggerGCCount) / 1000;
46     double Whorst = (double)WhorstGCTime / 1000;
47     double Best = (double)BestGCTime / 1000;
48     double AvgRoots = (double)TotalGCRoots / TriggerGCCount;
49     uint64_t Avail = MAX_ALLOC_SIZE - GCMemoryUsage;
50
51     std::cout << "Total GC Time: " << Tot << "ms\n";
52     std::cout << "Average GC Time: " << Avg << "ms\n";
53     std::cout << "Whorst GC Time: " << Whorst << "ms\n";
54     std::cout << "Best GC Time: " << Best << "ms\n";
55     std::cout << "Average number Roots: " << AvgRoots << '\n';
56     std::cout << "Number full GCs: " << FullGCCount << '\n';
57     std::cout << "First generation GCs: " << FirstGenGCCount <<
58         '\n';
59     std::cout << "Total Memory: " << MAX_ALLOC_SIZE <<
60         "B\n";
61     std::cout << "GC Memory Usage: " << GCMemoryUsage << "B\n";
62     std::cout << "Total Free Memory: " << Avail << "B\n";
63     std::cout << "*****" << std::endl;
64 }
65
66 static llvm::DenseSet<int64_t *> RememberSet;
67
68 static int64_t getObjectSize(int64_t *Obj);
69 static bool isFirstGenPointer(int64_t *Obj);
70
71 extern "C" void rememberset_insert(int64_t *DestBase) {
72     assert(!isFirstGenPointer(DestBase));
73     assert(getObjectSize(DestBase));
74     RememberSet.insert(DestBase);
75 }
76
77 // When allocating more memory than this, use a mark sweep

```

```

        algorithm instead.
76 static const int64_t BigAllocateThreshold = 16384; // 2^14 bytes.
77 // Both heaps are increased by MemoryIncrease when more memory is
    needed.
78 static const int64_t MemoryIncrease = 2097152 * 32; // 2 * 32 MB.
79 // First generation memory size (2 * 2 MB):
80 #define FIRST_GEN_SIZE (2097152 * 2)
81
82 class HeapBlock {
83 public:
84     // Block[PrevIndex] pointer to prev block.
85     // Block[NextIndex] pointer to next block.
86     // Block[FreeIndex] index to next free block.
87     int64_t *Block;
88
89 public:
90     static const uint64_t PrevIndex = 0;
91     static const uint64_t NextIndex = 1;
92     static const uint64_t FreeIndex = 2;
93     static const uint64_t InitialFreeIndex = 3;
94     static const int64_t EndIndex = MemoryIncrease / sizeof(int64_t);
95     static_assert(MemoryIncrease >= 4 * BigAllocateThreshold,
96         "MemoryIncrease too small");
97     static_assert(MemoryIncrease % sizeof(int64_t) == 0, "bad
        MemoryIncrease");
98
99 public:
100     explicit HeapBlock(int64_t *Block, HeapBlock Prev, HeapBlock
        Next);
101     HeapBlock(int64_t *InitializedBlock);
102
103     int64_t *allocate(int64_t Size);
104
105     operator bool();
106
107     HeapBlock getPrev() { return (int64_t *)this->Block[PrevIndex];
        };
108     HeapBlock getNext() { return (int64_t *)this->Block[NextIndex];
        };
109
110     void setPrev(HeapBlock B) { this->Block[PrevIndex] =
        (int64_t)B.Block; };
111     void setNext(HeapBlock B) { this->Block[NextIndex] =
        (int64_t)B.Block; };
112
113     void zero();
114
115     bool operator==(HeapBlock Oth) { return this->Block ==
        Oth.Block; }
116     bool operator!=(HeapBlock Oth) { return !(*this == Oth); }
117
118     uint64_t getNumberFreeBytes() {
119         return MemoryIncrease - this->Block[FreeIndex] *
            sizeof(int64_t);
120     }
121 };
122
123 HeapBlock::HeapBlock(int64_t *Block, HeapBlock Prev, HeapBlock
    Next)
124     : Block(Block) {
125     this->Block[PrevIndex] = (int64_t)Prev.Block;
126     this->Block[NextIndex] = (int64_t)Next.Block;

```

```

127     this->Block[FreeIndex] = InitialFreeIndex;
128 }
129
130 HeapBlock::HeapBlock(int64_t *InitializedBlock) :
131     Block(InitializedBlock) {}
132
133 int64_t *HeapBlock::allocate(int64_t WordSize) {
134     assert(WordSize > 0);
135     int64_t NextIndex = this->Block[HeapBlock::FreeIndex] + WordSize;
136     if (NextIndex > HeapBlock::EndIndex)
137         return nullptr;
138     int64_t PrevIndex = this->Block[HeapBlock::FreeIndex];
139     this->Block[FreeIndex] = NextIndex;
140     return &this->Block[PrevIndex];
141 }
142
143 void HeapBlock::zero() {
144     assert(this->Block);
145     bzero(this->Block + 2, MemoryIncrease - 2 * sizeof(int64_t));
146     this->Block[FreeIndex] = HeapBlock::InitialFreeIndex;
147 }
148
149 HeapBlock::operator bool() { return this->Block; }
150
151 struct FirstGenMemoryBlock {
152     int64_t *Block;
153     uint64_t Index;
154 };
155
156 FirstGenMemoryBlock FirstGenBlock;
157
158 struct BigMemoryNode {
159     BigMemoryNode *Prev;
160     BigMemoryNode *Next;
161     int64_t *MemoryBlock;
162 };
163
164 BigMemoryNode BigMemoryList = {&BigMemoryList, &BigMemoryList,
165     nullptr};
166
167 HeapBlock ActiveHeapBlockList(0);
168 HeapBlock InactiveHeapBlockList(0);
169 HeapBlock ActiveHeapBlock(0);
170
171 static bool activeHasCapacity(uint64_t Bytes) {
172     assert(Bytes < MemoryIncrease - sizeof(int64_t) * 64);
173     if (ActiveHeapBlock.getNumberFreeBytes() >= Bytes)
174         return true;
175     if (ActiveHeapBlock.getNext() != ActiveHeapBlockList)
176         return true;
177     return false;
178 }
179
180 static bool activeHasFirstGenCapacity() {
181     return activeHasCapacity(FIRST_GEN_SIZE);
182 }
183
184 static const uint64_t SmallMemoryBit = (uint64_t)1 << 63;
185 static void markSmallMemory(int64_t *Obj) { Obj[-1] |=
186     SmallMemoryBit; }
187
188 static bool isSmallMemory(int64_t *Obj) { return Obj[-1] &

```

```

186         SmallMemoryBit; }
187 static const uint64_t ForwardBit = (uint64_t)1 << 62;
188 static void markForwarded(int64_t *Obj) { Obj[-1] |= ForwardBit; }
189
190 static void unmarkForwarded(int64_t *Obj) { Obj[-1] &=
191     ~ForwardBit; }
192
193 static bool isForwarded(int64_t *Obj) { return Obj[-1] &
194     ForwardBit; }
195
196 static int64_t getObjectSize(int64_t *Obj) {
197     return Obj[-1] & ~(SmallMemoryBit | ForwardBit);
198 }
199
200 static void bigMemoryNodeAppend(BigMemoryNode *Node, BigMemoryNode
201     *Prev,
202     BigMemoryNode *Next) {
203     Node->Prev = Prev;
204     Node->Next = Next;
205     Prev->Next = Node;
206     Next->Prev = Node;
207 }
208
209 static void bigMemoryNodeRemove(BigMemoryNode *Node) {
210     assert(Node != &BigMemoryList);
211     Node->Prev->Next = Node->Next;
212     Node->Next->Prev = Node->Prev;
213 }
214
215 static void bigMemoryNodeAppend(BigMemoryNode *Node) {
216     bigMemoryNodeAppend(Node, BigMemoryList.Prev, &BigMemoryList);
217 }
218
219 static int64_t *heapBlockListAllocate1(int64_t N, HeapBlock
220     ListHead) {
221     assert(N % sizeof(int64_t) == 0);
222     N = N / sizeof(int64_t);
223     assert(N > 0);
224
225     int64_t *Mem = ActiveHeapBlock.allocate(N);
226     if (Mem)
227         goto out;
228
229     ActiveHeapBlock = ActiveHeapBlock.getNext();
230     if (ActiveHeapBlock == ListHead)
231         return nullptr;
232
233     Mem = ActiveHeapBlock.allocate(N);
234     assert(Mem);
235
236 out:
237     Mem[0] = N;
238     auto Ret = Mem + 1;
239     markSmallMemory(Ret);
240     return Ret;
241 }
242
243 static int64_t *newBlock(uint64_t Size) {
244     GCMemoryUsage += Size;
245     if (GCMemoryUsage > MAX_ALLOC_SIZE)
246         error("out of memory");

```

```

243
244 #ifdef MMAP_ALLOC
245     void *Mem = mmap(nullptr, Size, PROT_READ | PROT_WRITE,
246                       MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
247     if (Mem == MAP_FAILED) {
248         perror("unable to allocate memory");
249         error("out of memory");
250     }
251 #else
252     void *Mem = calloc(1, Size);
253     if (!Mem) {
254         perror("unable to allocate memory");
255         error("out of memory");
256     }
257 #endif
258     return (int64_t *)Mem;
259 }
260
261 static void increaseMemory() {
262     auto ActivePrev = ActiveHeapBlockList.getPrev();
263     auto Block =
264         HeapBlock(newBlock(MemoryIncrease), ActivePrev,
265                     ActiveHeapBlockList);
266     ActivePrev.setNext(Block);
267     ActiveHeapBlockList.setPrev(Block);
268
269     auto InactivePrev = InactiveHeapBlockList.getPrev();
270     Block =
271         HeapBlock(newBlock(MemoryIncrease), InactivePrev,
272                     InactiveHeapBlockList);
273     InactivePrev.setNext(Block);
274     InactiveHeapBlockList.setPrev(Block);
275 }
276
277 static int64_t *fullMemoryForwardAlloc(uint64_t NumBytes) {
278     int64_t *Forward = heapBlockListAllocate1(NumBytes,
279                                                InactiveHeapBlockList);
280     if (!Forward) {
281         increaseMemory();
282         ActiveHeapBlock = InactiveHeapBlockList.getPrev();
283         Forward = heapBlockListAllocate1(NumBytes,
284                                          InactiveHeapBlockList);
285         assert(Forward);
286     }
287     assert(Forward[-1]);
288     return Forward;
289 }
290
291 static int64_t *fullMemoryCopy(int64_t *Object) {
292     // Ret is new memory field of Object.
293     int64_t Ret;
294
295     std::forward_list<std::pair<int64_t *, int64_t *>> MemObjStack;
296     // Push new memory field and associated object.
297     MemObjStack.emplace_front(&Ret, Object);
298
299     do {
300         // Pop new memory field and associated object from stack.
301         auto P = MemObjStack.front();
302         MemObjStack.pop_front();
303         int64_t *DestMem = P.first;
304         int64_t *Obj = P.second;

```

```

301
302     if (!istraced(Obj)) {
303         // Obj is not a heap pointer, so should not be forwarded.
304         // Just store its value to its new memory field.
305         *DestMem = (int64_t)Obj;
306         continue;
307     }
308
309     int64_t Size = getObjectSize(Obj);
310     assert(Size);
311
312     if (!isSmallMemory(Obj)) {
313         // Then the object is an array. We do not forward the object.
314         // We only forward its elements if they are non-scalar and
315         // they have not been forwarded yet.
316
317         // Index 1 of arrays contain the first element.
318         // If first element is non-scalar, then so are the rest.
319         if (!isForwarded(Obj) && istraced((int64_t *)Obj[1])) {
320             // Array contains non-scalar elements which have not yet
321             // been forwarded.
322             // Add them to list so they can be copied:
323             for (int64_t I = 1; I < Size - 1; ++I) {
324                 int64_t *Child = (int64_t *)Obj[I];
325                 MemObjStack.emplace_front(&Obj[I], Child);
326             }
327             // Store Obj to its new memory location:
328             *DestMem = (int64_t)Obj;
329             markForwarded(Obj);
330             continue;
331         }
332
333         if (isForwarded(Obj)) {
334             // Obj has already been forwarded.
335             // Store its forwarded address to new memory field.
336             *DestMem = *Obj;
337             continue;
338         }
339
340         // If we get here; Obj is a heap allocated pointer and has not
341         // yet been forwarded. So we forward is now and store its forwarded
342         // address in the new memory field.
343
344         // 1. Allocate memory from inactive heap. This is destination
345         // memory for fields from Obj.
346         int64_t *Forward = fullMemoryForwardAlloc(Size *
347             sizeof(int64_t));
348
349         // 2. Put fields from Obj in stack. Together with memory
350         // destination.
351         for (int64_t I = 0; I < Size - 1; ++I) {
352             int64_t *Child = (int64_t *)Obj[I];
353             MemObjStack.emplace_front(&Forward[I], Child);
354         }
355
356         // 3. Insert forward address in Obj and mark Obj forwarded.
357         *Obj = (int64_t)Forward;
358         markForwarded(Obj);

```

```

357
358     // 4. Store new memory address of Obj to new memory field.
359     *DestMem = (int64_t)Forward;
360 } while (!MemObjStack.empty());
361
362 assert(!isFirstGenPointer((int64_t *)Ret));
363 return (int64_t *)Ret;
364 }
365
366 static void bigMemoryCopy() {
367     auto Block = BigMemoryList.Next;
368     auto Prev = &BigMemoryList;
369     while (Block != &BigMemoryList) {
370         auto New = (BigMemoryNode
371             *)fullMemoryForwardAlloc(sizeof(BigMemoryNode) +
372                                     sizeof(int64_t));
373         assert(getObjectSize((int64_t *)New));
374         Prev->Next = New;
375         New->Prev = Prev;
376         New->MemoryBlock = Block->MemoryBlock;
377         Block = Block->Next;
378         Prev = New;
379     }
380     Prev->Next = &BigMemoryList;
381     BigMemoryList.Prev = Prev;
382 }
383
384 static void heapBlockListZero(HeapBlock List) {
385     auto B = List;
386     B.zero();
387     for (B = B.getNext(); B != List; B = B.getNext())
388         B.zero();
389 }
390
391 static void firstGenZero() {
392     bzero(FirstGenBlock.Block, FIRST_GEN_SIZE);
393     FirstGenBlock.Index = 0;
394 }
395
396 static void bigRelease(int64_t *Block) {
397     uint64_t Size = getObjectSize(Block) * sizeof(int64_t);
398     GCMemoryUsage -= Size;
399 #ifdef MMAP_ALLOC
400     if (munmap(Block - 1, Size)) {
401         perror("unable to deallocate memory block");
402         error("will not continue with unexpected deallocation error");
403     }
404 #else
405     free(Block - 1);
406 #endif
407 }
408
409 static void sweep() {
410     for (auto B = BigMemoryList.Next; B != &BigMemoryList;) {
411         auto Block = B;
412         auto Mem = B->MemoryBlock;
413         B = B->Next;
414         if (!isForwarded(Mem)) {
415             bigMemoryNodeRemove(Block);
416             bigRelease(Mem);
417         } else {
418             unmarkForwarded(Mem);

```

```

418     }
419 }
420 }
421
422 static void triggerFullGC(void *PrevFrame) {
423     // std::cout << " ***** TRIGGER FULL GC *****
424         \n";
425
426 #ifndef SSML_SHADOW_STACK_GC
427     int64_t *PrevF = (int64_t *)PrevFrame;
428     assert(isfunc(PrevF[1]));
429     FunctionFrame FirstFrame = {&PrevF[2]};
430 #endif
431
432 #ifdef SSML_GC_STATISTICS
433     ++FullGCCount;
434     uint64_t NumberRoots = 0;
435     Timer Tim;
436     Tim.start();
437 #endif // SSML_GC_STATISTICS
438
439     RememberSet.clear();
440
441     // Now the active heap block is beginning of inactive heap.
442     ActiveHeapBlock = InactiveHeapBlockList;
443
444 #ifdef SSML_SHADOW_STACK_GC
445     for (StackEntry *R = llvm_gc_root_chain; R; R = R->Next) {
446         assert(R->Map->NumMeta == 0);
447         auto Roots = R->Roots;
448         for (unsigned I = 1, E = R->Map->NumRoots; I != E; ++I) {
449             auto Root = (int64_t *)Roots[I];
450             auto Copy = fullMemoryCopy(Root);
451             assert(istraced(Copy) ? !isFirstGenPointer(Copy) : true);
452             Roots[I] = Copy;
453
454             assert(istraced(Root) && isSmallMemory(Root) ? *Root ==
455                 (int64_t)Copy
456                 : true);
457             assert(istraced(Root) ? getObjectSize(Root) ==
458                 getObjectSize(Copy)
459                 : true);
460 #ifdef SSML_GC_STATISTICS
461             ++NumberRoots;
462 #endif // SSML_GC_STATISTICS
463         }
464
465         // Mark stack frame as containing pointers to old generation.
466         R->Roots[0] = (void *)1;
467     }
468
469 #else // !defined(SSML_SHADOW_STACK_GC)
470     FunctionFrame Frame = FirstFrame;
471     for (;;) {
472         FrameDescr *Descr = getGCEntry(Frame.getFunction());
473         int16_t *Offsets = Descr->RootOffsets;
474
475         for (int64_t I = 1, Count = Descr->RootCount; I < Count; ++I) {
476             int16_t RootIndex = Offsets[I];
477             int64_t *Root = Frame.getRoot(RootIndex);
478             int64_t *Copy = fullMemoryCopy(Root);
479             Frame.setRoot(RootIndex, Copy);
480         }
481         Frame = Frame.getNext();
482     }
483 #endif
484 }

```

```

477     assert(istraced(Root) && isFirstGenPointer(Root) &&
478            isSmallMemory(Root)
479                ? *Root == (int64_t)Copy
480                : true);
481     assert(istraced(Root) ? getObjectSize(Root) ==
482            getObjectSize(Copy)
483                : true);
484 #ifdef SSML_GC_STATISTICS
485     ++NumberRoots;
486 #endif // SSML_GC_STATISTICS
487 }
488     if (isEntryFunction(Frame.getFunction()))
489         break;
490     // Mark stack frame as containing pointers to old generation.
491     Frame.setRoot(Offsets[0], (int64_t *)1);
492     Frame = Frame.nextFrame(Descr);
493 }
494
495 #endif // SSML_SHADOW_STACK_GC
496
497     bigMemoryCopy();
498     sweep();
499
500     std::swap(ActiveHeapBlockList, InactiveHeapBlockList);
501     heapBlockListZero(InactiveHeapBlockList);
502     firstGenZero();
503
504     if (ActiveHeapBlock.getNext() == ActiveHeapBlockList &&
505         !activeHasCapacity(MemoryIncrease / 2) &&
506         GCMemoryUsage + MemoryIncrease <= MAX_ALLOC_SIZE)
507         increaseMemory();
508
509 #ifdef SSML_GC_STATISTICS
510     TotalGCRoots += NumberRoots;
511
512     uint64_t Time = Tim.time();
513     if (Time > WhorstGCTime)
514         WhorstGCTime = Time;
515     if (Time < BestGCTime)
516         BestGCTime = Time;
517     TotalGCTime += Time;
518     ++TriggerGCCount;
519 #endif // SSML_GC_STATISTICS
520 }
521
522 static int64_t *doHeapBlockListAllocate(int64_t N) {
523     int64_t *Mem = heapBlockListAllocate1(N, ActiveHeapBlockList);
524     assert(Mem);
525     assert(Mem[-1]);
526     return Mem;
527 }
528
529 static int64_t *heapBlockListAllocate(int64_t N) {
530     return doHeapBlockListAllocate(N);
531 }
532
533 static int64_t *firstGenAllocateWords(int64_t Words) {
534     assert(FirstGenBlock.Index + Words <= FIRST_GEN_SIZE /
535            sizeof(int64_t));
536     int64_t *Ret = &FirstGenBlock.Block[FirstGenBlock.Index];

```

```

536     FirstGenBlock.Index += Words;
537     *Ret = Words;
538     ++Ret;
539     markSmallMemory(Ret);
540     return Ret;
541 }
542
543 static bool isFirstGenPointer(int64_t *Obj) {
544     uint64_t P = (uint64_t)Obj;
545     uint64_t Begin = (uint64_t)FirstGenBlock.Block;
546     uint64_t End = (uint64_t)FirstGenBlock.Block + FIRST_GEN_SIZE;
547     return P >= Begin && P < End;
548 }
549
550 static int64_t *firstGenMemoryCopy(int64_t *Object) {
551     // Ret is new memory field of Object.
552     int64_t Ret;
553
554     std::forward_list<std::pair<int64_t *, int64_t *>> MemObjStack;
555     // Push new memory field and associated object.
556     MemObjStack.emplace_front(&Ret, Object);
557
558     do {
559         // Pop new memory field and associated object from stack.
560         auto P = MemObjStack.front();
561         MemObjStack.pop_front();
562         int64_t *DestMem = P.first;
563         int64_t *Obj = P.second;
564
565         if (!istraced(Obj)) {
566             // Obj is not a heap pointer, so should not be forwarded.
567             // Just store its value to its new memory field.
568             *DestMem = (int64_t)Obj;
569             continue;
570         }
571
572         if (!isFirstGenPointer(Obj)) {
573             // Obj is not pointer to first generation.
574             // Thus we don't forward it.
575             *DestMem = (int64_t)Obj;
576             continue;
577         }
578
579         if (isForwarded(Obj)) {
580             // Obj has already been forwarded.
581             // Store its forwarded address to new memory field.
582             *DestMem = *Obj;
583             continue;
584         }
585
586         // If we get here; Obj is a heap allocated pointer from first
587         // generation.
588         // It has not yet been forwarded, so we forward it now and
589         // store its
590         // forwarded address in the new memory field.
591
592         int64_t Size = getObjectSize(Obj);
593         assert(Size);
594         assert(Size < BigAllocateThreshold);
595
596         // Put fields from Obj in stack. Together with memory
597         // destination.

```

```

595     int64_t *Forward = heapBlockListAllocate(Size *
596         sizeof(int64_t));
597     for (int64_t I = 0; I < Size - 1; ++I) {
598         int64_t *Child = (int64_t *)Obj[I];
599         MemObjStack.emplace_front(&Forward[I], Child);
600     }
601     // Insert forward address in Obj and mark Obj forwarded.
602     *Obj = (int64_t)Forward;
603     markForwarded(Obj);
604
605     // Store new memory address of Obj to new memory field.
606     *DestMem = (int64_t)Forward;
607 } while (!MemObjStack.empty());
608
609 return (int64_t *)Ret;
610 }
611
612 static void firstGenRememberedCopy(int64_t *Obj) {
613     assert(istraced(Obj));
614     assert(!isFirstGenPointer(Obj));
615
616     int64_t Size = getObjectSize(Obj);
617     assert(Size);
618
619     for (int64_t I = 0; I < Size - 1; ++I) {
620         int64_t *Child = (int64_t *)Obj[I];
621         Obj[I] = (int64_t)firstGenMemoryCopy(Child);
622     }
623 }
624
625 static void triggerFirstGenGC(void *PrevFrame) {
626     // std::cout << " ***** FIRST GEN GC ***** \n";
627     #ifndef SSML_SHADOW_STACK_GC
628         int64_t *PrevF = (int64_t *)PrevFrame;
629         assert(isfunc(PrevF[1]));
630         FunctionFrame FirstFrame = {&PrevF[2]};
631     #endif
632
633     #ifdef SSML_GC_STATISTICS
634         ++FirstGenGCCount;
635         uint64_t NumberRoots = 0;
636         Timer Tim;
637         Tim.start();
638     #endif // SSML_GC_STATISTICS
639
640     for (auto P : RememberSet)
641         firstGenRememberedCopy(P);
642     RememberSet.clear();
643
644     #ifdef SSML_SHADOW_STACK_GC
645     for (StackEntry *R = llvm_gc_root_chain; R; R = R->Next) {
646         assert(R->Map->NumMeta == 0);
647         auto Roots = R->Roots;
648         for (unsigned I = 1, E = R->Map->NumRoots; I != E; ++I) {
649             auto Root = (int64_t *)Roots[I];
650             auto Copy = firstGenMemoryCopy(Root);
651             Roots[I] = Copy;
652
653             assert(istraced(Root) && isFirstGenPointer(Root) &&
654                 isSmallMemory(Root)
655                 ? *Root == (int64_t)Copy

```

```

655         : true);
656         assert(istraced(Root) ? getObjectSize(Root) ==
           getObjectSize(Copy)
           : true);
657
658 #ifdef SSML_GC_STATISTICS
659     ++NumberRoots;
660 #endif // SSML_GC_STATISTICS
661 }
662
663     if (R->Roots[0]) {
664         assert(R->Roots[0] == (void *)1);
665         // Is this stack frame is marked as old stack frame; break
           now.
666         break;
667     }
668     // Mark stack frame as containing pointers to old generation.
669     R->Roots[0] = (void *)1;
670 }
671
672 #else // !defined(SSML_SHADOW_STACK_GC)
673     FunctionFrame Frame = FirstFrame;
674     for (;;) {
675         FrameDescr *Descr = getGCEntry(Frame.getFunction());
676         int16_t *Offsets = Descr->RootOffsets;
677         for (int64_t I = 1, Count = Descr->RootCount; I < Count; ++I) {
678             int16_t RootIndex = Offsets[I];
679             int64_t *Root = Frame.getRoot(RootIndex);
680             int64_t *Copy = firstGenMemoryCopy(Root);
681             Frame.setRoot(RootIndex, Copy);
682
683             assert(istraced(Root) && isFirstGenPointer(Root) &&
               isSmallMemory(Root)
               ? *Root == (int64_t)Copy
               : true);
684             assert(istraced(Root) ? getObjectSize(Root) ==
               getObjectSize(Copy)
               : true);
685
686 #ifdef SSML_GC_STATISTICS
687             ++NumberRoots;
688 #endif // SSML_GC_STATISTICS
689         }
690
691         if (Frame.getRoot(Offsets[0])) {
692             assert(Frame.getRoot(Offsets[0]) == (int64_t *)1);
693             // Is this stack frame is marked as old stack frame; break
               now.
694             break;
695         }
696         // Mark stack frame as containing pointers to old generation.
697         Frame.setRoot(Offsets[0], (int64_t *)1);
698         Frame = Frame.nextFrame(Descr);
699     }
700 #endif // SSML_SHADOW_STACK_GC
701
702     firstGenZero();
703
704 #ifdef SSML_GC_STATISTICS
705     TotalGCRoots += NumberRoots;
706
707     uint64_t Time = Tim.time();
708     if (Time > WhorstGCTime)
709         WhorstGCTime = Time;

```

```

712     if (Time < BestGCTime)
713         BestGCTime = Time;
714     TotalGCTime += Time;
715     ++TriggerGCCCount;
716 #endif // SSML_GC_STATISTICS
717 }
718
719 static int64_t *firstGenAllocate(int64_t N, void *PrevFrame) {
720     int64_t Words = N / sizeof(int64_t);
721     if (FirstGenBlock.Index + Words <= FIRST_GEN_SIZE /
722         sizeof(int64_t)) {
723         return firstGenAllocateWords(Words);
724     }
725     if (activeHasFirstGenCapacity()) {
726         triggerFirstGenGC(PrevFrame);
727     } else {
728         triggerFullGC(PrevFrame);
729         if (!activeHasFirstGenCapacity())
730             increaseMemory();
731     }
732     assert(FirstGenBlock.Index == 0);
733     return firstGenAllocateWords(Words);
734 }
735
736 static int64_t *bigAllocate(int64_t N, void *PrevFrame) {
737     assert(N % sizeof(int64_t) == 0);
738
739     N = N + sizeof(int64_t);
740
741     static const uint64_t NodeSize = sizeof(BigMemoryNode) +
742         sizeof(int64_t);
743
744     if (N + GCMemoryUsage > MAX_ALLOC_SIZE ||
745         !activeHasCapacity(NodeSize))
746         triggerFullGC(PrevFrame);
747
748     if (!activeHasCapacity(NodeSize))
749         increaseMemory();
750
751     // Allocate node for holding big memory block.
752     BigMemoryNode *Node = (BigMemoryNode
753         *)heapBlockListAllocate(NodeSize);
754     bigMemoryNodeAppend(Node);
755
756     // Allocate memory. Need room for size of object; add size of
757     int64_t.
758     int64_t *Mem = newBlock(N);
759     *Mem = N / sizeof(int64_t);
760
761     Node->MemoryBlock = ++Mem;
762     return Mem;
763 }
764
765 extern "C" void ssmlGCInit() {
766     int64_t *Block = newBlock(MemoryIncrease);
767     ActiveHeapBlockList = HeapBlock(Block, Block, Block);
768
769     ActiveHeapBlock = ActiveHeapBlockList;
770
771     Block = newBlock(MemoryIncrease);
772     InactiveHeapBlockList = HeapBlock(Block, Block, Block);
773 }

```



```

769     FirstGenBlock.Block = newBlock(FIRST_GEN_SIZE);
770     setFirstGenBegin((uint64_t)FirstGenBlock.Block);
771     setFirstGenEnd((uint64_t)FirstGenBlock.Block + FIRST_GEN_SIZE);
772 }
773
774 extern "C" int64_t *allocateFrame(int64_t N, void *Frame) {
775     assert(N % sizeof(int64_t) == 0);
776
777     if (!N)
778         return nullptr;
779
780     // Need to store size of object also; add one more word to
781     // memory size.
782     N = N + sizeof(int64_t);
783     if (N <= BigAllocateThreshold) {
784         auto Ret = firstGenAllocate(N, Frame);
785         return Ret;
786     }
787     return bigAllocate(N, Frame);
788 }
789
790 extern "C" int64_t *allocate(int64_t N) {
791     return allocateFrame(N, __builtin_frame_address(0));
792 }
793 #endif // SSML_GENERATIONAL_GC

```

Listing 6: runtime/CopyingGC.cpp

```

1  #ifndef SSML_COPYING_GC
2
3  #include "GC.h"
4  #include "ssml/Common/Timer.h"
5
6  #include <strings.h>
7  #include <iostream>
8  #include <stdint.h>
9  #include <stdlib.h>
10 #include <sys/mman.h>
11 #include <forward_list>
12
13 #define MMAP_ALLOC
14
15 using namespace ssml;
16
17 extern "C" void error(const char *);
18
19 static uint64_t TriggerGCCount;
20 static uint64_t TotalGCTime;
21 static uint64_t BestGCTime = UINT64_MAX;
22 static uint64_t WhorstGCTime;
23 static uint64_t TotalGCRoots;
24 static uint64_t GCMemoryUsage;
25
26 extern "C" void ssmlGCPrintStatistics() {
27     std::cout << "***** Copying GC Statistics *****\n";
28     #ifdef SSML_SHADOW_STACK_GC
29         std::cout << "Used Shadow Stack GC strategy\n";
30     #else
31         std::cout << "Used Binary Map GC strategy\n";
32     #endif

```

```

33     std::cout << "Total GCs:                " << TriggerGCCount << '\n';
34     if (!TriggerGCCount)
35         return;
36
37     double Tot = (double)TotalGCTime / 1000;
38     double Avg = ((double)TotalGCTime / TriggerGCCount) / 1000;
39     double Whorst = (double)WhorstGCTime / 1000;
40     double Best = (double)BestGCTime / 1000;
41     double AvgRoots = (double)TotalGCRoots / TriggerGCCount;
42     uint64_t Avail = MAX_ALLOC_SIZE - GCMemoryUsage;
43
44     std::cout << "Total GC Time:                " << Tot << "ms\n";
45     std::cout << "Average GC Time:                " << Avg << "ms\n";
46     std::cout << "Whorst GC Time:                " << Whorst << "ms\n";
47     std::cout << "Best GC Time:                " << Best << "ms\n";
48     std::cout << "Average number Roots:        " << AvgRoots << '\n';
49     std::cout << "Total Memory:                " << MAX_ALLOC_SIZE <<
        "B\n";
50     std::cout << "GC Memory Usage:                " << GCMemoryUsage << "B\n";
51     std::cout << "Total Free Memory:            " << Avail << "B\n";
52     std::cout << "*****" << std::endl;
53 }
54
55 // When allocating more memory than this, use a mark sweep
    algorithm instead.
56 static const int64_t BigAllocateThreshold = 16384; // 2^14 bytes.
57 // static const int64_t BigAllocateThreshold = 1024;
58 // Both heaps are increased by MemoryIncrease when more memory is
    needed.
59 static const int64_t MemoryIncrease = 2097152 * 32; // 2 * 32 MB.
60
61 class HeapBlock {
62 private:
63     // Block[PrevIndex] pointer to prev block.
64     // Block[NextIndex] pointer to next block.
65     // Block[FreeIndex] index to next free block.
66     int64_t *Block;
67
68 public:
69     static const uint64_t PrevIndex = 0;
70     static const uint64_t NextIndex = 1;
71     static const uint64_t FreeIndex = 2;
72     static const uint64_t InitialFreeIndex = 3;
73     static const int64_t EndIndex = MemoryIncrease / sizeof(int64_t);
74     static_assert(MemoryIncrease >= 4 * BigAllocateThreshold,
75         "MemoryIncrease too small");
76     static_assert(MemoryIncrease % sizeof(int64_t) == 0, "bad
        MemoryIncrease");
77
78 public:
79     explicit HeapBlock(int64_t *Block, HeapBlock Prev, HeapBlock
        Next);
80     HeapBlock(int64_t *InitializedBlock);
81
82     int64_t *allocate(int64_t Size);
83
84     operator bool();
85
86     HeapBlock getPrev() { return (int64_t *)this->Block[PrevIndex];
        };
87     HeapBlock getNext() { return (int64_t *)this->Block[NextIndex];
        };

```

```

88
89     void setPrev(HeapBlock B) { this->Block[PrevIndex] =
90         (int64_t)B.Block; };
91     void setNext(HeapBlock B) { this->Block[NextIndex] =
92         (int64_t)B.Block; };
93
94     void zero();
95
96     bool operator==(HeapBlock Oth) { return this->Block ==
97         Oth.Block; }
98     bool operator!=(HeapBlock Oth) { return !(*this == Oth); }
99
100     uint64_t getNumberFreeBytes() {
101         return MemoryIncrease - this->Block[FreeIndex] *
102             sizeof(int64_t);
103     }
104 };
105
106 HeapBlock::HeapBlock(int64_t *Block, HeapBlock Prev, HeapBlock
107     Next)
108     : Block(Block) {
109     this->Block[PrevIndex] = (int64_t)Prev.Block;
110     this->Block[NextIndex] = (int64_t)Next.Block;
111     this->Block[FreeIndex] = InitialFreeIndex;
112 }
113
114 HeapBlock::HeapBlock(int64_t *InitializedBlock) :
115     Block(InitializedBlock) {}
116
117 int64_t *HeapBlock::allocate(int64_t WordSize) {
118     assert(WordSize > 0);
119     int64_t NextIndex = this->Block[HeapBlock::FreeIndex] + WordSize;
120     if (NextIndex > HeapBlock::EndIndex)
121         return nullptr;
122     int64_t PrevIndex = this->Block[HeapBlock::FreeIndex];
123     this->Block[FreeIndex] = NextIndex;
124     return &this->Block[PrevIndex];
125 }
126
127 void HeapBlock::zero() {
128     assert(this->Block);
129     bzero(this->Block + 2, MemoryIncrease - 2 * sizeof(int64_t));
130     this->Block[FreeIndex] = HeapBlock::InitialFreeIndex;
131 }
132
133 HeapBlock::operator bool() { return this->Block; }
134
135 struct BigMemoryNode {
136     BigMemoryNode *Prev;
137     BigMemoryNode *Next;
138     int64_t *MemoryBlock;
139 };
140
141 BigMemoryNode BigMemoryList = {&BigMemoryList, &BigMemoryList,
142     nullptr};
143
144 HeapBlock ActiveHeapBlockList(0);
145 HeapBlock InactiveHeapBlockList(0);
146 HeapBlock ActiveHeapBlock(0);
147
148 static bool activeHasCapacity(uint64_t Bytes) {
149     assert(Bytes < MemoryIncrease - sizeof(int64_t) * 64);

```

```

143     if (ActiveHeapBlock.getNumberFreeBytes() >= Bytes)
144         return true;
145     if (ActiveHeapBlock.getNext() != ActiveHeapBlockList)
146         return true;
147     return false;
148 }
149
150 static const uint64_t SmallMemoryBit = (uint64_t)1 << 63;
151 static void markSmallMemory(int64_t *Obj) { Obj[-1] |=
    SmallMemoryBit; }
152
153 static bool isSmallMemory(int64_t *Obj) { return Obj[-1] &
    SmallMemoryBit; }
154
155 static const uint64_t ForwardBit = (uint64_t)1 << 62;
156 static void markForwarded(int64_t *Obj) { Obj[-1] |= ForwardBit; }
157
158 static void unmarkForwarded(int64_t *Obj) { Obj[-1] &=
    ~ForwardBit; }
159
160 static bool isForwarded(int64_t *Obj) { return Obj[-1] &
    ForwardBit; }
161
162 static int64_t getObjectSize(int64_t *Obj) {
163     return Obj[-1] & ~(SmallMemoryBit | ForwardBit);
164 }
165
166 static void bigMemoryNodeAppend(BigMemoryNode *Node, BigMemoryNode
    *Prev,
167                                 BigMemoryNode *Next) {
168     Node->Prev = Prev;
169     Node->Next = Next;
170     Prev->Next = Node;
171     Next->Prev = Node;
172 }
173
174 static void bigMemoryNodeRemove(BigMemoryNode *Node) {
175     assert(Node != &BigMemoryList);
176     Node->Prev->Next = Node->Next;
177     Node->Next->Prev = Node->Prev;
178 }
179
180 static void bigMemoryNodeAppend(BigMemoryNode *Node) {
181     bigMemoryNodeAppend(Node, BigMemoryList.Prev, &BigMemoryList);
182 }
183
184 static int64_t *heapBlockListAllocate1(int64_t N, HeapBlock
    ListHead) {
185     assert(N % sizeof(int64_t) == 0);
186     N = N / sizeof(int64_t);
187     assert(N > 0);
188
189     int64_t *Mem = ActiveHeapBlock.allocate(N);
190     if (Mem)
191         goto out;
192
193     ActiveHeapBlock = ActiveHeapBlock.getNext();
194     if (ActiveHeapBlock == ListHead)
195         return nullptr;
196
197     Mem = ActiveHeapBlock.allocate(N);
198     assert(Mem);

```

```

199
200 out:
201     Mem[0] = N | SmallMemoryBit;
202     return Mem + 1;
203 }
204
205 static int64_t *newBlock(uint64_t Size) {
206     GCMemoryUsage += Size;
207     if (GCMemoryUsage > MAX_ALLOC_SIZE)
208         error("out of memory");
209
210 #ifdef MMAP_ALLOC
211     void *Mem = mmap(nullptr, Size, PROT_READ | PROT_WRITE,
212                      MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
213     if (Mem == MAP_FAILED) {
214         perror("unable to allocate memory");
215         error("out of memory");
216     }
217 #else
218     void *Mem = calloc(1, Size);
219     if (!Mem) {
220         perror("unable to allocate memory");
221         error("out of memory");
222     }
223 #endif
224
225     return (int64_t *)Mem;
226 }
227
228 static void increaseMemory() {
229     auto ActivePrev = ActiveHeapBlockList.getPrev();
230     auto Block =
231         HeapBlock(newBlock(MemoryIncrease), ActivePrev,
232                  ActiveHeapBlockList);
233     ActivePrev.setNext(Block);
234     ActiveHeapBlockList.setPrev(Block);
235
236     auto InactivePrev = InactiveHeapBlockList.getPrev();
237     Block =
238         HeapBlock(newBlock(MemoryIncrease), InactivePrev,
239                  InactiveHeapBlockList);
240     InactivePrev.setNext(Block);
241     InactiveHeapBlockList.setPrev(Block);
242 }
243
244 static int64_t *memoryForwardAlloc(uint64_t NumBytes) {
245     int64_t *Forward = heapBlockListAllocate1(NumBytes,
246                                                InactiveHeapBlockList);
247     if (!Forward) {
248         // This actually can happen if the copied objects are put in
249         // the new
250         // memory spaces in a less space efficient way.
251         increaseMemory();
252         ActiveHeapBlock = InactiveHeapBlockList.getPrev();
253         Forward = heapBlockListAllocate1(NumBytes,
254                                          InactiveHeapBlockList);
255         assert(Forward);
256     }
257     assert(Forward[-1]);
258     return Forward;
259 }

```

```

256 static int64_t *memoryCopy(int64_t *Object) {
257     // Ret is new memory field of Object.
258     int64_t Ret;
259
260     std::forward_list<std::pair<int64_t *, int64_t *>> MemObjStack;
261     // Push new memory field and associated object.
262     MemObjStack.emplace_front(&Ret, Object);
263
264     do {
265         // Pop new memory field and associated object from stack.
266         auto P = MemObjStack.front();
267         MemObjStack.pop_front();
268         int64_t *DestMem = P.first;
269         int64_t *Obj = P.second;
270
271         if (!istraced(Obj)) {
272             // Obj is not a heap pointer, so should not be forwarded.
273             // Just store its value to its new memory field.
274             *DestMem = (int64_t)Obj;
275             continue;
276         }
277
278         int64_t Size = getObjectSize(Obj);
279         assert(Size);
280
281         if (!isSmallMemory(Obj)) {
282             // Then the object is an array. We do not forward the object.
283             // We only forward its elements if they are non-scalar and
284             // they have not been forwarded yet.
285
286             // Index 1 of arrays contain the first element.
287             // If first element is non-scalar, then so are the rest.
288             if (!isForwarded(Obj) && istraced((int64_t *)Obj[1])) {
289                 // Array contains non-scalar elements which have not yet
290                 // been forwarded.
291                 // Add them to list so they can be copied:
292                 for (int64_t I = 1; I < Size - 1; ++I) {
293                     int64_t *Child = (int64_t *)Obj[I];
294                     MemObjStack.emplace_front(&Obj[I], Child);
295                 }
296                 // Store Obj to its new memory location:
297                 *DestMem = (int64_t)Obj;
298                 markForwarded(Obj);
299                 continue;
300             }
301
302             if (isForwarded(Obj)) {
303                 // Obj has already been forwarded.
304                 // Store its forwarded address to new memory field.
305                 *DestMem = *Obj;
306                 continue;
307             }
308
309             // If we get here; Obj is a heap allocated pointer and has not
310             // yet been forwarded. So we forward is now and store its forwarded
311             // address in the new memory field.
312
313             // 1. Allocate memory from inactive heap. This is destination
314             // memory for

```

```

314     // fields from Obj.
315     int64_t *Forward = memoryForwardAlloc(Size * sizeof(int64_t));
316
317     // 2. Put fields from Obj in stack. Together with memory
318         destination.
319     for (int64_t I = 0; I < Size - 1; ++I) {
320         int64_t *Child = (int64_t *)Obj[I];
321         MemObjStack.emplace_front(&Forward[I], Child);
322     }
323
324     // 3. Insert forward address in Obj and mark Obj forwarded.
325     *Obj = (int64_t)Forward;
326     markForwarded(Obj);
327
328     // 4. Store new memory address of Obj to new memory field.
329     *DestMem = (int64_t)Forward;
330 } while (!MemObjStack.empty());
331
332 return (int64_t *)Ret;
333 }
334
335 static void bigMemoryCopy() {
336     auto Block = BigMemoryList.Next;
337     auto Prev = &BigMemoryList;
338     while (Block != &BigMemoryList) {
339         auto New = (BigMemoryNode
340             *)memoryForwardAlloc(sizeof(BigMemoryNode) +
341                                 sizeof(int64_t));
342         Prev->Next = New;
343         New->Prev = Prev;
344         New->MemoryBlock = Block->MemoryBlock;
345         Block = Block->Next;
346         Prev = New;
347     }
348     Prev->Next = &BigMemoryList;
349     BigMemoryList.Prev = Prev;
350 }
351
352 static void heapBlockListZero(HeapBlock List) {
353     auto B = List;
354     B.zero();
355     for (B = B.getNext(); B != List; B = B.getNext())
356         B.zero();
357 }
358
359 static void bigRelease(int64_t *Block) {
360     uint64_t Size = getObjectSize(Block) * sizeof(int64_t);
361     GCMemoryUsage -= Size;
362 #ifdef MMAP_ALLLOC
363     if (munmap(Block - 1, Size)) {
364         perror("unable to deallocate memory block");
365         error("will not continue with unexpected deallocation error");
366     }
367 #else
368     free(Block - 1);
369 #endif
370 }
371
372 static void sweep() {
373     for (auto B = BigMemoryList.Next; B != &BigMemoryList;) {
374         auto Block = B;
375         auto Mem = B->MemoryBlock;

```

```

374     B = B->Next;
375     if (!isForwarded(Mem)) {
376         bigMemoryNodeRemove(Block);
377         bigRelease(Mem);
378     } else {
379         unmarkForwarded(Mem);
380     }
381 }
382 }
383
384 static void triggerGC(FunctionFrame FirstFrame) {
385     // std::cout << " ***** TRIGGER GC ***** \n";
386     #ifdef SSML_GC_STATISTICS
387         uint64_t NumberRoots = 0;
388         Timer Tim;
389         Tim.start();
390     #endif // SSML_GC_STATISTICS
391
392     // Now the active heap block is beginning of inactive heap.
393     ActiveHeapBlock = InactiveHeapBlockList;
394
395     #ifdef SSML_SHADOW_STACK_GC
396     for (StackEntry *R = llvm_gc_root_chain; R; R = R->Next) {
397         assert(R->Map->NumMeta == 0);
398         auto Roots = R->Roots;
399         for (unsigned I = 0, E = R->Map->NumRoots; I != E; ++I) {
400             auto Root = (int64_t *)Roots[I];
401             auto Copy = memoryCopy(Root);
402             Roots[I] = Copy;
403
404             assert(istraced(Root) && isSmallMemory(Root) ? *Root ==
405                    (int64_t)Copy
406                    : true);
407             assert(istraced(Root) ? getObjectSize(Root) ==
408                    getObjectSize(Copy)
409                    : true);
410         }
411     }
412     #else // !defined(SSML_SHADOW_STACK_GC)
413     FunctionFrame Frame = FirstFrame;
414     for (;;) {
415         FrameDescr *Descr = getGCEntry(Frame.getFunction());
416         int16_t *Offsets = Descr->RootOffsets;
417
418         for (int64_t I = 0, Count = Descr->RootCount; I < Count; ++I) {
419             int16_t RootIndex = Offsets[I];
420             int64_t *Root = Frame.getRoot(RootIndex);
421             int64_t *Copy = memoryCopy(Root);
422             Frame.setRoot(RootIndex, Copy);
423
424             assert(istraced(Root) && isSmallMemory(Root) ? *Root ==
425                    (int64_t)Copy
426                    : true);
427         }
428         #ifdef SSML_GC_STATISTICS
429             ++NumberRoots;
430         #endif // SSML_GC_STATISTICS
431     }
432     if (isEntryFunction(Frame.getFunction()))

```



```

433         break;
434         Frame = Frame.nextFrame(Descr);
435     }
436 #endif // SSML_SHADOW_STACK_GC
437
438     bigMemoryCopy();
439     sweep();
440
441     std::swap(ActiveHeapBlockList, InactiveHeapBlockList);
442     heapBlockListZero(InactiveHeapBlockList);
443
444     if (ActiveHeapBlock.getNext() == ActiveHeapBlockList &&
445         !activeHasCapacity(MemoryIncrease / 2) &&
446         GCMemoryUsage + MemoryIncrease <= MAX_ALLOC_SIZE)
447         increaseMemory();
448
449 #ifdef SSML_GC_STATISTICS
450     TotalGCRoots += NumberRoots;
451
452     uint64_t Time = Tim.time();
453     if (Time > WhorstGCTime)
454         WhorstGCTime = Time;
455     if (Time < BestGCTime)
456         BestGCTime = Time;
457     TotalGCTime += Time;
458     ++TriggerGCCount;
459 #endif // SSML_GC_STATISTICS
460 }
461
462 static int64_t *doHeapBlockListAllocate(int64_t N, void
463     *PrevFrame) {
464     int64_t *Mem = heapBlockListAllocate1(N, ActiveHeapBlockList);
465     if (Mem) {
466         assert(Mem[-1]);
467         return Mem;
468     }
469
470     // If we get here there is not enough allocated memory left.
471     // Trigger a GC and see if we get enough new memory.
472     int64_t *Prev = (int64_t *)PrevFrame;
473 #ifndef SSML_SHADOW_STACK_GC
474     assert(isfunc(Prev[1]));
475 #endif
476     triggerGC(&Prev[2]);
477
478     // Now after GC, we might have some free memory.
479     Mem = heapBlockListAllocate1(N, ActiveHeapBlockList);
480     if (Mem) {
481         assert(Mem[-1]);
482         return Mem;
483     }
484
485     // Still, after GC there is not enough memory.
486     // Allocate a new memory block.
487     increaseMemory();
488     ActiveHeapBlock = ActiveHeapBlockList.getPrev();
489
490     // Now, we must have memory. Otherwise the program should have
491     // crashed.
492     Mem = heapBlockListAllocate1(N, ActiveHeapBlockList);
493     assert(Mem);
494     assert(Mem[-1]);

```

```

493     return Mem;
494 }
495
496 static int64_t *heapBlockListAllocate(int64_t N, void *PrevFrame) {
497     auto Ret = doHeapBlockListAllocate(N, PrevFrame);
498     markSmallMemory(Ret);
499     return Ret;
500 }
501
502 static int64_t *bigAllocate(int64_t N, void *PrevFrame) {
503     assert(N % sizeof(int64_t) == 0);
504
505     N = N + sizeof(int64_t);
506
507     if (N + GCMemoryUsage > MAX_ALLOC_SIZE) {
508         int64_t *Prev = (int64_t *)PrevFrame;
509 #ifndef SSML_SHADOW_STACK_GC
510         assert(isfunc(Prev[1]));
511 #endif
512         triggerGC({&Prev[2]});
513     }
514
515     // Allocate node for holding big memory block.
516     BigMemoryNode *Node =
517         (BigMemoryNode *)allocateFrame(sizeof(BigMemoryNode),
518                                         PrevFrame);
519     bigMemoryNodeAppend(Node);
520
521     int64_t *Mem = newBlock(N);
522     *Mem = N / sizeof(int64_t);
523
524     Node->MemoryBlock = ++Mem;
525     return Mem;
526 }
527
528 extern "C" void ssmlGCInit() {
529     int64_t *Block = newBlock(MemoryIncrease);
530     ActiveHeapBlockList = HeapBlock(Block, Block, Block);
531
532     ActiveHeapBlock = ActiveHeapBlockList;
533
534     Block = newBlock(MemoryIncrease);
535     InactiveHeapBlockList = HeapBlock(Block, Block, Block);
536 }
537
538 extern "C" int64_t *allocateFrame(int64_t N, void *Frame) {
539     assert(N % sizeof(int64_t) == 0);
540
541     if (!N)
542         return nullptr;
543
544     // Need to store size of object also; add one more word to
545     // memory size.
546     N = N + sizeof(int64_t);
547     if (N <= BigAllocateThreshold) {
548         auto Ret = heapBlockListAllocate(N, Frame);
549         return Ret;
550     }
551     return bigAllocate(N, Frame);
552 }
553
554 extern "C" int64_t *allocate(int64_t N) {

```

```

553     return allocateFrame(N, __builtin_frame_address(0));
554 }
555
556 #endif // SSML_COPYING_GC

```

Listing 7: runtime/MarkSweepAlloc.cpp

```

1  #ifdef SSML_MARK_SWEEP_GC
2
3  #include "GC.h"
4  #include "ssml/Common/Timer.h"
5
6  #include <stdlib.h>
7  #include <assert.h>
8  #include <stdint.h>
9  #include <errno.h>
10 #include <sys/mman.h>
11 #include <strings.h>
12
13 #include <algorithm>
14 #include <iostream>
15 #include <set>
16 #include <forward_list>
17
18 #pragma GCC diagnostic ignored "-pedantic"
19
20 using namespace ssml;
21
22 #define MMAP_ALLOC
23
24 #define UNUSED __attribute__((unused))
25
26 extern "C" void ssmlGCInit() {}
27
28 static uint64_t BigAllocSize;
29 static uint64_t NormalAllocSize;
30 static uint64_t NormalAllocUsage;
31 static const uint64_t GCTriggerIncrease = MAX_ALLOC_SIZE / 16;
32 static uint64_t GCTriggerPoint = GCTriggerIncrease;
33
34 static uint64_t TriggerGCCount;
35 static uint64_t TotalGCTime;
36 static uint64_t BestGCTime = UINT64_MAX;
37 static uint64_t WhorstGCTime;
38 static uint64_t TotalGCRoots;
39
40 extern "C" void ssmlGCPrintStatistics() {
41     std::cout << "**** Mark Sweep: GC Statistics ****\n";
42     #ifdef SSML_SHADOW_STACK_GC
43         std::cout << "Used Shadow Stack GC strategy\n";
44     #else
45         std::cout << "Used Binary Map GC strategy\n";
46     #endif
47     std::cout << "Total GCs:          " << TriggerGCCount << '\n';
48     if (!TriggerGCCount)
49         return;
50
51     double Tot = (double)TotalGCTime / 1000;
52     double Avg = ((double)TotalGCTime / TriggerGCCount) / 1000;
53     double Whorst = (double)WhorstGCTime / 1000;
54     double Best = (double)BestGCTime / 1000;

```

```

55     double AvgRoots = (double)TotalGCRoots / TriggerGCCount;
56
57     uint64_t Avail = MAX_ALLOC_SIZE - BigAllocSize - GCTriggerPoint;
58
59     std::cout << "Total GC Time:          " << Tot << "ms\n";
60     std::cout << "Average GC Time:          " << Avg << "ms\n";
61     std::cout << "Whorst GC Time:          " << Whorst << "ms\n";
62     std::cout << "Best GC Time:           " << Best << "ms\n";
63     std::cout << "Average number Roots:   " << AvgRoots << '\n';
64     std::cout << "Total Memory:          " << MAX_ALLOC_SIZE <<
        "B\n";
65     std::cout << "GC Memory Usage:        " << NormalAllocSize <<
        "B\n";
66     std::cout << "GC Threshold:           " << GCTriggerPoint <<
        "B\n";
67     std::cout << "Big Memory Usage:        " << BigAllocSize << "B\n";
68     std::cout << "Total Free Memory:       " << Avail << "B\n";
69     std::cout << "*****" << std::endl;
70 }
71
72 namespace {
73 class MemoryBlock {
74     friend MemoryBlock splitFreeList(int64_t FreeListIndex);
75     friend MemoryBlock popFreeList(int64_t FreeListIndex);
76     friend void removeFreeList(int64_t FreeListIndex, MemoryBlock B);
77     friend void pushFreeList(int64_t FreeListIndex, MemoryBlock B);
78
79 public:
80     static const int64_t ObjectMarkBit = (int64_t)1 << 63;
81     static const int64_t ObjectMarkMask = ~ObjectMarkBit;
82     static const int64_t IsAllocatedBit = (int64_t)1 << 62;
83     static const int64_t IsAllocatedMask = ~IsAllocatedBit;
84     // SizeBitmask assumes FreeListMaxIndex <= 20.
85     static const int64_t SizeBitmask = 0x7fffff;
86     static const int OffsetShift = std::ceil(std::log2(SizeBitmask));
87     static const int64_t OffsetMask = 0x7fffffffffffffff;
88     // Bits indices 61-23 are free for offset bytes.
89     // Smallest block is 3 words + 1 word for size and offset.
90     static const int MinimumWordSize = 3;
91
92 private:
93     // When block is in a free list. Block[1] is prev block in the
94     // free list
95     // and Block[2] is next block in free list.
96     int64_t *Block;
97
98 private:
99     // Split this block into two equally sized blocks.
100     // Assumes the block is nulled.
101     std::pair<MemoryBlock, MemoryBlock> split();
102     // Set pointer to prev block in free list.
103     void setPrev(MemoryBlock N);
104     // Set pointer to next block in free list.
105     void setNext(MemoryBlock N);
106
107 public:
108     // Initialize with nullptr.
109     explicit MemoryBlock();
110     // Assumes Block is an initialized memory block, already
111     // contains word size
112     // and byte offset.
113     explicit MemoryBlock(int64_t *Block);

```

```

112 // Assumes Block is nulled. It will override the size/offset
    field.
113 explicit MemoryBlock(int64_t *Block, int64_t WordSize, int64_t
    ByteOffset);
114 // Returns number of words in memory words
115 // (excluding the first word in the block).
116 int64_t getWordSize();
117 // Returns byte offset of this block to the base block.
118 int64_t getOffset();
119 // Address of first usable word of this block. (&this->Block[1]).
120 int64_t *get();
121 // Address of this block.
122 int64_t *getRaw();
123 // Get prev memory block in free list.
124 MemoryBlock getPrev();
125 // Get next memory block in free list.
126 MemoryBlock getNext();
127 // Returns true if all usable memory is zeroed. Used for
    debugging.
128 bool isNulled();
129
130 operator bool();
131 };
132 } // End anonymous namespace.
133
134 // Mark object starting at Obj[-1] as visited. Return word size of
    object.
135 static int64_t mark(int64_t *Obj);
136
137 // Mark object starting at Obj[-1] as not visited.
138 static void unmark(int64_t *Obj);
139
140 // Return whether object starting at Obj[-1] is marked as visited.
141 static bool ismarked(int64_t *Obj);
142
143 // Recursively mark object starting at Root[-1] and all objects
    directly or
144 // indirectly pointed to.
145 static void markall(int64_t *Root);
146
147 // Free heap allocated object starting at Obj[-1]. Return memory
    block which
148 // have been freed. The returned block is not the same block as
    Obj when
149 // freeing of Obj results in a merge with its buddy block(s).
150 static MemoryBlock release(int64_t *Obj);
151
152 // Sweep phase in mark & sweep algorithm.
153 static void sweep();
154
155 // Free list index 16 is for 2MB blocks.
156 static const int64_t FreeListMaxIndex = 16;
157 static_assert(FreeListMaxIndex < 20, "FreeLists size too big");
158
159 static const int64_t FreeListCount = FreeListMaxIndex + 1;
160 // Index 0 contains blocks of size 32 bytes.
161 // Index 1 contains blocks of size 64 bytes.
162 // ...
163 // Index FreeListMaxIndex contains blocks of size MemoryIncrease
    bytes.
164 static MemoryBlock FreeLists[FreeListCount];
165 // Index 0 of each block contains the block size in words.

```

```

166 // Index 1 contains address of the base block.
167 // Index 2 is pointer to next block in the free list.
168
169 // Memory increase for each base block allocation.
170 static const int64_t MemoryIncrease = 32 * std::pow(2,
    FreeListMaxIndex);
171
172 std::pair<MemoryBlock, MemoryBlock> MemoryBlock::split() {
173     auto Size = this->getWordSize() + 1;
174     auto Off = this->getOffset();
175     assert(this->getWordSize() > MemoryBlock::MinimumWordSize);
176
177     auto NewSize = Size >> 1;
178     MemoryBlock First(this->Block, NewSize - 1, Off);
179
180     auto NewOff = Off + NewSize * sizeof(int64_t);
181     MemoryBlock Second(&this->Block[NewSize], NewSize - 1, NewOff);
182
183     assert(First.getWordSize() == Size / 2 - 1);
184     assert(First.getWordSize() == Second.getWordSize());
185     assert(First.getWordSize() >= MemoryBlock::MinimumWordSize);
186     assert(First.isNulled());
187     assert(Second.isNulled());
188
189     return {First, Second};
190 }
191
192 // Merge memory block A and memory block B into 1 block.
193 // Assumes the blocks are not currently in a free list
194 // And that the blocks are nulled.
195 static MemoryBlock memoryBlockMergeNulled(MemoryBlock A,
    MemoryBlock B) {
196     assert(A.getWordSize() >= MemoryBlock::MinimumWordSize);
197     assert(A.getWordSize() == B.getWordSize());
198
199     if ((uint64_t)A.getRaw() > (uint64_t)B.getRaw())
200         std::swap(A, B);
201
202     assert((uint64_t)A.getRaw() + (A.getWordSize() + 1) *
        sizeof(int64_t) ==
203            (uint64_t)B.getRaw());
204
205     int64_t NewWordSize = (A.getWordSize() + 1) * 2 - 1;
206     int64_t Offset = A.getOffset();
207
208     assert((NewWordSize + 1) * sizeof(int64_t) % 32 == 0);
209
210     B.getRaw()[0] = 0;
211     MemoryBlock Ret(A.getRaw(), NewWordSize, Offset);
212     assert(Ret.isNulled());
213     return Ret;
214 }
215
216 MemoryBlock::MemoryBlock() : Block(nullptr) {}
217
218 MemoryBlock::MemoryBlock(int64_t *Block) : Block(Block) {}
219
220 MemoryBlock::MemoryBlock(int64_t *Block, int64_t WordSize, int64_t
    ByteOffset)
221     : Block(Block) {
222     this->Block[0] = WordSize;
223     this->Block[0] |= ByteOffset << MemoryBlock::OffsetShift;

```

```

224 }
225
226 int64_t MemoryBlock::getWordSize() {
227     return this->Block[0] & MemoryBlock::SizeBitmask;
228 }
229
230 int64_t MemoryBlock::getOffset() {
231     return (this->Block[0] & MemoryBlock::OffsetMask) >>
        MemoryBlock::OffsetShift;
232 }
233
234 int64_t *MemoryBlock::get() {
235     assert(this->Block);
236     return &this->Block[1];
237 }
238
239 int64_t *MemoryBlock::getRaw() { return this->Block; }
240
241 MemoryBlock MemoryBlock::getPrev() {
242     return MemoryBlock((int64_t *)this->Block[1]);
243 }
244
245 MemoryBlock MemoryBlock::getNext() {
246     return MemoryBlock((int64_t *)this->Block[2]);
247 }
248
249 void MemoryBlock::setPrev(MemoryBlock N) {
250     assert(N ? this->getWordSize() == N.getWordSize() : 1);
251     this->Block[1] = (int64_t)N.Block;
252 }
253
254 void MemoryBlock::setNext(MemoryBlock N) {
255     assert(N ? this->getWordSize() == N.getWordSize() : 1);
256     this->Block[2] = (int64_t)N.Block;
257 }
258
259 bool MemoryBlock::isNulled() {
260     assert(this->Block);
261
262     int64_t Size = this->getWordSize();
263     int64_t *Ptr = this->get();
264     for (int64_t I = 0; I < Size; ++I)
265         if (Ptr[I])
266             return false;
267     return true;
268 }
269
270 MemoryBlock::operator bool() { return this->Block; }
271
272 // Get index of free list in FreeLists when (word) size of block
    is Size.
273 // Note that Size is not including the first word of the block.
274 static int64_t freeListIndex(int64_t Size) {
275     int64_t Ret;
276     Size += sizeof(int64_t);
277     asm("bsrq %1, %0\n\t" : "=r"(Ret) : "r"(Size));
278     if (Size & (Size - 1))
279         Ret += 1;
280     return std::max(Ret - 5, (int64_t)0);
281 }
282
283 // Get head of free list at FreeListIndex.

```

```

284 static MemoryBlock headFreeList(int64_t FreeListIndex) {
285     assert(FreeListIndex < FreeListCount);
286
287     MemoryBlock Ret = FreeLists[FreeListIndex];
288
289     assert(
290         Ret ? freeListIndex(Ret.getWordSize() * sizeof(int64_t)) ==
                FreeListIndex
                : 1);
291
292
293     return Ret;
294 }
295
296 namespace {
297 // Append to free list at FreeListIndex by making B the new head
    of the list.
298 void pushFreeList(int64_t FreeListIndex, MemoryBlock B) {
299     assert(!B);
300     assert(!B.getPrev());
301     assert(B.getWordSize() >= MemoryBlock::MinimumWordSize);
302     assert(freeListIndex(B.getWordSize() * sizeof(int64_t)) ==
            FreeListIndex);
303
304     MemoryBlock Next = FreeLists[FreeListIndex];
305     B.setNext(Next);
306     if (Next)
307         Next.setPrev(B);
308     FreeLists[FreeListIndex] = B;
309 }
310 } // End anonymous namespace.
311
312 namespace {
313 // Remove head of free list at FreeListIndex.
314 MemoryBlock popFreeList(int64_t FreeListIndex) {
315     MemoryBlock B = headFreeList(FreeListIndex);
316     assert(!B);
317
318     MemoryBlock Next = B.getNext();
319     if (Next)
320         Next.setPrev(MemoryBlock());
321     B.setNext(MemoryBlock());
322     FreeLists[FreeListIndex] = Next;
323
324     assert(B.getWordSize() >= MemoryBlock::MinimumWordSize);
325     return B;
326 }
327 } // End anonymous namespace.
328
329 namespace {
330 void removeFreeList(int64_t FreeListIndex, MemoryBlock B) {
331     MemoryBlock Prev = B.getPrev();
332     MemoryBlock Next = B.getNext();
333     if (!Prev) {
334         FreeLists[FreeListIndex] = Next;
335     } else {
336         Prev.setNext(Next);
337         B.setPrev(MemoryBlock());
338     }
339     if (Next) {
340         Next.setPrev(Prev);
341         B.setNext(MemoryBlock());
342     }

```

```

343     assert(Prev ? Prev.getWordSize() >= MemoryBlock::MinimumWordSize
344             : 1);
345     assert(Next ? Next.getWordSize() >= MemoryBlock::MinimumWordSize
346             : 1);
347     assert(Next ? (Prev ? Next.getWordSize() == Prev.getWordSize() :
348                   1) : 1);
349 }
350 } // End anonymous namespace.
351
352 namespace {
353 // Split head of free list at FreeListIndex. Insert one part
354 // in free list at FreeListIndex - 1 and return the other part.
355 MemoryBlock splitFreeList(int64_t FreeListIndex) {
356     assert(FreeListIndex > 0);
357     assert(FreeListIndex < FreeListCount);
358
359     MemoryBlock B = popFreeList(FreeListIndex);
360
361     auto P = B.split();
362     pushFreeList(FreeListIndex - 1, P.second);
363
364     assert(P.first.getWordSize() >= MemoryBlock::MinimumWordSize);
365     assert(P.second.getWordSize() >= MemoryBlock::MinimumWordSize);
366     assert(P.first.getWordSize() == P.second.getWordSize());
367     return P.first;
368 }
369 } // End anonymous namespace.
370
371 struct BaseBlockNode {
372     int64_t *BlockOffset;
373     BaseBlockNode *Prev;
374     BaseBlockNode *Next;
375 };
376 static BaseBlockNode *BaseBlockList;
377
378 struct BigBlockNode {
379     BigBlockNode *Prev;
380     BigBlockNode *Next;
381     int64_t WordSize;
382 };
383 static BigBlockNode *BigBlockList;
384
385 static void triggerGC(FunctionFrame FirstFrame) {
386     // std::cout << " ***** TRIGGER GC ***** \n";
387 #ifdef SSML_GC_STATISTICS
388     uint64_t NumberRoots = 0;
389     Timer Tim;
390     Tim.start();
391 #endif // SSML_GC_STATISTICS
392
393     assert(NormalAllocSize >= NormalAllocUsage);
394
395     // First mark list nodes used to hold base blocks.
396     for (BaseBlockNode *N = BaseBlockList; N; N = N->Next)
397         mark((int64_t *)N);
398
399 #ifdef SSML_SHADOW_STACK_GC
400     for (StackEntry *R = llvm_gc_root_chain; R; R = R->Next) {
401         assert(R->Map->NumMeta == 0);
402         auto Roots = R->Roots;
403         for (unsigned I = 0, E = R->Map->NumRoots; I != E; ++I) {

```

```

402         markall((int64_t *)Roots[I]);
403 #ifdef SSML_GC_STATISTICS
404         ++NumberRoots;
405 #endif // SSML_GC_STATISTICS
406     }
407 }
408 #else // !defined(SSML_SHADOW_STACK_GC)
409     FunctionFrame Frame = FirstFrame;
410     for (;;) {
411         FrameDescr *Descr = getGCEntry(Frame.getFunction());
412         int16_t *Offsets = Descr->RootOffsets;
413
414         for (int64_t I = 0, Count = Descr->RootCount; I < Count; ++I) {
415             markall(Frame.getRoot(Offsets[I]));
416 #ifdef SSML_GC_STATISTICS
417             ++NumberRoots;
418 #endif // SSML_GC_STATISTICS
419         }
420
421         if (isEntryFunction(Frame.getFunction()))
422             break;
423         Frame = Frame.nextFrame(Descr);
424     }
425 #endif // SSML_SHADOW_STACK_GC
426
427     sweep();
428
429     assert(NormalAllocSize >= NormalAllocUsage);
430     if ((NormalAllocSize - NormalAllocUsage) * 4 < GCTriggerIncrease
        * 3)
431         GCTriggerPoint = std::min(GCTriggerPoint + GCTriggerIncrease,
432                                   MAX_ALLOC_SIZE - BigAllocSize);
433
434 #ifdef SSML_GC_STATISTICS
435     TotalGCRoots += NumberRoots;
436
437     uint64_t Time = Tim.time();
438     if (Time > WhorstGCTime)
439         WhorstGCTime = Time;
440     if (Time < BestGCTime)
441         BestGCTime = Time;
442     TotalGCTime += Time;
443     ++TriggerGCCount;
444 #endif // SSML_GC_STATISTICS
445 }
446
447 static bool needGC(int64_t Size) {
448     return NormalAllocSize + Size > GCTriggerPoint;
449 }
450
451 // Allocate zero initialized memory aligned with page size.
452 static int64_t *newBlock(uint64_t Size) {
453 #ifdef MMAP_ALLOC
454     void *Mem = mmap(nullptr, Size, PROT_READ | PROT_WRITE,
455                      MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
456     if (Mem == MAP_FAILED) {
457         perror("unable to allocate memory");
458         error("out of memory");
459     }
460 #else
461     void *Mem = calloc(1, Size);
462     if (!Mem) {

```

```

463     perror("unable to allocate memory");
464     error("out of memory");
465 }
466 #endif
467 return (int64_t *)Mem;
468 }
469
470 static int64_t *pushNewMemoryBlock() {
471     int64_t *Block = newBlock(MemoryIncrease);
472     assert(Block);
473
474     NormalAllocSize += MemoryIncrease;
475
476     /* Block with word size and offset 0. */
477     MemoryBlock MemB(Block, (MemoryIncrease / sizeof(int64_t)) - 1,
478         0);
479     pushFreeList(FreeListMaxIndex, MemB);
480     return Block;
481 }
482
483 static int64_t *addNewBlock(void *PrevFrame) {
484     int64_t *Block;
485     if (needGC(MemoryIncrease)) {
486         int64_t *Prev = (int64_t *)PrevFrame;
487         #ifndef SSML_SHADOW_STACK_GC
488             assert(isfunc(Prev[1]));
489         #endif
490         FunctionFrame Frame = {&Prev[2]};
491         triggerGC(Frame);
492         Block = nullptr;
493     } else {
494         Block = pushNewMemoryBlock();
495     }
496     return Block;
497 }
498
499 static MemoryBlock multiSplitFreeList(int64_t HighIdx, int64_t
500     LowIdx) {
501     assert(HighIdx < FreeListCount);
502     assert(LowIdx >= 0);
503     assert(!headFreeList(HighIdx));
504
505     if (HighIdx == LowIdx)
506         return popFreeList(HighIdx);
507
508     MemoryBlock Ret;
509     for (;;) {
510         Ret = splitFreeList(HighIdx);
511         --HighIdx;
512         if (HighIdx == LowIdx)
513             break;
514         pushFreeList(HighIdx, Ret);
515     }
516     return Ret;
517 }
518
519 static int64_t *getFromFreeList(int64_t Idx) {
520     if (headFreeList(Idx))
521         return popFreeList(Idx).get();
522
523     for (int64_t H = Idx + 1; H < FreeListCount; ++H) {
524         if (headFreeList(H))

```

```

523         return multiSplitFreeList(H, Idx).get();
524     }
525
526     return nullptr;
527 }
528
529 static int64_t *freeListAllocate(int64_t N, void *Frame) {
530     assert(N <= MemoryIncrease);
531     int64_t Idx = freeListIndex(N);
532
533     int64_t *Ret = getFromFreeList(Idx);
534     if (Ret)
535         goto free_list_allocate_out;
536
537     {
538         int64_t *NewB = addNewBlock(Frame);
539         if (!NewB) {
540             /* We allocated as much memory as we are allowed to at
541              * the moment. GC have been triggered, we might have a
542              * block in the free list now. */
543             Ret = getFromFreeList(Idx);
544             if (Ret)
545                 goto free_list_allocate_out;
546
547             /* We were not able to find a block.
548              * Now we try to increase the amount of memory allocated
549              * before a
550              * GC is triggered and allocate a new base block. */
551             auto PrevGCPoint = GCTriggerPoint;
552             assert(PrevGCPoint <= MAX_ALLOC_SIZE - BigAllocSize);
553             GCTriggerPoint = std::min(GCTriggerPoint + GCTriggerIncrease,
554                                     MAX_ALLOC_SIZE - BigAllocSize);
555             if (GCTriggerPoint - PrevGCPoint < (uint64_t)MemoryIncrease)
556             {
557                 /* Not allowed to increase amount of memory before GC. */
558                 error("out of memory doing normal allocation");
559             }
560             NewB = pushNewMemoryBlock();
561         }
562         /* We have just allocated a new base block of memory.
563          * Which we have put in the free list. Split this block to
564          * get a block of correct size for the user. */
565         Ret = multiSplitFreeList(FreeListMaxIndex, Idx).get();
566         /* Alloc memory for MemoryBlockNode to put NewB in the
567          * allocated list. */
568         BaseBlockNode *Node = (BaseBlockNode
569                                *)allocate(sizeof(BaseBlockNode));
570         Node->BlockOffset = NewB;
571         Node->Next = BaseBlockList;
572         if (BaseBlockList)
573             BaseBlockList->Prev = Node;
574         BaseBlockList = Node;
575     }
576
577 free_list_allocate_out:
578
579     NormalAllocUsage += (Ret[-1] & MemoryBlock::SizeBitmask) *
580                         sizeof(int64_t);
581     assert(NormalAllocUsage <= NormalAllocSize);
582     return Ret;
583 }
584
585 }

```

```

580 static int64_t *bigAllocate(int64_t N, void *PrevFrame) {
581     uint64_t Total = N + 3 * sizeof(int64_t);
582
583     /*std::cout << "BIG ALLOCATE: " << Total << "\n";*/
584
585     if (GCTriggerPoint + BigAllocSize + Total > MAX_ALLOC_SIZE) {
586         /* There is not enough memory.
587          * Trigger GC. Maybe we can find the necessary memory. */
588         int64_t *Prev = (int64_t *)PrevFrame;
589 #ifndef SSML_SHADOW_STACK_GC
590         assert(isfunc(Prev[1]));
591 #endif
592         FunctionFrame Frame = {&Prev[2]};
593         triggerGC(Frame);
594         if (GCTriggerPoint + BigAllocSize + Total > MAX_ALLOC_SIZE)
595             error("out of memory doing big allocation");
596     }
597
598     BigBlockNode *Block = (BigBlockNode *)newBlock(Total);
599     BigAllocSize += Total;
600
601     Block->WordSize = N / sizeof(int64_t);
602     Block->Next = BigBlockList;
603     if (BigBlockList)
604         BigBlockList->Prev = Block;
605     BigBlockList = Block;
606
607     return &((int64_t *)Block)[3];
608 }
609
610 static void bigRelease(BigBlockNode *Block) {
611     uint64_t Total = (Block->WordSize + 3) * sizeof(int64_t);
612
613     // std::cout << "BIG RELEASE: " << Total << "\n";
614
615     BigAllocSize -= Total;
616
617 #ifdef MMAP_ALLOC
618     if (munmap(Block, Total)) {
619         perror("unable to deallocate memory block");
620         error("will not continue with unexpected deallocation error");
621     }
622 #else
623     free(Block);
624 #endif
625
626     // std::cout << "BigAllocMemory is now: " << BigAllocSize <<
627     // "\n";
628 }
629
630 // Mark object starting at Obj[-1] as allocated.
631 static void markAllocated(int64_t *Obj) {
632     assert((Obj[-1] & MemoryBlock::SizeBitmask) >=
633           MemoryBlock::MinimumWordSize);
634     Obj[-1] |= MemoryBlock::IsAllocatedBit;
635 }
636
637 // Return whether this block pointed to by Obj is marked as
638 // allocated.
639 bool isAllocated(int64_t *Obj) {
640     assert((Obj[-1] & MemoryBlock::SizeBitmask) >=
641           MemoryBlock::MinimumWordSize);

```

```

638     return Obj[-1] & MemoryBlock::IsAllocatedBit;
639 }
640
641 // Mark ibject starting at Obj[-1] as free.
642 static void unmarkAllocated(int64_t *Obj) {
643     assert((Obj[-1] & MemoryBlock::SizeBitmask) >=
644             MemoryBlock::MinimumWordSize);
645     Obj[-1] &= MemoryBlock::IsAllocatedMask;
646 }
647
648 extern "C" int64_t *allocateFrame(int64_t N, void *Frame) {
649     int64_t *Ret;
650
651     assert(NormalAllocSize <= GCTriggerPoint);
652     assert(GCTriggerPoint + BigAllocSize <= MAX_ALLOC_SIZE);
653
654     assert(N % sizeof(int64_t) == 0);
655     if (!N)
656         return 0;
657
658     if (N < MemoryIncrease)
659         Ret = freeListAllocate(N, Frame);
660     else
661         Ret = bigAllocate(N, Frame);
662
663     markAllocated(Ret);
664     assert((Ret[-1] & MemoryBlock::SizeBitmask) >=
665             MemoryBlock::MinimumWordSize);
666
667     assert(MemoryBlock(&Ret[-1]).isNull());
668     assert(NormalAllocSize <= GCTriggerPoint);
669     assert(GCTriggerPoint + BigAllocSize <= MAX_ALLOC_SIZE);
670
671     return Ret;
672 }
673
674 extern "C" int64_t *allocate(int64_t N) {
675     return allocateFrame(N, __builtin_frame_address(0));
676 }
677
678 static MemoryBlock release(int64_t *Obj) {
679     assert(isAllocated(Obj));
680     assert(!ismarked(Obj));
681
682     unmarkAllocated(Obj);
683
684     MemoryBlock B(&Obj[-1]);
685     // Zero the released memory block.
686     bzero(B.get(), B.getWordSize() * sizeof(int64_t));
687
688     assert(NormalAllocUsage >= B.getWordSize() * sizeof(int64_t));
689     NormalAllocUsage -= B.getWordSize() * sizeof(int64_t);
690
691     int64_t FreeListIndex;
692     for (;;) {
693         assert(B.getWordSize());
694
695         int64_t TotalByteSize = (B.getWordSize() + 1) *
696             sizeof(int64_t);
697         assert(TotalByteSize % 32 == 0);
698
699         FreeListIndex = freeListIndex(TotalByteSize - sizeof(int64_t));

```

```

697     if (TotalByteSize == MemoryIncrease)
698         break;
699
700     uint64_t Address = (uint64_t)B.getRaw();
701     uint64_t NextSize = TotalByteSize * 2;
702
703     uint64_t Buddy;
704     if (B.getOffset() % NextSize == 0)
705         Buddy = Address + TotalByteSize;
706     else
707         Buddy = Address - TotalByteSize;
708
709     if (isAllocated((int64_t *)Buddy + 1))
710         break;
711
712     MemoryBlock BuddyBlock((int64_t *)Buddy);
713
714     assert(!ismarked((int64_t *)Buddy + 1));
715     assert((BuddyBlock.getWordSize() + 1) * sizeof(int64_t) % 32
716           == 0);
717     assert(BuddyBlock.getWordSize() <= B.getWordSize());
718
719     if (BuddyBlock.getWordSize() != B.getWordSize())
720         break;
721
722     removeFreeList(FreeListIndex, BuddyBlock);
723     B = memoryBlockMergeNulled(B, BuddyBlock);
724     assert((uint64_t)B.getWordSize() ==
725           (TotalByteSize * 2) / sizeof(int64_t) - 1);
726 }
727
728 assert(B.getWordSize() >= MemoryBlock::MinimumWordSize);
729 pushFreeList(FreeListIndex, B);
730
731 return B;
732 }
733
734 static int64_t mark(int64_t *Obj) {
735     int64_t Size = Obj[-1] & MemoryBlock::SizeBitmask;
736     Obj[-1] |= MemoryBlock::ObjectMarkBit;
737     return Size;
738 }
739
740 static void unmark(int64_t *Obj) { Obj[-1] &=
741     MemoryBlock::ObjectMarkMask; }
742
743 static bool ismarked(int64_t *Obj) {
744     return Obj[-1] & MemoryBlock::ObjectMarkBit;
745 }
746
747 #if 0
748 static void markall(int64_t *Root) {
749     if (!istraced(Root) || ismarked(Root))
750         return;
751     assert(isAllocated(Root));
752
753     int64_t Size = mark(Root);
754     for (int64_t I = 0; I < Size; ++I)
755         markall((int64_t *)Root[I]);
756 }
757 #else
758 // Function markall marks object Root and all objects

```

```

757 // reachable from Root. If Root is not dynamically allocated
758 // or if is not a pointer, then the function does nothing.
759 static void markall(int64_t *Root) {
760     // If Root is not a dynamically allocated pointer or
761     // if Root is already marked, then return:
762     if (!istraced(Root) || ismarked(Root))
763         return;
764
765     // Linked list containing reachable unmarked objects:
766     std::forward_list<int64_t *> Objects;
767     // Initialliy Root is unmarked:
768     Objects.push_front(Root);
769
770     do {
771         // Get Obj from front of list:
772         int64_t *Obj = Objects.front();
773         // Remove front of list:
774         Objects.pop_front();
775
776         assert(isAllocated(Obj));
777
778         // Mark Obj, and return the word size of Obj:
779         int64_t Size = mark(Obj);
780         // For each word in Obj:
781         for (int64_t I = 0; I < Size; ++I) {
782             // Get Child from index I of Obj:
783             int64_t *Child = (int64_t *)Obj[I];
784             // If Child is a dynamically allocated object and
785             // it has not been marked, then put it in list with
786             // unmarked reachable objects:
787             if (istraced(Child) && !ismarked(Child))
788                 Objects.push_front(Child);
789         }
790         // While there are reachable unmarked objects:
791     } while (!Objects.empty());
792 }
793 #endif
794
795 static void sweep() {
796     for (auto N = BaseBlockList; N; N = N->Next) {
797         uint64_t Addr = (uint64_t)N->BlockOffset;
798         uint64_t End = Addr + MemoryIncrease;
799         uint64_t NextAddr = Addr;
800         for (MemoryBlock B((int64_t *)Addr); NextAddr < End;) {
801             auto Obj = B.get();
802             if (ismarked(Obj))
803                 unmark(Obj);
804             else if (isAllocated(Obj))
805                 B = release(Obj);
806
807             uint64_t Inc = (B.getWordSize() + 1) * sizeof(int64_t);
808             assert(Inc <= (uint64_t)MemoryIncrease);
809
810             Addr = (uint64_t)B.getRaw();
811             NextAddr = Addr + Inc;
812             B = MemoryBlock((int64_t *)NextAddr);
813         }
814     }
815
816     for (auto Block = BigBlockList; Block;) {
817         BigBlockNode *B = Block;
818         Block = Block->Next;

```



```

819
820     int64_t *Obj = &((int64_t *)B)[3];
821     if (ismarked(Obj)) {
822         unmark(Obj);
823     } else {
824         if (B->Prev)
825             B->Prev->Next = Block;
826         else
827             BigBlockList = Block;
828         if (Block)
829             Block->Prev = B->Prev;
830         assert(isAllocated(Obj));
831         bigRelease(B);
832     }
833 }
834 }
835
836 #endif // SSML_MARK_SWEEP_GC

```

Listing 8: runtime/Main.c

```

1  #include <stdlib.h>
2  #include <stdint.h>
3
4  #ifdef SSML_GENERATIONAL_GC
5  uint64_t FirstGenBegin;
6  uint64_t FirstGenEnd;
7
8  void setFirstGenBegin(uint64_t V) { FirstGenBegin = V; }
9  void setFirstGenEnd(uint64_t V) { FirstGenEnd = V; }
10
11 #endif /* SSML_GENERATIONAL_GC */
12
13 int64_t entry();
14
15 void initFramePrintVal();
16
17 void initStdLib();
18
19 void ssmlGCInit();
20
21 #ifdef SSML_GC_STATISTICS
22 void ssmlGCPrintStatistics();
23 #endif /* SSML_GC_STATISTICS */
24
25 /*void *gcmap;*/
26 int16_t gclookup_size;
27 int64_t *gclookup_map;
28
29 static void initGC() {
30 #if !defined(SSML_SHADOW_STACK_GC) && !defined(SSML_DISABLE_GC)
31     /*asm("\tmovq $ssml_gcmap, %0\n" : "=r"(gcmap) :);*/
32     asm("\tmovw ssml_gclookup_size, %0\n" : "=r"(gclookup_size) :);
33     asm("\tmovq $ssml_gclookup_map, %0\n" : "=r"(gclookup_map) :);
34 #endif
35     ssmlGCInit();
36 }
37
38 int main() {
39     int ret;
40     initFramePrintVal();

```

```

41     initStdLib();
42     initGC();
43     ret = entry();
44 #ifdef SSML_GC_STATISTICS
45     ssmlGCPrintStatistics();
46 #endif /* SSML_GC_STATISTICS */
47     return ret;
48 }

```

Listing 9: runtime/GC.h

```

1  #ifndef LLVM_TOOLS_SSML_RUNTIME_GC_H
2  #define LLVM_TOOLS_SSML_RUNTIME_GC_H
3
4  #include <iostream>
5  #include <stdint.h>
6  #include <assert.h>
7
8  #pragma GCC diagnostic ignored "-pedantic"
9
10 #define MAX_ALLOC_SIZE ((uint64_t)1048576 * 2048) // 1MB * 2048.
11
12 extern "C" void error(const char *);
13
14 #ifdef SSML_SHADOW_STACK_GC
15 struct FrameMap {
16     int32_t NumRoots;
17     int32_t NumMeta;
18     const void *Meta[0];
19 };
20
21 struct StackEntry {
22     StackEntry *Next;
23     const FrameMap *Map;
24     void *Roots[0];
25 };
26
27 extern StackEntry *llvm_gc_root_chain;
28 #endif
29
30 extern "C" int16_t gclookup_size;
31 extern "C" int64_t *gclookup_map;
32
33 inline bool isstatic(int64_t *Obj) {
34     extern char __executable_start;
35     //extern char __etext;
36     extern char end;
37     return &__executable_start <= (char *)Obj && (char *)Obj <= &end;
38 }
39
40 inline bool istraced(int64_t *Obj) {
41     if (!Obj || isstatic(Obj))
42         return false;
43     return !((int64_t)Obj & 1);
44 }
45
46 inline int64_t codeBegin() { return gclookup_map[0]; }
47
48 inline int64_t codeEnd() { return gclookup_map[gclookup_size * 2]; }
49

```

```

50 inline bool isfunc(int64_t Obj) {
51     return Obj >= codeBegin() && Obj < codeEnd();
52 }
53
54 struct FrameDescr {
55     int16_t StackFrameSize;
56     int16_t RootCount;
57     int16_t RootOffsets[0];
58 };
59
60 namespace {
61 struct FunctionFrame {
62     int64_t *Frame;
63
64     int64_t getFunction() { return Frame[-1]; }
65
66     struct FunctionFrame nextFrame(FrameDescr *D) {
67         FunctionFrame Ret = {&Frame[D->StackFrameSize + 1]};
68         assert(isfunc(Ret.getFunction()));
69         return Ret;
70     }
71
72     int64_t *getRoot(int16_t Index) {
73         return (int64_t *)this->Frame[Index];
74     }
75
76     void setRoot(int16_t Index, int64_t *Root) {
77         this->Frame[Index] = (int64_t)Root;
78     }
79 };
80 } // End anonymous namespace.
81
82 inline int64_t *gclookup(int64_t Idx) {
83     assert(Idx < gclookup_size);
84     return &gclookup_map[Idx * 2];
85 }
86
87 inline int64_t gclookup_compare(int64_t *LookupElem, int64_t Addr)
88 {
89     if (LookupElem[2] <= Addr)
90         return 1;
91     if (LookupElem[0] > Addr)
92         return -1;
93     return 0;
94 }
95
96 inline bool isEntryFunction(int64_t Addr) {
97     return !gclookup_compare(gclookup(0), Addr);
98 }
99
100 inline FrameDescr *getGCEntry(int64_t Addr) {
101     int64_t Min = 0;
102     int64_t Max = gclookup_size - 1;
103
104     while (Min <= Max) {
105         int64_t Mid = Min + (Max - Min) / 2;
106         int64_t *LookupElem = gclookup(Mid);
107         int64_t Cmp = gclookup_compare(LookupElem, Addr);
108         if (!Cmp)
109             return (FrameDescr *)LookupElem[1];
110         if (Cmp == 1)
111             Min = Mid + 1;

```

```

111     else
112         Max = Mid - 1;
113     }
114
115     std::cerr << "unable to find function address: " << (void *)Addr
116         << std::endl;
117     error("GC function address was not found");
118     return nullptr;
119 }
120
121 extern "C" void ssmlGCPrintStatistics();
122
123 extern "C" int64_t *allocateFrame(int64_t N, void *Frame);
124
125 extern "C" int64_t *allocate(int64_t N);
126
127 #endif // LLVM_TOOLS_SSML_RUNTIME_GC_H

```

Listing 10: runtime/MallocMarkSweepAlloc.cpp

```

1  #if 0
2  #ifdef SSML_MARK_SWEEP_GC
3
4  #include "GC.h"
5  #include "ssml/Common/Timer.h"
6
7  #include <stdlib.h>
8  #include <assert.h>
9  #include <stdint.h>
10 #include <errno.h>
11 #include <sys/mman.h>
12 #include <strings.h>
13
14 #include <algorithm>
15 #include <iostream>
16 #include <set>
17 #include <forward_list>
18
19 #pragma GCC diagnostic ignored "-pedantic"
20
21 using namespace ssml;
22
23 static const uint64_t TriggerPoint = pow(2, 26);
24 static uint64_t AllocatedSize;
25
26 extern "C" void ssmlGCInit() {}
27
28 extern "C" void ssmlGCPrintStatistics() {
29     std::cout << "***** Malloc Mark Sweep GC *****\n";
30     std::cout << "*****" << std::endl;
31 }
32
33 static std::set<int64_t *> Objects;
34
35 static void markall(int64_t *Root);
36 static void sweep();
37
38 static void triggerGC(void *PrevFrame) {
39     std::cout << "***** TRIGGER GC *****" <<
40         std::endl;
41 #ifndef SSML_SHADOW_STACK_GC

```

```

41     int64_t *PrevF = (int64_t *)PrevFrame;
42     assert(isfunc(PrevF[1]));
43     FunctionFrame FirstFrame = {&PrevF[2]};
44 #endif
45
46 #ifdef SSML_SHADOW_STACK_GC
47     for (StackEntry *R = llvm_gc_root_chain; R; R = R->Next) {
48         assert(R->Map->NumMeta == 0);
49         auto Roots = R->Roots;
50         for (unsigned I = 0, E = R->Map->NumRoots; I != E; ++I) {
51             markall((int64_t *)Roots[I]);
52         }
53     }
54
55 #else // !defined(SSML_SHADOW_STACK_GC)
56     FunctionFrame Frame = FirstFrame;
57     for (;;) {
58         FrameDescr *Descr = getGCEntry(Frame.getFunction());
59         int16_t *Offsets = Descr->RootOffsets;
60
61         for (int64_t I = 0, Count = Descr->RootCount; I < Count; ++I) {
62             markall(Frame.getRoot(Offsets[I]));
63         }
64
65         if (isEntryFunction(Frame.getFunction()))
66             break;
67         Frame = Frame.nextFrame(Descr);
68     }
69 #endif // SSML_SHADOW_STACK_GC
70
71     sweep();
72 }
73
74 static int64_t ObjectNumber;
75
76 extern "C" int64_t *allocateFrame(int64_t N, void *Frame) {
77     uint64_t Size = N + sizeof(int64_t) * 2;
78     if (AllocatedSize + Size > TriggerPoint)
79         triggerGC(Frame);
80     if (AllocatedSize + Size > TriggerPoint)
81         error("out of memory");
82
83     AllocatedSize += Size;
84     int64_t *Mem = (int64_t *)calloc(1, Size);
85     Mem[0] = ++ObjectNumber;
86     Mem[1] = N / sizeof(int64_t);
87
88     Objects.insert(Mem + 2);
89
90     return (int64_t *)Mem + 2;
91 }
92
93 extern "C" int64_t *allocate(int64_t N) {
94     return allocateFrame(N, __builtin_frame_address(0));
95 }
96
97 static const int64_t ObjectMarkBit = (int64_t)1 << 63;
98 static const int64_t ObjectMarkMask = ~ObjectMarkBit;
99 static const int64_t SizeBitmask = ObjectMarkMask;
100
101 static int64_t getSize(int64_t *Obj) {
102     return Obj[-1] & SizeBitmask;

```

```

103 }
104
105 static int64_t mark(int64_t *Obj) {
106     int64_t Size = getSize(Obj);
107     Obj[-1] |= ObjectMarkBit;
108     return Size;
109 }
110
111 static void unmark(int64_t *Obj) { Obj[-1] &= ObjectMarkMask; }
112
113 static bool ismarked(int64_t *Obj) {
114     return Obj[-1] & ObjectMarkBit;
115 }
116
117 static void markall(int64_t *Root) {
118     if (!istraced(Root) || ismarked(Root))
119         return;
120
121     static int64_t *LastGood;
122     if (!Objects.count(Root))
123         std::cout << "Error object: " << Root << " PREV GOOD: " <<
124             LastGood << std::endl;
125     LastGood = Root;
126     assert(Objects.count(Root));
127
128     int64_t Size = mark(Root);
129     for (int64_t I = 0; I < Size; ++I)
130         markall((int64_t *)Root[I]);
131 }
132
133 static void sweep() {
134     for(auto It = Objects.begin(); It != Objects.end(); ) {
135         if(!ismarked(*It)) {
136             AllocatedSize -= getSize(*It) * sizeof(int64_t);
137             free(*It - 2);
138             It = Objects.erase(It);
139         } else {
140             unmark(*It);
141             ++It;
142         }
143     }
144 }
145 #endif // SSML_MARK_SWEEP_GC
146 #endif // 0

```

Listing 11: runtime/DisableGC.cpp

```

1  #ifndef SSML_DISABLE_GC
2
3  #include "GC.h"
4
5  #include <iostream>
6
7  #include <stdint.h>
8  #include <stdlib.h>
9
10 extern "C" void error(const char *);
11
12 extern "C" void ssmlGCInit() {}
13

```

```

14 static uint64_t MemoryUsage;
15
16 extern "C" int64_t *allocate(int64_t N) {
17     size_t Total = N + sizeof(int64_t);
18     MemoryUsage += Total;
19     if (MemoryUsage > MAX_ALLOC_SIZE)
20         error("out of memory");
21     int64_t *Ret = (int64_t *)calloc(1, Total);
22     if (!Ret)
23         error("out of memory");
24     return Ret;
25 }
26
27 extern "C" int64_t *allocateFrame(int64_t N, void *) {
28     return allocate(N);
29 }
30
31 extern "C" void ssmlGCPrintStatistics() {
32     uint64_t Avail = MAX_ALLOC_SIZE - MemoryUsage;
33     std::cout << "***** GC Disabled *****\n";
34     std::cout << "Memory Usage: " << MemoryUsage << "B\n";
35     std::cout << "Total Free Memory: " << Avail << "B\n";
36     std::cout << "*****" << std::endl;
37 }
38
39 #endif // SSML_DISABLE_GC

```

Listing 12: tools/driver/Driver.cpp

```

1 #define DEBUG_TYPE "main"
2
3 #include "ssml/AST/Declaration.h"
4 #include "ssml/AST/DumpVisitor.h"
5 #include "ssml/AST/Fixup/Fixup.h"
6
7 #include "ssml/Typecheck/Typecheck.h"
8 #include "ssml/Codegen/Codegen.h"
9
10 #include "llvm/Support/CommandLine.h"
11 #include "llvm/Support/Debug.h"
12 #include "llvm/Support/raw_ostream.h"
13 #include "llvm/Support/ManagedStatic.h"
14
15 using namespace llvm;
16
17 ssml::ast::Root *getParseRoot_TEMPORARY();
18 int yyparse();
19 namespace ssml {
20 namespace parse {
21 void flexFinalize();
22 } // End namespace ssml.
23 } // End namespace parse.
24
25 static void versionPrinter() {
26     errs() << "SSML:\n"
27             << "    SSML version 0.1\n";
28 }
29
30 static cl::opt<bool> DumpASTOption("dump-ast",
31                                   cl::desc("Print AST after

```

```

32                                     cl::init(false));
33 static cl::opt<bool>
34     DumpASTFixupOption("dump-ast-fixup",
35                         cl::desc("Print AST after parsing and AST
36                                 fixup"),
37                         cl::init(false));
38 static int start(int Argc, const char *const *Argv) {
39     cl::SetVersionPrinter(versionPrinter);
40     cl::ParseCommandLineOptions(Argc, Argv, "SSML program overview
41                                     description");
42     DEBUG(errs() << "TEST DEBUG\n");
43
44     int ret = yyparse();
45     if (ret)
46         return ret;
47
48     auto Root = getParseRoot_TEMPORARY();
49     ssml::ast::DumpVisitor V;
50     if (DumpASTOption) {
51         errs() << "Initial AST:\n";
52         Root->accept(&V);
53     }
54
55     ssml::parse::flexFinalize();
56
57     ssml::ast::fixup(Root);
58     if (DumpASTFixupOption) {
59         if (DumpASTOption)
60             errs() << "\n";
61         errs() << "Fixup AST:\n";
62         Root->accept(&V);
63     }
64
65     ssml::typecheck::typecheck(Root);
66     ssml::codegen::codegen(Root);
67
68     delete Root;
69     return 0;
70 }
71
72 int main(int Argc, const char *const *Argv) {
73     llvm_shutdown_obj ShutdownObj;
74     (void)ShutdownObj;
75     return start(Argc, Argv);
76 }

```

Listing 13: include/ssml/Codegen/Codegen.h

```

1  #ifndef SSML_CODEGEN_CODEGEN_H
2  #define SSML_CODEGEN_CODEGEN_H
3
4  namespace ssml {
5      namespace ast {
6          class Root;
7      } // End namespace ast.
8  } // End namespace ssml.
9
10 namespace ssml {
11     namespace codegen {
12         void codegen(ssml::ast::Root *);
13     }
14 }

```

```

13 } // End namespace codegen.
14 } // End namespace ssml.
15
16 #endif // SSML_CODEGEN_CODEGEN_H

```

Listing 14: include/ssml/Common/Memory.h

```

1 #ifndef SSML_COMMON_MEMORY_H
2 #define SSML_COMMON_MEMORY_H
3
4 #include <memory>
5
6 namespace ssml {
7 template <typename T>
8 std::shared_ptr<T> toSharedPtr(T *P) { return
    std::shared_ptr<T>(P); }
9 }
10
11 #endif // SSML_COMMON_MEMORY_H

```

Listing 15: include/ssml/Common/StringManipulation.h

```

1 #ifndef SSML_COMMON_STRINGMANIPULATION_H
2 #define SSML_COMMON_STRINGMANIPULATION_H
3
4 namespace ssml {
5 char *duplicate(const char *str);
6 } // End namespace ssml.
7
8 #endif // SSML_COMMON_STRINGMANIPULATION_H

```

Listing 16: include/ssml/Common/Timer.h

```

1 #ifndef SSML_COMMON_TIMER_H
2 #define SSML_COMMON_TIMER_H
3
4 #include <cstdint>
5 #include <sys/time.h>
6
7 namespace ssml {
8 class Timer {
9     timeval Start;
10    timeval Stop;
11
12 public:
13    void start() {
14        gettimeofday(&this->Start, nullptr);
15    }
16
17    uint64_t time() {
18        timeval Diff;
19        gettimeofday(&this->Stop, nullptr);
20        timersub(&Stop, &Start, &Diff);
21        return (uint64_t)Diff.tv_sec * 1000000 +
            (uint64_t)Diff.tv_usec;
22    }
23 };
24 } // End namespace ssml.
25

```

```
26 #endif // SSML_COMMON_TIMER_H
```

Listing 17: include/ssml/Common/Limits.h

```
1 #ifndef SSML_COMMON_LIMITS_H
2 #define SSML_COMMON_LIMITS_H
3
4 #include <cstdlib>
5
6 namespace ssml {
7     namespace limits {
8         namespace int64 {
9             constexpr int64_t min() { return -9223372036854775807; }
10            constexpr int64_t max() { return 9223372036854775807; }
11        } // End namespace int64.
12    } // End namespace limits.
13 } // End namespace ssml.
14
15 #endif // SSML_COMMON_LIMITS_H
```

Listing 18: include/ssml/Common/FatalExit.h

```
1 #ifndef SSML_COMMON_FATALEXIT_H
2 #define SSML_COMMON_FATALEXIT_H
3
4 #include "llvm/ADT/Twine.h"
5
6 namespace ssml {
7     void fatalExit(llvm::Twine Msg);
8 } // End namespace ssml.
9
10 namespace ssml {
11     void fatalExitOutOfMem();
12 } // End namespace ssml.
13
14 #endif // SSML_COMMON_FATALEXIT_H
```

Listing 19: include/ssml/Common/Compare.h

```
1 #ifndef SSML_COMMON_COMPARE_H
2 #define SSML_COMMON_COMPARE_H
3
4 #include <memory>
5
6 namespace ssml {
7     template <typename T> struct SharedPtrLess {
8         bool operator()(std::shared_ptr<const T> L,
9             std::shared_ptr<const T> R) const {
10             return *L < *R;
11         }
12     };
13 } // End namespace ssml.
14
15 namespace ssml {
16     extern template struct SharedPtrLess<std::string>;
17 } // End namespace ssml.
18
19 #endif // SSML_COMMON_COMPARE_H
```

Listing 20: include/ssml/Common/SourceLocation.h

```

1  #ifndef SSML_SOURCELOCATION_H
2  #define SSML_SOURCELOCATION_H
3
4  #include <string>
5
6  namespace llvm {
7  class raw_ostream;
8  } // End namespace llvm.
9
10 namespace ssml {
11 class SourceLocation {
12     unsigned Line;
13     unsigned Column;
14
15 public:
16     explicit SourceLocation() = default;
17     constexpr SourceLocation(unsigned Line, unsigned Col)
18         : Line(Line), Column(Col) {}
19
20     constexpr unsigned getLine() const { return this->Line; }
21     constexpr unsigned getColumn() const { return this->Column; }
22
23     std::string toString() const;
24 };
25 } // End namespace ssml
26
27 namespace llvm {
28 raw_ostream &operator<<(raw_ostream &, ssml::SourceLocation);
29 } // End namespace llvm.
30
31 #endif // SSML_SOURCELOCATION_H

```

Listing 21: include/ssml/Common/ErrorMessage.h

```

1  #ifndef SSML_COMMON_ERRORMESSAGES_H
2  #define SSML_COMMON_ERRORMESSAGES_H
3
4  #include "ssml/Common/SourceLocation.h"
5
6  namespace llvm {
7  class StringRef;
8  } // End namespace llvm.
9
10 namespace ssml {
11 void errorExit(llvm::StringRef FileName, SourceLocation Loc,
12               llvm::StringRef Message);
13 } // End namespace ssml.
14
15 namespace ssml {
16 void errorExit(SourceLocation Loc, llvm::StringRef Message);
17 } // End namespace ssml.
18
19 #endif // SSML_COMMON_ERRORMESSAGES_H

```

Listing 22: include/ssml/Common/Commandline.h

```

1  #ifndef SSML_COMMON_COMMANDLINE_H
2  #define SSML_COMMON_COMMANDLINE_H
3

```

```

4  #include <memory>
5  #include <string>
6
7  namespace ssml {
8  class CommandlineImpl {
9  public:
10     const std::shared_ptr<std::string> getFilename() const;
11 };
12 } // End namespace ssml.
13
14 namespace ssml {
15 extern CommandlineImpl Commandline;
16 }
17
18 #endif // SSML_COMMON_COMMANDLINE_H

```

Listing 23: include/ssml/AST/All.h

```

1  #ifndef SSML_AST_ALL_H
2  #define SSML_AST_ALL_H
3
4  #include "ssml/AST/Node.h"
5  #include "ssml/AST/Declaration.h"
6  #include "ssml/AST/Identifier.h"
7  #include "ssml/AST/Literal.h"
8  #include "ssml/AST/Match.h"
9  #include "ssml/AST/Pattern.h"
10 #include "ssml/AST/Expression.h"
11 #include "ssml/AST/Type.h"
12 #include "ssml/AST/Definition.h"
13
14 #endif // SSML_AST_ALL_H

```

Listing 24: include/ssml/AST/Definition.h

```

1  #ifndef SSML_AST_DEFINITION_H
2  #define SSML_AST_DEFINITION_H
3
4  #include "Node.h"
5
6  #include <memory>
7
8  namespace ssml {
9  namespace ast {
10     class LongIdentifier;
11
12     class SeqDeclaration;
13 } // End namespace ast.
14 } // End namespace ssml.
15
16 namespace ssml {
17 namespace ast {
18     class Definition : public Node {
19     protected:
20         explicit Definition(SourceLocation);
21     };
22 } // End namespace ast.
23 } // End namespace ssml.
24
25 namespace ssml {
26 namespace ast {

```

```

27 class LongIdentifierDefinition : public Definition {
28 private:
29     std::shared_ptr<LongIdentifier> ID;
30
31 public:
32     LongIdentifierDefinition(SourceLocation,
33                             std::shared_ptr<LongIdentifier> ID);
33     void accept(Visitor *) override;
34
35     std::shared_ptr<LongIdentifier> getID() { return this->ID; }
36 };
37 } // End namespace ast.
38 } // End namespace ssml.
39
40 namespace ssml {
41 namespace ast {
42 class StructDefinition : public Definition {
43 private:
44     std::shared_ptr<SeqDeclaration> Decls;
45
46 public:
47     StructDefinition(SourceLocation,
48                     std::shared_ptr<SeqDeclaration> Declarations);
49     void accept(Visitor *) override;
50
51     std::shared_ptr<SeqDeclaration> getDecl() { return this->Decl; }
52 };
53 } // End namespace ast.
54 } // End namespace ssml.
55
56 namespace ssml {
57 namespace ast {
58 class AnnotationDefinition : public Definition {
59 private:
60     std::shared_ptr<Definition> StructDef;
61     std::shared_ptr<Definition> SigDef;
62     bool Transparent;
63
64 public:
65     AnnotationDefinition(SourceLocation, std::shared_ptr<Definition>
66                         StructDef,
67                         std::shared_ptr<Definition> SigDef, bool
68                         isTransparent);
69     void accept(Visitor *) override;
70
71     std::shared_ptr<Definition> getStructDef() { return
72         this->StructDef; }
73     std::shared_ptr<Definition> getSigDef() { return this->SigDef; }
74     bool isTransparent() const { return this->Transparent; }
75 };
76 } // End namespace ast.
77 } // End namespace ssml.
78
79 namespace ssml {
80 namespace ast {
81 class ShortFunctorDefinition : public Definition {
82 private:
83     std::shared_ptr<LongIdentifier> FunctorID;
84     std::shared_ptr<Definition> StructDef;
85
86 public:

```

```

84     ShortFunctorDefinition(SourceLocation,
85                             std::shared_ptr<LongIdentifier> FunctorID,
86                             std::shared_ptr<Definition> StructDef);
87     void accept(Visitor *) override;
88
89     std::shared_ptr<LongIdentifier> getFunctorID() { return
90         this->FunctorID; }
91     std::shared_ptr<Definition> getStructDef() { return
92         this->StructDef; }
93 };
94 // End namespace ast.
95 // End namespace ssml.
96
97 namespace ssml {
98     namespace ast {
99         class LongFunctorDefinition : public Definition {
100         private:
101             std::shared_ptr<LongIdentifier> FunctorID;
102             std::shared_ptr<SeqDeclaration> Decls;
103
104         public:
105             LongFunctorDefinition(SourceLocation,
106                                   std::shared_ptr<LongIdentifier> FunctorID,
107                                   std::shared_ptr<SeqDeclaration>
108                                     Declarations);
109             void accept(Visitor *) override;
110
111             std::shared_ptr<LongIdentifier> getFunctorID() { return
112                 this->FunctorID; }
113             std::shared_ptr<SeqDeclaration> getDecl() { return this->Decl; }
114         };
115     } // End namespace ast.
116 } // End namespace ssml.
117
118 namespace ssml {
119     namespace ast {
120         class SigDefinition : public Definition {
121         private:
122             std::shared_ptr<SeqDeclaration> Decls;
123
124         public:
125             SigDefinition(SourceLocation, std::shared_ptr<SeqDeclaration>
126                           Declarations);
127             void accept(Visitor *) override;
128
129             std::shared_ptr<SeqDeclaration> getDecl() { return this->Decl; }
130         };
131     } // End namespace ast.
132 } // End namespace ssml.
133
134 #endif // SSML_AST_DEFINITION_H

```

Listing 25: include/ssml/AST/Literal.h

```

1  #ifndef SSML_AST_LITERAL_H
2  #define SSML_AST_LITERAL_H
3
4  #include "Node.h"
5

```

```

6  #include <string>
7  #include <memory>
8
9  namespace ssml {
10 namespace ast {
11 class Literal : public Node {
12     std::shared_ptr<std::string> Value;
13
14 protected:
15     explicit Literal(SourceLocation, std::shared_ptr<std::string>
16         Lit);
17 public:
18     const std::shared_ptr<std::string> getValueString() { return
19         this->Value; }
20     virtual int64_t toInt() = 0;
21 };
22 } // End namespace ast.
23 } // End namespace ssml.
24
25 namespace ssml {
26 namespace ast {
27 class IntLiteral : public Literal {
28 public:
29     using IntType = std::int64_t;
30
31 private:
32     IntType IntValue;
33
34 public:
35     explicit IntLiteral(SourceLocation, std::shared_ptr<std::string>
36         Value);
37     void accept(Visitor *) override;
38     int64_t getIntValue() const;
39     int64_t toInt() override;
40 };
41 } // End namespace ast.
42 } // End namespace ssml.
43
44 namespace ssml {
45 namespace ast {
46 class RealLiteral : public Literal {
47 public:
48     explicit RealLiteral(SourceLocation,
49         std::shared_ptr<std::string> Value);
50     void accept(Visitor *) override;
51     int64_t toInt() override;
52 };
53 } // End namespace ast.
54 } // End namespace ssml.
55
56 namespace ssml {
57 namespace ast {
58 class CharLiteral : public Literal {
59 public:
60     explicit CharLiteral(SourceLocation,
61         std::shared_ptr<std::string> Value);
62     void accept(Visitor *) override;
63     int64_t toInt() override;
64 };
65 } // End namespace ast.
66 } // End namespace ssml.

```

```

63
64 namespace ssml {
65 namespace ast {
66 class StringLiteral : public Literal {
67 public:
68     explicit StringLiteral(SourceLocation,
69                             std::shared_ptr<std::string> Value);
69     void accept(Visitor *) override;
70     int64_t toInt() override;
71 };
72 } // End namespace ast.
73 } // End namespace ssml.
74
75 #endif // SSML_AST_LITERAL_H

```

Listing 26: include/ssml/AST/Pattern.h

```

1  #ifndef SSML_AST_PATTERN_H
2  #define SSML_AST_PATTERN_H
3
4  #include "Node.h"
5
6  #include <vector>
7  #include <memory>
8
9  namespace ssml {
10 namespace ast {
11 class Literal;
12 class Identifier;
13 class ShortIdentifier;
14 class LongIdentifier;
15 class Type;
16 class TypePattern;
17 class LongIdentifierPattern;
18 class ApplyPattern;
19 } // End namespace ast.
20 } // End namespace ssml.
21
22 namespace ssml {
23 namespace ast {
24 class Pattern : public Node {
25 protected:
26     explicit Pattern(SourceLocation);
27     bool HasSimpleType = false;
28 public:
29     virtual TypePattern *asTypePattern();
30     virtual LongIdentifierPattern *asLongIdentifierPattern();
31     virtual ApplyPattern *asApplyPattern();
32
33     bool hasSimpleType() { return this->HasSimpleType; }
34     void setHasSimpleType(bool B = true) { this->HasSimpleType = B; }
35 };
36 } // End namespace ast.
37 } // End namespace ssml.
38
39 namespace ssml {
40 namespace ast {
41 class SeqPattern : public Pattern,
42                   public std::vector<std::shared_ptr<Pattern>> {
43 public:
44     explicit SeqPattern(SourceLocation);

```

```

45     explicit SeqPattern(SourceLocation, std::shared_ptr<Pattern>
46         First);
47     void accept(Visitor *V) override;
48 };
49 } // End namespace ast.
50 } // End namespace ssml.
51
52 namespace ssml {
53     namespace ast {
54         class LiteralPattern : public Pattern {
55             std::shared_ptr<Literal> Value;
56         public:
57             explicit LiteralPattern(SourceLocation,
58                 std::shared_ptr<Literal>);
59             void accept(Visitor *) override;
60             std::shared_ptr<Literal> getValue() { return this->Value; }
61         };
62     } // End namespace ast.
63 } // End namespace ssml.
64
65 namespace ssml {
66     namespace ast {
67         class WildcardPattern : public Pattern {
68         public:
69             explicit WildcardPattern(SourceLocation);
70             void accept(Visitor *) override;
71         };
72     } // End namespace ast.
73 } // End namespace ssml.
74
75 namespace ssml {
76     namespace ast {
77         class LongIdentifierPattern : public Pattern {
78         private:
79             std::shared_ptr<LongIdentifier> ID;
80             bool opKeyPrefixed;
81         public:
82             explicit LongIdentifierPattern(SourceLocation,
83                 std::shared_ptr<LongIdentifier>
84                     ID,
85                     bool isPrefixedWithOpKey = false);
86             void accept(Visitor *) override;
87             bool isPrefixedWithOpKey() const { return this->opKeyPrefixed; }
88             std::shared_ptr<LongIdentifier> getIDs() { return this->ID; }
89             LongIdentifierPattern *asLongIdentifierPattern() override;
90         };
91     } // End namespace ast.
92 } // End namespace ssml.
93
94 namespace ssml {
95     namespace ast {
96         class TypePattern : public Pattern {
97         private:
98             std::shared_ptr<Pattern> Pat;
99             std::shared_ptr<Type> Typ;
100         public:
101             explicit TypePattern(SourceLocation, std::shared_ptr<Pattern> P,
102                 std::shared_ptr<Type> T);
103             void accept(Visitor *) override;

```

```

104     std::shared_ptr<Pattern> getPattern() { return this->Pat; }
105     std::shared_ptr<Type> getType() { return this->Typ; }
106     TypePattern *asTypePattern() override;
107 };
108 } // End namespace ast.
109 } // End namespace ssml.
110
111 namespace ssml {
112 namespace ast {
113 class AsPattern : public Pattern {
114 private:
115     std::shared_ptr<Pattern> LeftPattern;
116     std::shared_ptr<Pattern> RightPattern;
117
118 public:
119     explicit AsPattern(SourceLocation, std::shared_ptr<Pattern> LHS,
120                        std::shared_ptr<Pattern> RHS);
121     void accept(Visitor *) override;
122     std::shared_ptr<Pattern> getLeftPattern() { return
123         this->LeftPattern; }
124     std::shared_ptr<Pattern> getRightPattern() { return
125         this->RightPattern; }
126 };
127 } // End namespace ast.
128 } // End namespace ssml.
129
130 namespace ssml {
131 namespace ast {
132 class ApplyPattern : public Pattern,
133                    public std::vector<std::shared_ptr<Pattern>> {
134 public:
135     explicit ApplyPattern(SourceLocation, std::shared_ptr<Pattern>
136                          First);
137     void accept(Visitor *V) override;
138     ApplyPattern *asApplyPattern() override;
139 };
140 } // End namespace ast.
141 } // End namespace ssml.
142
143 namespace ssml {
144 namespace ast {
145 class IdentifierEqualsPattern : public Pattern {
146 private:
147     std::shared_ptr<Identifier> Iden;
148     std::shared_ptr<Pattern> Pat;
149
150 public:
151     explicit IdentifierEqualsPattern(SourceLocation,
152                                     std::shared_ptr<Identifier> LHS,
153                                     std::shared_ptr<Pattern> RHS);
154     void accept(Visitor *V) override;
155     std::shared_ptr<Identifier> getID() { return this->Iden; }
156     std::shared_ptr<Pattern> getPattern() { return this->Pat; }
157 };
158 } // End namespace ast.
159 } // End namespace ssml.
160
161 namespace ssml {
162 namespace ast {
163 class ShortIdentifierPattern : public Pattern {
164 private:
165     std::shared_ptr<ShortIdentifier> ID;

```

```

163
164 public:
165     explicit ShortIdentifierPattern(SourceLocation,
166                                     std::shared_ptr<ShortIdentifier>
167                                     ID);
168
169     void accept(Visitor *) override;
170     std::shared_ptr<ShortIdentifier> getID() { return this->ID; }
171 };
172 // End namespace ast.
173 // End namespace ssml.
174
175 namespace ssml {
176     namespace ast {
177         class RecordPattern : public Pattern,
178                               public std::vector<std::shared_ptr<Pattern>>
179                               {
180 private:
181     bool EllipsisTerminated = false;
182
183 public:
184     explicit RecordPattern(SourceLocation, std::shared_ptr<Pattern>
185                             First);
186     void accept(Visitor *V) override;
187     bool isEllipsisTerminated() const { return
188         this->EllipsisTerminated; }
189     void makeEllipsisTerminated() { EllipsisTerminated = true; }
190 };
191 // End namespace ast.
192 // End namespace ssml.
193
194 namespace ssml {
195     namespace ast {
196         class ListPattern : public Pattern {
197 private:
198     std::shared_ptr<SeqPattern> Patterns;
199
200 public:
201     explicit ListPattern(SourceLocation, std::shared_ptr<SeqPattern>
202                             Patterns);
203     void accept(Visitor *V) override;
204     std::shared_ptr<SeqPattern> getPatterns() { return
205         this->Patterns; }
206 };
207 // End namespace ast.
208 // End namespace ssml.
209
210 namespace ssml {
211     namespace ast {
212         class TuplePattern : public Pattern {
213 private:
214     std::shared_ptr<SeqPattern> Patterns;
215
216 public:
217     explicit TuplePattern(SourceLocation,
218                             std::shared_ptr<SeqPattern> Patterns);
219     void accept(Visitor *V) override;
220     std::shared_ptr<SeqPattern> getPatterns() { return
221         this->Patterns; }
222 };
223 // End namespace ast.
224 // End namespace ssml.
225
226

```

```
217 #endif // SSML_AST_PATTERN_H
```

Listing 27: include/ssml/AST/Match.h

```
1 #ifndef SSML_AST_MATCH_H
2 #define SSML_AST_MATCH_H
3
4 #include "Node.h"
5
6 #include <vector>
7 #include <memory>
8
9 namespace ssml {
10 namespace ast {
11 class Pattern;
12 class Expression;
13 } // End namespace ast.
14 } // End namespace ssml.
15
16 namespace ssml {
17 namespace ast {
18 class Match : public Node {
19 private:
20     std::shared_ptr<Pattern> Pat;
21     std::shared_ptr<Expression> Expr;
22
23 public:
24     Match(SourceLocation, std::shared_ptr<Pattern> P,
25           std::shared_ptr<Expression> E);
26     void accept(Visitor *V) override;
27     std::shared_ptr<Pattern> getPattern() { return this->Pat; }
28     std::shared_ptr<Expression> getExpr() { return this->Expr; }
29 };
30 } // End namespace ast.
31 } // End namespace ssml.
32
33 namespace ssml {
34 namespace ast {
35 class SeqMatch : public Node, public
36                 std::vector<std::shared_ptr<Match>> {
37 public:
38     SeqMatch(SourceLocation, std::shared_ptr<Match> First);
39     void accept(Visitor *V) override;
40 };
41 } // End namespace ast.
42 } // End namespace ssml.
43 #endif // SSML_AST_MATCH_H
```

Listing 28: include/ssml/AST/Fixup/Fixup.h

```
1 #ifndef SSML_AST_PASS_FIXUP_H
2 #define SSML_AST_PASS_FIXUP_H
3
4 namespace ssml {
5 namespace ast {
6 class Root;
7 } // End namespace ast.
8 } // End namespace ssml.
9
10 namespace ssml {
```

```

11 namespace ast {
12 void fixup(Root *);
13 } // End namespace ast.
14 } // End namespace ssml.
15
16 #endif // SSML_AST_PASS_FIXUP_H

```

Listing 29: include/ssml/AST/FunctionVals.h

```

1 #ifndef SSML_AST_FUNCTIONVALS_H
2 #define SSML_AST_FUNCTIONVALS_H
3
4 #include <vector>
5
6 namespace ssml {
7 namespace ast {
8 class FunctionVals {
9 private:
10     std::vector<bool> Table;
11
12 public:
13     void addVal(bool IsPointer) {
14         this->Table.push_back(IsPointer);
15     }
16     std::vector<bool> &getFlags() { return this->Table; }
17 };
18 } // End namespace ast.
19 } // End namespace ssml.
20
21 #endif // SSML_AST_FUNCTIONVALS_H

```

Listing 30: include/ssml/AST/Expression.h

```

1 #ifndef SSML_AST_EXPRESSION_H
2 #define SSML_AST_EXPRESSION_H
3
4 #include "Literal.h"
5 #include "FunctionVals.h"
6
7 #include <vector>
8
9 namespace ssml {
10 namespace ast {
11 class SeqMatch;
12 class Type;
13 class SeqDeclaration;
14 class Identifier;
15 class LongIdentifier;
16 class LongIdentifierExpression;
17 class ApplyExpression;
18 class TupleExpression;
19 } // End namespace ast.
20 } // End namespace ssml.
21
22 namespace ssml {
23 namespace ast {
24 class Expression : public Node {
25 protected:
26     explicit Expression(SourceLocation);
27 public:
28     virtual TupleExpression *asTupleExpression();

```

```

29     virtual LongIdentifierExpression *asLongIdentifierExpression();
30 };
31 } // End namespace ast.
32 } // End namespace ssml.
33
34 namespace ssml {
35     namespace ast {
36         class SeqExpression : public Expression,
37                               public
38                               {
39             std::vector<std::shared_ptr<Expression>>
40             {
41 public:
42             explicit SeqExpression(SourceLocation);
43             explicit SeqExpression(SourceLocation,
44                                   std::shared_ptr<Expression> First);
45             void accept(Visitor *V) override;
46         };
47     } // End namespace ast.
48 } // End namespace ssml.
49
50 namespace ssml {
51     namespace ast {
52         class LiteralExpression : public Expression {
53             std::shared_ptr<Literal> Value;
54 public:
55             explicit LiteralExpression(SourceLocation,
56                                       std::shared_ptr<Literal> Lit);
57             void accept(Visitor *) override;
58             std::shared_ptr<Literal> getValue() { return this->Value; }
59         };
60     } // End namespace ast.
61 } // End namespace ssml.
62
63 namespace ssml {
64     namespace ast {
65         class LongIdentifierExpression : public Expression {
66 private:
67             std::shared_ptr<LongIdentifier> ID;
68             bool opKeyPrefixed;
69 public:
70             explicit LongIdentifierExpression(SourceLocation,
71                                               std::shared_ptr<LongIdentifier>
72                                               ID,
73                                               bool isPrefixedWithOpKey =
74                                               false);
75             void accept(Visitor *) override;
76             bool isPrefixedWithOpKey() const { return this->opKeyPrefixed; }
77             std::shared_ptr<LongIdentifier> getIDs() { return this->ID; }
78             LongIdentifierExpression *asLongIdentifierExpression() override;
79         };
80     } // End namespace ast.
81 } // End namespace ssml.
82
83 namespace ssml {
84     namespace ast {
85         class TupleExpression : public Expression {
86             std::shared_ptr<SeqExpression> Expressions;
87 public:
88             explicit TupleExpression(SourceLocation,

```

```

85         std::shared_ptr<SeqExpression> Exprs);
86     void accept(Visitor *) override;
87     std::shared_ptr<SeqExpression> getExprs() { return
88         this->Expressions; }
89     TupleExpression *asTupleExpression() override;
90 };
91 // End namespace ast.
92 // End namespace ssml.
93 namespace ssml {
94     namespace ast {
95         class ListExpression : public Expression {
96             std::shared_ptr<SeqExpression> Expressions;
97         public:
98             explicit ListExpression(SourceLocation,
99                 std::shared_ptr<SeqExpression> Exprs);
100             void accept(Visitor *) override;
101             std::shared_ptr<SeqExpression> getExprs() { return
102                 this->Expressions; }
103         };
104     } // End namespace ast.
105 } // End namespace ssml.
106 namespace ssml {
107     namespace ast {
108         class IdentifierEqualsExpression : public Expression {
109         private:
110             std::shared_ptr<Identifier> Iden;
111             std::shared_ptr<Expression> Expr;
112         public:
113             explicit IdentifierEqualsExpression(SourceLocation,
114                 std::shared_ptr<Identifier>
115                     LHS,
116                 std::shared_ptr<Expression>
117                     RHS);
118             void accept(Visitor *V) override;
119             std::shared_ptr<Identifier> getID() { return this->Iden; }
120             std::shared_ptr<Expression> getExpr() { return this->Expr; }
121         };
122     } // End namespace ast.
123 } // End namespace ssml.
124 namespace ssml {
125     namespace ast {
126         class RecordExpression
127             : public Expression,
128             public
129                 std::vector<std::shared_ptr<IdentifierEqualsExpression>>
130                 {
131         public:
132             explicit RecordExpression(SourceLocation,
133                 std::shared_ptr<IdentifierEqualsExpression>
134                     First);
135             void accept(Visitor *V) override;
136         };
137     } // End namespace ast.
138 } // End namespace ssml.
139 namespace ssml {
140     namespace ast {

```

```

139 class SelectorExpression : public Expression {
140 private:
141     std::shared_ptr<Identifier> ID;
142
143 public:
144     explicit SelectorExpression(SourceLocation,
145                               std::shared_ptr<Identifier> ID);
145     void accept(Visitor *V) override;
146     std::shared_ptr<Identifier> getID() { return this->ID; }
147 };
148 } // End namespace ast.
149 } // End namespace ssml.
150
151 namespace ssml {
152 namespace ast {
153 class ApplyExpression : public Expression,
154                        public
155                        std::vector<std::shared_ptr<Expression>>
156                        {
157 public:
158     explicit ApplyExpression(SourceLocation,
159                             std::shared_ptr<Expression> First);
159     void accept(Visitor *V) override;
160 };
161 } // End namespace ast.
162 } // End namespace ssml.
163
164 namespace ssml {
165 namespace ast {
166 class TypeExpression : public Expression {
167 private:
168     std::shared_ptr<Expression> Expr;
169     std::shared_ptr<Type> Typ;
170
171 public:
172     explicit TypeExpression(SourceLocation,
173                             std::shared_ptr<Expression> E,
174                             std::shared_ptr<Type> T);
175     void accept(Visitor *V) override;
176     std::shared_ptr<Expression> getExpr() { return this->Expr; }
177     std::shared_ptr<Type> getType() { return this->Typ; }
178 };
179 } // End namespace ast.
180 } // End namespace ssml.
181
182 namespace ssml {
183 namespace ast {
184 class OrElseExpression : public Expression {
185 private:
186     std::shared_ptr<Expression> Left;
187     std::shared_ptr<Expression> Right;
188
189 public:
190     explicit OrElseExpression(SourceLocation,
191                               std::shared_ptr<Expression> LHS,
192                               std::shared_ptr<Expression> RHS);
193     void accept(Visitor *V) override;
194     std::shared_ptr<Expression> getLeftExpr() { return this->Left; }
195     std::shared_ptr<Expression> getRightExpr() { return this->Right; }
196 };
197 } // End namespace ast.

```

```

194 } // End namespace ssml.
195
196 namespace ssml {
197 namespace ast {
198 class AndAlsoExpression : public Expression {
199 private:
200     std::shared_ptr<Expression> Left;
201     std::shared_ptr<Expression> Right;
202
203 public:
204     explicit AndAlsoExpression(SourceLocation,
205                               std::shared_ptr<Expression> LHS,
206                               std::shared_ptr<Expression> RHS);
207     void accept(Visitor *V) override;
208     std::shared_ptr<Expression> getLeftExpr() { return this->Left; }
209     std::shared_ptr<Expression> getRightExpr() { return this->Right; }
210 };
211 } // End namespace ast.
212 } // End namespace ssml.
213
214 namespace ssml {
215 namespace ast {
216 class LetExpression : public Expression {
217 private:
218     std::shared_ptr<SeqDeclaration> Declarations;
219     std::shared_ptr<SeqExpression> Expressions;
220
221 public:
222     explicit LetExpression(SourceLocation,
223                           std::shared_ptr<SeqDeclaration> LHS,
224                           std::shared_ptr<SeqExpression> RHS);
225     void accept(Visitor *V) override;
226
227     std::shared_ptr<SeqDeclaration> getDecls() {
228         return this->Declarations;
229     }
230
231     std::shared_ptr<SeqExpression> getExprs() { return
232         this->Expressions; }
233 };
234 } // End namespace ast.
235 } // End namespace ssml.
236
237 namespace ssml {
238 namespace ast {
239 class IfExpression : public Expression {
240 private:
241     std::shared_ptr<Expression> CondExpr;
242     std::shared_ptr<Expression> ThenExpr;
243     std::shared_ptr<Expression> ElseExpr;
244
245 public:
246     explicit IfExpression(SourceLocation,
247                           std::shared_ptr<Expression> Cond,
248                           std::shared_ptr<Expression> Then,
249                           std::shared_ptr<Expression> Else);
250     void accept(Visitor *V) override;
251     std::shared_ptr<Expression> getCondExpr() { return
252         this->CondExpr; }
253     std::shared_ptr<Expression> getThenExpr() { return
254         this->ThenExpr; }

```

```

249     std::shared_ptr<Expression> getElseExpr() { return
        this->ElseExpr; }
250 };
251 } // End namespace ast.
252 } // End namespace ssml.
253
254 namespace ssml {
255     namespace ast {
256         class WhileExpression : public Expression {
257         private:
258             std::shared_ptr<Expression> CondExpr;
259             std::shared_ptr<Expression> BodyExpr;
260
261         public:
262             explicit WhileExpression(SourceLocation,
                std::shared_ptr<Expression> Cond,
263                 std::shared_ptr<Expression> Body);
264             void accept(Visitor *V) override;
265             std::shared_ptr<Expression> getCondExpr() { return
                this->CondExpr; }
266             std::shared_ptr<Expression> getBodyExpr() { return
                this->BodyExpr; }
267         };
268     } // End namespace ast.
269 } // End namespace ssml.
270
271 namespace ssml {
272     namespace ast {
273         class CaseExpression : public Expression {
274         private:
275             std::shared_ptr<Expression> Expr;
276             std::shared_ptr<SeqMatch> Cases;
277
278         public:
279             explicit CaseExpression(SourceLocation,
                std::shared_ptr<Expression> Expr,
280                 std::shared_ptr<SeqMatch> Cases);
281             void accept(Visitor *V) override;
282             std::shared_ptr<Expression> getExpr() { return this->Expr; }
283             std::shared_ptr<SeqMatch> getCases() { return this->Cases; }
284         };
285     } // End namespace ast.
286 } // End namespace ssml.
287
288 namespace ssml {
289     namespace ast {
290         class LambdaExpression : public Expression {
291         private:
292             std::shared_ptr<SeqMatch> Cases;
293             std::shared_ptr<FunctionVals> FunVals;
294
295         public:
296             explicit LambdaExpression(SourceLocation,
                std::shared_ptr<SeqMatch> Cases);
297             void accept(Visitor *V) override;
298             std::shared_ptr<SeqMatch> getCases() { return this->Cases; }
299
300             void setFunctionVals(std::shared_ptr<FunctionVals> V) {
                this->FunVals = V; }
301             std::shared_ptr<FunctionVals> getFunctionVals() { return
                this->FunVals; }
302     };

```

```

303 } // End namespace ast.
304 } // End namespace ssml.
305
306 #endif // SSML_AST_EXPRESSION_H

```

Listing 31: include/ssml/AST/Type.h

```

1  #ifndef SSML_AST_TYPE_H
2  #define SSML_AST_TYPE_H
3
4  #include "Node.h"
5
6  #include <vector>
7  #include <memory>
8
9  namespace ssml {
10 namespace ast {
11 class ApplyType;
12 class LongIdentifier;
13 class ShortIdentifier;
14 class Identifier;
15 class TupleType;
16 } // End namespace ast.
17 } // End namespace ssml.
18
19 namespace ssml {
20 namespace ast {
21 class Type : public Node {
22 protected:
23     explicit Type(SourceLocation);
24
25 public:
26     virtual TupleType *asTupleType();
27 };
28 } // End namespace ast.
29 } // End namespace ssml.
30
31 namespace ssml {
32 namespace ast {
33 class LongIdentifierType : public Type {
34 private:
35     std::shared_ptr<LongIdentifier> ID;
36
37 public:
38     explicit LongIdentifierType(SourceLocation,
39                               std::shared_ptr<LongIdentifier> ID);
40     void accept(Visitor *) override;
41     std::shared_ptr<LongIdentifier> getIDs() { return this->ID; }
42 };
43 } // End namespace ast.
44 } // End namespace ssml.
45
46 namespace ssml {
47 namespace ast {
48 class VariableType : public Type {
49 private:
50     std::shared_ptr<ShortIdentifier> ID;
51
52 public:
53     explicit VariableType(SourceLocation,
54                           std::shared_ptr<ShortIdentifier> ID);

```

```

54     void accept(Visitor *) override;
55     std::shared_ptr<ShortIdentifier> getID() { return this->ID; }
56 };
57 } // End namespace ast.
58 } // End namespace ssml.
59
60 namespace ssml {
61     namespace ast {
62     class SeqType : public Type, public
63         std::vector<std::shared_ptr<Type>> {
64     public:
65         explicit SeqType(SourceLocation, std::shared_ptr<Type> First);
66         void accept(Visitor *) override;
67     };
68     } // End namespace ast.
69 } // End namespace ssml.
70
71 namespace ssml {
72     namespace ast {
73     class SeqVariableType : public Type,
74                             public
75                             std::vector<std::shared_ptr<VariableType>>
76                             {
77     public:
78         explicit SeqVariableType(SourceLocation,
79                                 std::shared_ptr<VariableType> First);
80         void accept(Visitor *) override;
81     };
82     } // End namespace ast.
83 } // End namespace ssml.
84
85 namespace ssml {
86     namespace ast {
87     class TupleType : public Type {
88     private:
89         std::shared_ptr<SeqType> Types;
90     public:
91         explicit TupleType(SourceLocation, std::shared_ptr<SeqType>
92                             Types);
93         void accept(Visitor *) override;
94         std::shared_ptr<SeqType> getTypes() { return this->Types; }
95         TupleType *asTupleType() override;
96     };
97     } // End namespace ast.
98 } // End namespace ssml.
99
100 namespace ssml {
101     namespace ast {
102     class IdentifierColonType : public Type {
103     private:
104         std::shared_ptr<Identifier> Iden;
105         std::shared_ptr<Type> Typ;
106     public:
107         explicit IdentifierColonType(SourceLocation,
108                                     std::shared_ptr<Identifier> ID,
109                                     std::shared_ptr<Type> T);
110         void accept(Visitor *) override;
111         std::shared_ptr<Identifier> getIDs() { return this->Iden; }
112         std::shared_ptr<Type> getType() { return this->Typ; }
113     };
114     } // End namespace ast.
115 } // End namespace ssml.

```

```

110 } // End namespace ast.
111 } // End namespace ssml.
112
113 namespace ssml {
114     namespace ast {
115         class RecordType : public Type,
116                             public
                                std::vector<std::shared_ptr<IdentifierColonType>>
                                {
117     public:
118         explicit RecordType(SourceLocation,
119                             std::shared_ptr<IdentifierColonType> First);
120         void accept(Visitor *) override;
121     };
122 } // End namespace ast.
123 } // End namespace ssml.
124
125 namespace ssml {
126     namespace ast {
127         class ApplyType : public Type, public
                                std::vector<std::shared_ptr<Type>> {
128     public:
129         explicit ApplyType(SourceLocation, std::shared_ptr<Type> First);
130         void accept(Visitor *) override;
131     };
132 } // End namespace ast.
133 } // End namespace ssml.
134
135 namespace ssml {
136     namespace ast {
137         class ProductType : public Type, public
                                std::vector<std::shared_ptr<Type>> {
138     public:
139         explicit ProductType(SourceLocation, std::shared_ptr<Type>
                                First);
140         void accept(Visitor *) override;
141     };
142 } // End namespace ast.
143 } // End namespace ssml.
144
145 namespace ssml {
146     namespace ast {
147         class FunctionType : public Type, public
                                std::vector<std::shared_ptr<Type>> {
148     public:
149         explicit FunctionType(SourceLocation, std::shared_ptr<Type>
                                First);
150         void accept(Visitor *) override;
151     };
152 } // End namespace ast.
153 } // End namespace ssml.
154
155 #endif // SSML_AST_TYPE_H

```

Listing 32: include/ssml/AST/Declaration.h

```

1 #ifndef SSML_AST_DECLARATION_H
2 #define SSML_AST_DECLARATION_H
3
4 #include "Node.h"
5 #include "FunctionVals.h"

```

```

6
7 #include <vector>
8 #include <memory>
9
10 namespace ssml {
11 namespace ast {
12 class IntLiteral;
13 class ShortIdentifier;
14 class LongIdentifier;
15 class SeqLongIdentifier;
16 class SeqShortIdentifier;
17 class Pattern;
18 class Expression;
19 class Definition;
20 class Type;
21 class SeqVariableType;
22 } // End namespace ast.
23 } // End namespace ssml.
24
25 namespace ssml {
26 namespace ast {
27 class Declaration : public Node {
28 protected:
29     explicit Declaration(SourceLocation);
30 };
31 } // End namespace ast.
32 } // End namespace ssml.
33
34 namespace ssml {
35 namespace ast {
36 class SeqDeclaration : public Declaration,
37                        public
38                        std::vector<std::shared_ptr<Declaration>>
39                        {
38 public:
39     explicit SeqDeclaration(SourceLocation);
40     explicit SeqDeclaration(SourceLocation,
41                             std::shared_ptr<Declaration> First);
41     void accept(Visitor *) override;
42 };
43 } // End namespace ast.
44 } // End namespace ssml.
45
46 namespace ssml {
47 namespace ast {
48 class Root : public SeqDeclaration {
49 private:
50     std::shared_ptr<FunctionVals> MainVals;
51
52 public:
53     explicit Root(SourceLocation);
54     explicit Root(SourceLocation, std::shared_ptr<Declaration>
55                     First);
55     void accept(Visitor *) override;
56
57     void setFunctionVals(std::shared_ptr<FunctionVals> V) {
58         this->MainVals = V; }
58     std::shared_ptr<FunctionVals> getFunctionVals() { return
59         this->MainVals; }
59 };
60 } // End namespace ast.
61 } // End namespace ssml.

```

```

62
63 namespace ssml {
64 namespace ast {
65 class ValDeclaration : public Declaration {
66     std::shared_ptr<Pattern> Dest;
67     std::shared_ptr<Expression> Source;
68
69 public:
70     explicit ValDeclaration(SourceLocation L,
71                             std::shared_ptr<Pattern> Dest,
72                             std::shared_ptr<Expression> Src);
73     void accept(Visitor *) override;
74     std::shared_ptr<Pattern> getDest() { return Dest; }
75     std::shared_ptr<Expression> getSource() { return Source; }
76 };
77 } // End namespace ast.
78 } // End namespace ssml.
79
80 namespace ssml {
81 namespace ast {
82 class OpenDeclaration : public Declaration {
83 private:
84     std::shared_ptr<SeqLongIdentifier> IDs;
85
86 public:
87     explicit OpenDeclaration(SourceLocation L,
88                             std::shared_ptr<SeqLongIdentifier> IDs);
89     void accept(Visitor *) override;
90     std::shared_ptr<SeqLongIdentifier> getIDs() { return this->IDs; }
91 };
92 } // End namespace ast.
93 } // End namespace ssml.
94
95 namespace ssml {
96 namespace ast {
97 class NonfixDeclaration : public Declaration {
98 private:
99     std::shared_ptr<SeqShortIdentifier> IDs;
100
101 public:
102     explicit NonfixDeclaration(SourceLocation L,
103                               std::shared_ptr<SeqShortIdentifier>
104                               IDs);
105     void accept(Visitor *) override;
106     std::shared_ptr<SeqShortIdentifier> getIDs() { return this->IDs; }
107 };
108 } // End namespace ast.
109 } // End namespace ssml.
110
111 namespace ssml {
112 namespace ast {
113 class InfixDeclaration : public Declaration {
114 private:
115     std::shared_ptr<IntLiteral> Precedence;
116     std::shared_ptr<SeqShortIdentifier> Idens;
117
118 public:
119     explicit InfixDeclaration(SourceLocation L,
120                               std::shared_ptr<IntLiteral> Precedence,
121                               std::shared_ptr<SeqShortIdentifier>
122                               IDs);

```

```

120     void accept(Visitor *) override;
121     std::shared_ptr<IntLiteral> getPrecedence() { return
        this->Precedence; }
122     std::shared_ptr<SeqShortIdentifier> getIDs() { return
        this->Idens; }
123 };
124 } // End namespace ast.
125 } // End namespace ssml.
126
127 namespace ssml {
128     namespace ast {
129         class InfixRDeclaration : public Declaration {
130         private:
131             std::shared_ptr<IntLiteral> Precedence;
132             std::shared_ptr<SeqShortIdentifier> Idens;
133         public:
134             explicit InfixRDeclaration(SourceLocation L,
135                                     std::shared_ptr<IntLiteral>
136                                     Precedence,
137                                     std::shared_ptr<SeqShortIdentifier>
138                                     IDs);
139             void accept(Visitor *) override;
140             std::shared_ptr<IntLiteral> getPrecedence() { return
                this->Precedence; }
141             std::shared_ptr<SeqShortIdentifier> getIDs() { return
                this->Idens; }
142         };
143     } // End namespace ast.
144 } // End namespace ssml.
145
146 namespace ssml {
147     namespace ast {
148         class RestrictingSigDefinition {
149         private:
150             std::shared_ptr<Definition> Def;
151             bool IsTransparent;
152         public:
153             explicit RestrictingSigDefinition(std::shared_ptr<Definition>
154                                     SigDef,
155                                     bool isTransparent);
156             std::shared_ptr<Definition> getSigDef() { return this->Def; }
157             bool isTransparent() const { return this->IsTransparent; }
158             bool isOpaque() const { return !this->isTransparent(); }
159         };
160     } // End namespace ast.
161 } // End namespace ssml.
162
163 namespace ssml {
164     namespace ast {
165         class BareStructDeclaration : public Declaration {
166         private:
167             std::shared_ptr<ShortIdentifier> ID;
168             std::shared_ptr<Definition> StructDef;
169         public:
170             explicit BareStructDeclaration(SourceLocation,
171                                     std::shared_ptr<ShortIdentifier>
172                                     ID,
173                                     std::shared_ptr<Definition>
174                                     StructDef);

```

```

173     void accept(Visitor *) override;
174     std::shared_ptr<ShortIdentifier> getID() { return this->ID; }
175     std::shared_ptr<Definition> getStructDef() { return
        this->StructDef; }
176 };
177 } // End namespace ast.
178 } // End namespace ssml.
179
180 namespace ssml {
181     namespace ast {
182         class RestrictedStructDeclaration : public BareStructDeclaration {
183         private:
184             RestrictingSigDefinition SigDef;
185
186         public:
187             explicit RestrictedStructDeclaration(SourceLocation,
188                                                 std::shared_ptr<ShortIdentifier>
189                                                 ID,
190                                                 RestrictingSigDefinition
191                                                 SigDef,
192                                                 std::shared_ptr<Definition>
193                                                 StructDef);
194
195             void accept(Visitor *) override;
196
197             std::shared_ptr<Definition> getSigDef() { return
198                 this->SigDef.getSigDef(); }
199             bool isSigTransparent() const { return
200                 this->SigDef.isTransparent(); }
201         };
202     } // End namespace ast.
203 } // End namespace ssml.
204
205 namespace ssml {
206     namespace ast {
207         class AbstractValDeclaration : public Declaration {
208         private:
209             std::shared_ptr<ShortIdentifier> ID;
210             std::shared_ptr<Type> Typ;
211
212         public:
213             explicit AbstractValDeclaration(SourceLocation,
214                                             std::shared_ptr<ShortIdentifier>
215                                             ID,
216                                             std::shared_ptr<Type> T);
217
218             void accept(Visitor *) override;
219             std::shared_ptr<ShortIdentifier> getID() { return this->ID; }
220             std::shared_ptr<Type> getType() { return this->Typ; }
221         };
222     } // End namespace ast.
223 } // End namespace ssml.
224
225 namespace ssml {
226     namespace ast {
227         class AbstractTypeDeclaration : public Declaration {
228         private:
229             std::shared_ptr<ShortIdentifier> ID;
230
231         public:
232             explicit AbstractTypeDeclaration(SourceLocation,
233                                             std::shared_ptr<ShortIdentifier>
234                                             ID);
235
236             void accept(Visitor *) override;

```

```

227     std::shared_ptr<ShortIdentifier> getID() { return this->ID; }
228 };
229 } // End namespace ast.
230 } // End namespace ssml.
231
232 namespace ssml {
233     namespace ast {
234         class AbstractStructDeclaration : public Declaration {
235         private:
236             std::shared_ptr<ShortIdentifier> ID;
237             std::shared_ptr<Definition> SigDef;
238
239         public:
240             explicit AbstractStructDeclaration(SourceLocation,
241                 std::shared_ptr<ShortIdentifier>
242                     ID,
243                     std::shared_ptr<Definition>
244                         SigDef);
245
246             void accept(Visitor *) override;
247             std::shared_ptr<ShortIdentifier> getID() { return this->ID; }
248             std::shared_ptr<Definition> getSigDef() { return this->SigDef; }
249         };
250     } // End namespace ast.
251 } // End namespace ssml.
252
253 namespace ssml {
254     namespace ast {
255         class SharingTypeDeclaration
256             : public Declaration,
257             public std::vector<std::shared_ptr<LongIdentifier>> {
258         public:
259             explicit SharingTypeDeclaration(SourceLocation,
260                 std::shared_ptr<LongIdentifier>
261                     First);
262
263             void accept(Visitor *) override;
264         };
265     } // End namespace ast.
266 } // End namespace ssml.
267
268 namespace ssml {
269     namespace ast {
270         class SigDeclaration : public Declaration {
271         private:
272             std::shared_ptr<ShortIdentifier> ID;
273             std::shared_ptr<Definition> Def;
274
275         public:
276             explicit SigDeclaration(SourceLocation,
277                 std::shared_ptr<ShortIdentifier> ID,
278                 std::shared_ptr<Definition> Def);
279
280             void accept(Visitor *) override;
281             std::shared_ptr<ShortIdentifier> getID() { return this->ID; }
282             std::shared_ptr<Definition> getSigDef() { return this->Def; }
283         };
284     } // End namespace ast.
285 } // End namespace ssml.
286
287 namespace ssml {
288     namespace ast {
289         class FunPatternDeclaration : public Declaration {
290         private:
291             std::shared_ptr<Pattern> IDParams;

```

```

285     std::shared_ptr<Expression> Def;
286
287 public:
288     explicit FunPatternDeclaration(SourceLocation,
289                                   std::shared_ptr<Pattern>
290                                   IDParameters,
291                                   std::shared_ptr<Expression>
292                                   FunDef);
293
294     void accept(Visitor *) override;
295     std::shared_ptr<Pattern> getIDParams() { return this->IDParams; }
296     std::shared_ptr<Expression> getDef() { return this->Def; }
297 };
298 // End namespace ast.
299 // End namespace ssml.
300
301 namespace ssml {
302 namespace ast {
303 class FunDeclaration
304     : public Declaration,
305     public std::vector<std::shared_ptr<FunPatternDeclaration>> {
306 private:
307     std::shared_ptr<FunctionVals> FunVals;
308
309 public:
310     explicit FunDeclaration(SourceLocation,
311                             std::shared_ptr<FunPatternDeclaration>
312                             First);
313
314     void accept(Visitor *) override;
315
316     void setFunctionVals(std::shared_ptr<FunctionVals> V) {
317         this->FunVals = V; }
318     std::shared_ptr<FunctionVals> getFunctionVals() { return
319         this->FunVals; }
320 };
321 // End namespace ast.
322 // End namespace ssml.
323
324 namespace ssml {
325 namespace ast {
326 class TypeDeclaration : public Declaration {
327 private:
328     std::shared_ptr<ShortIdentifier> ID;
329     std::shared_ptr<Type> Typ;
330
331 public:
332     explicit TypeDeclaration(SourceLocation,
333                             std::shared_ptr<ShortIdentifier> ID,
334                             std::shared_ptr<Type> Typ);
335
336     void accept(Visitor *) override;
337     std::shared_ptr<ShortIdentifier> getID() { return this->ID; }
338     std::shared_ptr<Type> getType() { return this->Typ; }
339 };
340 // End namespace ast.
341 // End namespace ssml.
342
343 namespace ssml {
344 namespace ast {
345 class DatatypeBareInstanceDeclaration : public Declaration {
346 private:
347     std::shared_ptr<ShortIdentifier> ID;
348
349 public:

```

```

341     explicit DatatypeBareInstanceDeclaration(SourceLocation,
342                                               std::shared_ptr<ShortIdentifier>
                                               ID);
343     void accept(Visitor *) override;
344     std::shared_ptr<ShortIdentifier> getID() { return this->ID; }
345 };
346 } // End namespace ast.
347 } // End namespace ssml.
348
349 namespace ssml {
350 namespace ast {
351 class DatatypeTypedInstanceDeclaration
352     : public DatatypeBareInstanceDeclaration {
353 private:
354     std::shared_ptr<Type> Typ;
355
356 public:
357     explicit DatatypeTypedInstanceDeclaration(SourceLocation,
358                                               std::shared_ptr<ShortIdentifier>
                                               ID,
359                                               std::shared_ptr<Type>
                                               T);
360     void accept(Visitor *) override;
361     std::shared_ptr<Type> getType() { return this->Typ; }
362 };
363 } // End namespace ast.
364 } // End namespace ssml.
365
366 namespace ssml {
367 namespace ast {
368 class BareDatatypeDeclaration : public Declaration {
369 private:
370     std::shared_ptr<ShortIdentifier> ID;
371     std::shared_ptr<SeqDeclaration> Def;
372
373 public:
374     BareDatatypeDeclaration(SourceLocation,
375                             std::shared_ptr<ShortIdentifier> ID,
376                             std::shared_ptr<SeqDeclaration> Def);
377     void accept(Visitor *) override;
378     std::shared_ptr<ShortIdentifier> getID() { return this->ID; }
379     std::shared_ptr<SeqDeclaration> getDef() { return this->Def; }
380 };
381 } // End namespace ast.
382 } // End namespace ssml.
383
384 namespace ssml {
385 namespace ast {
386 class TypedDatatypeDeclaration : public BareDatatypeDeclaration {
387 private:
388     std::shared_ptr<SeqVariableType> TypeParams;
389
390 public:
391     TypedDatatypeDeclaration(SourceLocation,
392                             std::shared_ptr<SeqVariableType>
393                             TypeParams,
394                             std::shared_ptr<ShortIdentifier> ID,
395                             std::shared_ptr<SeqDeclaration> Def);
396     void accept(Visitor *) override;
397     std::shared_ptr<SeqVariableType> getTypeParams() { return
398         this->TypeParams; }
399 };

```

```

397 } // End namespace ast.
398 } // End namespace ssml.
399
400 namespace ssml {
401 namespace ast {
402 class ShortBareFunctorDeclaration : public Declaration {
403 private:
404     std::shared_ptr<ShortIdentifier> FunctorID;
405     std::shared_ptr<ShortIdentifier> ParamID;
406     std::shared_ptr<Definition> ParamSigDef;
407     std::shared_ptr<Definition> FunctorStructDef;
408
409 public:
410     ShortBareFunctorDeclaration(SourceLocation,
411                                std::shared_ptr<ShortIdentifier>
412                                FunctorID,
413                                std::shared_ptr<ShortIdentifier>
414                                ParamID,
415                                std::shared_ptr<Definition> ParamSig,
416                                std::shared_ptr<Definition>
417                                StructDef);
418
419     void accept(Visitor *) override;
420
421     std::shared_ptr<ShortIdentifier> getFunctorID() { return
422         this->FunctorID; }
423
424     std::shared_ptr<ShortIdentifier> getParamID() { return
425         this->ParamID; }
426
427     std::shared_ptr<Definition> getParamSigDef() { return
428         this->ParamSigDef; }
429
430     std::shared_ptr<Definition> getFunctorStructDef() {
431         return this->FunctorStructDef;
432     }
433 };
434 } // End namespace ast.
435 } // End namespace ssml.
436
437 namespace ssml {
438 namespace ast {
439 class ShortRestrictedFunctorDeclaration : public
440     ShortBareFunctorDeclaration {
441 private:
442     RestrictingSigDefinition FunctorSigDef;
443
444 public:
445     ShortRestrictedFunctorDeclaration(SourceLocation,
446                                       std::shared_ptr<ShortIdentifier>
447                                       FunctorID,
448                                       std::shared_ptr<ShortIdentifier>
449                                       ParamID,
450                                       std::shared_ptr<Definition>
451                                       ParamSig,
452                                       RestrictingSigDefinition
453                                       FunctorSigDef,
454                                       std::shared_ptr<Definition>
455                                       StructDef);
456
457     void accept(Visitor *) override;
458
459     std::shared_ptr<Definition> getFunctorSigDef() {
460         return this->FunctorSigDef.getSigDef();
461     }

```

```

447     }
448
449     bool isFunctorSigDefTransparent() {
450         return this->FunctorSigDef.isTransparent();
451     }
452 };
453 } // End namespace ast.
454 } // End namespace ssml.
455
456 namespace ssml {
457     namespace ast {
458         class LongBareFunctorDeclaration : public Declaration {
459         private:
460             std::shared_ptr<ShortIdentifier> FunctorID;
461             std::shared_ptr<SeqDeclaration> Params;
462             std::shared_ptr<Definition> FunctorStructDef;
463
464         public:
465             LongBareFunctorDeclaration(SourceLocation,
466                                     std::shared_ptr<ShortIdentifier>
467                                         FunctorID,
468                                     std::shared_ptr<SeqDeclaration>
469                                         Params,
470                                     std::shared_ptr<Definition>
471                                         StructDef);
472
473             void accept(Visitor *) override;
474
475             std::shared_ptr<ShortIdentifier> getFunctorID() { return
476                 this->FunctorID; }
477
478             std::shared_ptr<SeqDeclaration> getParams() { return
479                 this->Params; }
480
481             std::shared_ptr<Definition> getFunctorStructDef() {
482                 return this->FunctorStructDef;
483             }
484         };
485     } // End namespace ast.
486 } // End namespace ssml.
487
488 namespace ssml {
489     namespace ast {
490         class LongRestrictedFunctorDeclaration : public
491             LongBareFunctorDeclaration {
492         private:
493             RestrictingSigDefinition FunctorSigDef;
494
495         public:
496             LongRestrictedFunctorDeclaration(SourceLocation,
497                                     std::shared_ptr<ShortIdentifier>
498                                         FunctorID,
499                                     std::shared_ptr<SeqDeclaration>
500                                         Params,
501                                     RestrictingSigDefinition
502                                         FunctorSigDef,
503                                     std::shared_ptr<Definition>
504                                         StructDef);
505
506             void accept(Visitor *) override;
507
508             std::shared_ptr<Definition> getFunctorSigDef() {
509                 return this->FunctorSigDef.getSigDef();
510             }
511         };
512     }
513 }

```

```

499     bool isFunctorSigDefTransparent() {
500         return this->FunctorSigDef.isTransparent();
501     }
502 };
503 } // End namespace ast.
504 } // End namespace ssml.
505
506 #endif // SSML_AST_DECLARATION_H
507

```

Listing 33: include/ssml/AST/Node.h

```

1  #ifndef SSML_AST_NODE_H
2  #define SSML_AST_NODE_H
3
4  #include "ssml/Common/SourceLocation.h"
5
6  namespace ssml {
7      namespace ast {
8          class Visitor;
9      } // End namespace ast.
10 } // End namespace ssml.
11
12 namespace ssml {
13     namespace ast {
14         class Node {
15             SourceLocation Location;
16
17         protected:
18             explicit Node(SourceLocation Loc);
19
20         public:
21             Node(const Node &) = delete;
22             Node &operator=(const Node &) = delete;
23             virtual ~Node() = 0;
24
25             virtual void accept(Visitor *) = 0;
26
27             SourceLocation getLocation() const { return Location; }
28             void setLocation(SourceLocation L) { this->Location = L; }
29         };
30     } // End namespace ast.
31 } // End namespace ssml.
32
33 #endif // SSML_AST_NODE_H

```

Listing 34: include/ssml/AST/Identifier.h

```

1  #ifndef SSML_AST_IDENTIFIER_H
2  #define SSML_AST_IDENTIFIER_H
3
4  #include "Node.h"
5
6  #include <vector>
7  #include <string>
8  #include <memory>
9
10 namespace ssml {
11     namespace ast {
12         class IntLiteral;
13     } // End namespace ast.

```

```

14 } // End namespace ssml.
15
16 namespace ssml {
17 namespace ast {
18 class Identifier : public Node {
19 protected:
20     Identifier(SourceLocation);
21
22 public:
23     virtual std::string toString() = 0;
24 };
25 } // End namespace ast.
26 } // End namespace ssml.
27
28 namespace ssml {
29 namespace ast {
30 class ShortIdentifier : public Identifier {
31 private:
32     std::shared_ptr<std::string> ID;
33
34 public:
35     ShortIdentifier(SourceLocation, std::shared_ptr<std::string> ID);
36     void accept(Visitor *) override;
37     std::string toString() override;
38     std::shared_ptr<const std::string> getID() const { return
        this->ID; }
39 };
40 } // End namespace ast.
41 } // End namespace ssml.
42
43 namespace ssml {
44 namespace ast {
45 class LongIdentifier : public Identifier,
46                       public
47                       std::vector<std::shared_ptr<ShortIdentifier>>
48 {
49 public:
50     LongIdentifier(SourceLocation, std::shared_ptr<ShortIdentifier>
51         First);
52     void accept(Visitor *) override;
53     std::string toString() override;
54 };
55 } // End namespace ast.
56 } // End namespace ssml.
57
58 namespace ssml {
59 namespace ast {
60 class IntIdentifier : public Identifier {
61 private:
62     std::shared_ptr<IntLiteral> Literal;
63
64 public:
65     IntIdentifier(SourceLocation, std::shared_ptr<IntLiteral>
66         Literal);
67     void accept(Visitor *) override;
68     std::string toString() override;
69     std::shared_ptr<IntLiteral> getLiteral() { return this->Literal;
70     }
71 };
72 } // End namespace ast.
73 } // End namespace ssml.

```

```

70 namespace ssml {
71 namespace ast {
72 class SeqShortIdentifier
73     : public Identifier,
74     public std::vector<std::shared_ptr<ShortIdentifier>> {
75 public:
76     SeqShortIdentifier(SourceLocation,
77         std::shared_ptr<ShortIdentifier> First);
77     void accept(Visitor *) override;
78     std::string toString() override;
79 };
80 } // End namespace ast.
81 } // End namespace ssml.
82
83 namespace ssml {
84 namespace ast {
85 class SeqLongIdentifier
86     : public Identifier,
87     public std::vector<std::shared_ptr<LongIdentifier>> {
88 public:
89     SeqLongIdentifier(SourceLocation,
90         std::shared_ptr<LongIdentifier> First);
90     void accept(Visitor *) override;
91     std::string toString() override;
92 };
93 } // End namespace ast.
94 } // End namespace ssml.
95
96 #endif // SSML_AST_IDENTIFIER_H

```

Listing 35: include/ssml/AST/DumpVisitor.h

```

1  #ifndef SSML_AST_DUMPVISITOR_H
2  #define SSML_AST_DUMPVISITOR_H
3
4  #include "Visitor.h"
5
6  #include "ssml/Common/SourceLocation.h"
7
8  #include <memory>
9
10 namespace ssml {
11 namespace ast {
12 class Node;
13 } // End namespace ast.
14 } // End namespace ssml.
15
16 namespace ssml {
17 namespace ast {
18 class DumpVisitor : public Visitor {
19     unsigned Indent = 0;
20
21 private:
22     void printIndent();
23     void incrementIndent();
24     void decrementIndent();
25     void printAttribute(const char *Name, const std::string &Value);
26     void printAttribute(const char *Name,
27         std::shared_ptr<const std::string> Value);
28     void openBeginTag(const char *Tag, SourceLocation L);
29     void closeBeginTag();

```

```

30     void openCloseEndTag(const char *Tag);
31
32 public:
33     void visit(IntLiteral *) override;
34     void visit(RealLiteral *) override;
35     void visit(CharLiteral *) override;
36     void visit(StringLiteral *) override;
37
38     void visit(ShortIdentifier *) override;
39     void visit(LongIdentifier *) override;
40     void visit(IntIdentifier *) override;
41     void visit(SeqShortIdentifier *) override;
42     void visit(SeqLongIdentifier *) override;
43
44     void visit(Match *) override;
45     void visit(SeqMatch *) override;
46
47     void visit(SeqExpression *) override;
48     void visit(LiteralExpression *) override;
49     void visit(LongIdentifierExpression *) override;
50     void visit(TupleExpression *) override;
51     void visit(ListExpression *) override;
52     void visit(IdentifierEqualsExpression *) override;
53     void visit(RecordExpression *) override;
54     void visit(SelectorExpression *) override;
55     void visit(ApplyExpression *) override;
56     void visit(TypeExpression *) override;
57     void visit(OrElseExpression *) override;
58     void visit(AndAlsoExpression *) override;
59     void visit(LetExpression *) override;
60     void visit(IfExpression *) override;
61     void visit(WhileExpression *) override;
62     void visit(CaseExpression *) override;
63     void visit(LambdaExpression *) override;
64
65     void visit(SeqPattern *) override;
66     void visit(LiteralPattern *) override;
67     void visit(WildcardPattern *) override;
68     void visit(ShortIdentifierPattern *) override;
69     void visit(LongIdentifierPattern *) override;
70     void visit(TypePattern *) override;
71     void visit(AsPattern *) override;
72     void visit(ApplyPattern *) override;
73     void visit(IdentifierEqualsPattern *) override;
74     void visit(RecordPattern *) override;
75     void visit(ListPattern *) override;
76     void visit(TuplePattern *) override;
77
78     void visit(SeqDeclaration *) override;
79     void visit(Root *) override;
80     void visit(ValDeclaration *) override;
81     void visit(OpenDeclaration *) override;
82     void visit(NonfixDeclaration *) override;
83     void visit(InfixDeclaration *) override;
84     void visit(InfixRDeclaration *) override;
85     void visit(RestrictedStructDeclaration *) override;
86     void visit(BareStructDeclaration *) override;
87     void visit(AbstractValDeclaration *) override;
88     void visit(AbstractTypeDeclaration *) override;
89     void visit(AbstractStructDeclaration *) override;
90     void visit(SharingTypeDeclaration *) override;
91     void visit(SigDeclaration *) override;

```

```

92     void visit(FunPatternDeclaration *) override;
93     void visit(FunDeclaration *) override;
94     void visit(TypeDeclaration *) override;
95     void visit(DatatypeBareInstanceDeclaration *) override;
96     void visit(DatatypeTypedInstanceDeclaration *) override;
97     void visit(BareDatatypeDeclaration *) override;
98     void visit(TypedDatatypeDeclaration *) override;
99     void visit(ShortBareFunctorDeclaration *) override;
100    void visit(ShortRestrictedFunctorDeclaration *) override;
101    void visit(LongBareFunctorDeclaration *) override;
102    void visit(LongRestrictedFunctorDeclaration *) override;
103
104    void visit(LongIdentifierDefinition *) override;
105    void visit(StructDefinition *) override;
106    void visit(AnnotationDefinition *) override;
107    void visit(ShortFunctorDefinition *) override;
108    void visit(LongFunctorDefinition *) override;
109    void visit(SigDefinition *) override;
110
111    void visit(LongIdentifierType *) override;
112    void visit(VariableType *) override;
113    void visit(SeqType *) override;
114    void visit(SeqVariableType *) override;
115    void visit(TupleType *) override;
116    void visit(IdentifierColonType *) override;
117    void visit(RecordType *) override;
118    void visit(ApplyType *) override;
119    void visit(ProductType *) override;
120    void visit(FunctionType *) override;
121 };
122 } // End namespace ast.
123 } // End namespace ssml.
124
125 #endif // SSML_AST_DUMPVISITOR_H

```

Listing 36: include/ssml/AST/Visitor.h

```

1  #ifndef SSML_AST_VISITOR_H
2  #define SSML_AST_VISITOR_H
3
4  namespace ssml {
5  namespace ast {
6  class IntLiteral;
7  class ReallLiteral;
8  class CharLiteral;
9  class StringLiteral;
10
11  class ShortIdentifier;
12  class LongIdentifier;
13  class IntIdentifier;
14  class SeqShortIdentifier;
15  class SeqLongIdentifier;
16
17  class Match;
18  class SeqMatch;
19
20  class SeqExpression;
21  class LiteralExpression;
22  class LongIdentifierExpression;
23  class TupleExpression;
24  class ListExpression;

```

```

25 class IdentifierEqualsExpression;
26 class RecordExpression;
27 class SelectorExpression;
28 class ApplyExpression;
29 class TypeExpression;
30 class OrElseExpression;
31 class AndAlsoExpression;
32 class LetExpression;
33 class IfExpression;
34 class WhileExpression;
35 class CaseExpression;
36 class LambdaExpression;
37
38 class SeqPattern;
39 class LiteralPattern;
40 class WildcardPattern;
41 class ShortIdentifierPattern;
42 class LongIdentifierPattern;
43 class TypePattern;
44 class AsPattern;
45 class ApplyPattern;
46 class IdentifierEqualsPattern;
47 class RecordPattern;
48 class ListPattern;
49 class TuplePattern;
50
51 class SeqDeclaration;
52 class Root;
53 class ValDeclaration;
54 class OpenDeclaration;
55 class NonfixDeclaration;
56 class InfixDeclaration;
57 class InfixRDeclaration;
58 class RestrictedStructDeclaration;
59 class BareStructDeclaration;
60 class AbstractValDeclaration;
61 class AbstractTypeDeclaration;
62 class AbstractStructDeclaration;
63 class SharingTypeDeclaration;
64 class SigDeclaration;
65 class FunPatternDeclaration;
66 class FunDeclaration;
67 class TypeDeclaration;
68 class DatatypeBareInstanceDeclaration;
69 class DatatypeTypedInstanceDeclaration;
70 class BareDatatypeDeclaration;
71 class TypedDatatypeDeclaration;
72 class ShortBareFunctorDeclaration;
73 class ShortRestrictedFunctorDeclaration;
74 class LongBareFunctorDeclaration;
75 class LongRestrictedFunctorDeclaration;
76
77 class LongIdentifierDefinition;
78 class StructDefinition;
79 class AnnotationDefinition;
80 class ShortFunctorDefinition;
81 class LongFunctorDefinition;
82 class SigDefinition;
83
84 class LongIdentifierType;
85 class VariableType;
86 class SeqType;

```

```

87 class SeqVariableType;
88 class TupleType;
89 class IdentifierColonType;
90 class RecordType;
91 class ApplyType;
92 class ProductType;
93 class FunctionType;
94 } // End namespace ast.
95 } // End namespace ssml.
96
97 namespace ssml {
98 namespace ast {
99 class Visitor {
100 public:
101     virtual ~Visitor() = 0;
102
103     virtual void visit(IntLiteral *) = 0;
104     virtual void visit(RealLiteral *);
105     virtual void visit(CharLiteral *);
106     virtual void visit(StringLiteral *);
107
108     virtual void visit(ShortIdentifier *) = 0;
109     virtual void visit(LongIdentifier *) = 0;
110     virtual void visit(IntIdentifier *);
111     virtual void visit(SeqShortIdentifier *) = 0;
112     virtual void visit(SeqLongIdentifier *) = 0;
113
114     virtual void visit(Match *) = 0;
115     virtual void visit(SeqMatch *) = 0;
116
117     virtual void visit(SeqExpression *) = 0;
118     virtual void visit(LiteralExpression *) = 0;
119     virtual void visit(LongIdentifierExpression *) = 0;
120     virtual void visit(TupleExpression *) = 0;
121     virtual void visit(ListExpression *) = 0;
122     virtual void visit(IdentifierEqualsExpression *);
123     virtual void visit(RecordExpression *);
124     virtual void visit(SelectorExpression *);
125     virtual void visit(ApplyExpression *) = 0;
126     virtual void visit(TypeExpression *);
127     virtual void visit(OrElseExpression *) = 0;
128     virtual void visit(AndAlsoExpression *) = 0;
129     virtual void visit(IfExpression *) = 0;
130     virtual void visit(LetExpression *) = 0;
131     virtual void visit(WhileExpression *) = 0;
132     virtual void visit(CaseExpression *);
133     virtual void visit(LambdaExpression *) = 0;
134
135     virtual void visit(SeqPattern *) = 0;
136     virtual void visit(LiteralPattern *) = 0;
137     virtual void visit(WildcardPattern *) = 0;
138     virtual void visit(ShortIdentifierPattern *);
139     virtual void visit(LongIdentifierPattern *) = 0;
140     virtual void visit(TypePattern *) = 0;
141     virtual void visit(AsPattern *);
142     virtual void visit(ApplyPattern *) = 0;
143     virtual void visit(IdentifierEqualsPattern *);
144     virtual void visit(RecordPattern *);
145     virtual void visit(ListPattern *) = 0;
146     virtual void visit(TuplePattern *) = 0;
147
148     virtual void visit(SeqDeclaration *) = 0;

```

```

149     virtual void visit(Root *) = 0;
150     virtual void visit(ValDeclaration *) = 0;
151     virtual void visit(OpenDeclaration *);
152     virtual void visit(NonfixDeclaration *) = 0;
153     virtual void visit(InfixDeclaration *) = 0;
154     virtual void visit(InfixRDeclaration *) = 0;
155     virtual void visit(RestrictedStructDeclaration *);
156     virtual void visit(BareStructDeclaration *);
157     virtual void visit(AbstractValDeclaration *);
158     virtual void visit(AbstractTypeDeclaration *);
159     virtual void visit(AbstractStructDeclaration *);
160     virtual void visit(SharingTypeDeclaration *);
161     virtual void visit(SigDeclaration *);
162     virtual void visit(FunPatternDeclaration *) = 0;
163     virtual void visit(FunDeclaration *) = 0;
164     virtual void visit(TypeDeclaration *);
165     virtual void visit(DatatypeBareInstanceDeclaration *) = 0;
166     virtual void visit(DatatypeTypedInstanceDeclaration *) = 0;
167     virtual void visit(BareDatatypeDeclaration *) = 0;
168     virtual void visit(TypedDatatypeDeclaration *) = 0;
169     virtual void visit(ShortBareFunctorDeclaration *);
170     virtual void visit(ShortRestrictedFunctorDeclaration *);
171     virtual void visit(LongBareFunctorDeclaration *);
172     virtual void visit(LongRestrictedFunctorDeclaration *);
173
174     virtual void visit(LongIdentifierDefinition *);
175     virtual void visit(StructDefinition *);
176     virtual void visit(AnnotationDefinition *);
177     virtual void visit(ShortFunctorDefinition *);
178     virtual void visit(LongFunctorDefinition *);
179     virtual void visit(SigDefinition *);
180
181     virtual void visit(LongIdentifierType *) = 0;
182     virtual void visit(VariableType *) = 0;
183     virtual void visit(SeqType *) = 0;
184     virtual void visit(SeqVariableType *) = 0;
185     virtual void visit(TupleType *) = 0;
186     virtual void visit(IdentifierColonType *);
187     virtual void visit(RecordType *);
188     virtual void visit(ApplyType *) = 0;
189     virtual void visit(ProductType *) = 0;
190     virtual void visit(FunctionType *) = 0;
191 };
192 } // End namespace ast.
193 } // End namespace ssml.
194
195 #endif // SSML_AST_VISITOR_H

```

Listing 37: include/ssml/Typecheck/Typecheck.h

```

1  #ifndef SSML_TYPECHECK_TYPECHECK_H
2  #define SSML_TYPECHECK_TYPECHECK_H
3
4  namespace ssml {
5  namespace ast {
6  class Root;
7  } // End namespace ast.
8  } // End namespace ssml.
9
10 namespace ssml {
11 namespace typecheck {

```

```

12 void typecheck(ssml::ast::Root *);
13 } // End namespace typecheck.
14 } // End namespace ssml.
15
16 #endif // SSML_TYPECHECK_TYPECHECK_H

```

Listing 38: lib/Codegen/TypeMap.cpp

```

1  #include "TypeMap.h"
2
3  #include "ssml/Common/FatalExit.h"
4
5  #include <cassert>
6
7  using namespace ssml::codegen;
8
9  void TypeInstanceMap::insert(const std::string &In, int64_t V,
10                             bool B) {
11      Maps.back()[In] = {V, B};
12  }
13
14  std::pair<int64_t, bool> TypeInstanceMap::get(const std::string
15      &In, bool ANo) {
16      for (auto It = Maps.rbegin(), End = Maps.rend(); It != End; ++It)
17          if (It->count(In))
18              return It->find(In)->second;
19      if (!ANo)
20          fatalError("value " + In + " not in type map");
21      return {-1, false};
22  }
23
24  void TypeInstanceMap::enterScope() { Maps.push_back(MapType()); }
25
26  void TypeInstanceMap::leaveScope() { Maps.pop_back(); }

```

Listing 39: lib/Codegen/NameMap.h

```

1  #ifndef LLVM_TOOLS_SSML_LIB_CODEGEN_NAMEMAP_H
2  #define LLVM_TOOLS_SSML_LIB_CODEGEN_NAMEMAP_H
3
4  #include <map>
5  #include <vector>
6  #include <string>
7
8  namespace llvm {
9      class Value;
10  } // End namespace llvm.
11
12  namespace ssml {
13      namespace codegen {
14          class NameMap {
15          public:
16              struct ScopeValue {
17                  int32_t Scope;           // Which function scope. Starting from
18                                          // 0.
19                  int32_t Index;          // Index to function variables.
20
21                  bool isTypeInstance() const { return this->Scope >= 0 &&
22                      this->Index < 0; }
23                  bool isExisting() const { return this->Scope >= 0; }
24              };

```

```

23
24     using MapType = std::map<std::string, ScopeValue>;
25
26 private:
27     std::vector<MapType> Maps;
28
29 public:
30     void insert(const std::string &K, ScopeValue V);
31     ScopeValue get(const std::string &K);
32     void enterScope();
33     void leaveScope();
34 };
35 } // End namespace codegen.
36 } // End namespace ssml.
37
38 #endif // LLVM_TOOLS_SSML_LIB_CODEGEN_NAMEMAP_H

```

Listing 40: lib/Codegen/CodegenVisitor.h

```

1 #ifndef LLVM_TOOLS_SSML_LIB_CODEGEN_CODEGENVISITOR_H
2 #define LLVM_TOOLS_SSML_LIB_CODEGEN_CODEGENVISITOR_H
3
4 #include "TypeMap.h"
5 #include "NameMap.h"
6
7 #include "ssml/AST/FunctionVals.h"
8 #include "ssml/AST/Visitor.h"
9
10 #include "llvm/Support/raw_ostream.h"
11 #include "llvm/IR/Instructions.h"
12 #include "llvm/IR/InstrTypes.h"
13
14 #include <cstdlib>
15 #include <vector>
16 #include <map>
17 #include <string>
18 #include <memory>
19
20 #if !defined(SSML_DISABLE_GC) && !defined(SSML_MARK_SWEEP_GC)
21 #define SSML_VOLATILE_MEMORY
22 #endif
23
24 namespace llvm {
25     class Value;
26     class Module;
27     class Function;
28     class BasicBlock;
29     class AllocaInst;
30     class ConstantInt;
31 } // End namespace llvm.
32
33 namespace ssml {
34     namespace ast {
35         class Node;
36     } // End namespace ast.
37 } // End namespace ssml.
38
39 namespace ssml {
40     namespace codegen {
41         class FuncContext {
42     private:

```

```

43     llvm::Function *Fun;
44     FuncContext *Prev;
45     std::vector<std::map<std::string, int32_t>> LocalOffsets;
46     std::vector<std::map<std::string, int32_t>>
        LocalOffsetsRemembered;
47     llvm::AllocaInst *LocalPtr = nullptr;
48     std::vector<std::vector<size_t>> ScopeLocalIndicess;
49     std::vector<std::vector<llvm::Value *>> ScopeTempss;
50     int32_t FramePtrIndex;
51     int32_t LocalMemSize;
52     int32_t MaxLocalMemSize;
53     int32_t LocalMemSizeRemembered;
54     std::shared_ptr<ssml::ast::FunctionVals> FunVals;
55     std::pair<std::string, llvm::Function *> CurrFuncPair;
56     bool IsStepFunction = false;
57     bool IsInsideLetDecl = false;
58     llvm::Value *FirstArg = nullptr;
59     llvm::Value *SecondArg = nullptr;
60
61 private:
62     int32_t getNest(int32_t Depth);
63     int32_t getTrueNest(int32_t Depth);
64
65 public:
66     explicit FuncContext(llvm::Function *F,
67                         std::shared_ptr<ssml::ast::FunctionVals> Vs,
68                         const std::string &Name = "",
69                         llvm::Function *CurrFunc = nullptr)
70         : Fun(F), Prev(0), LocalOffsets{std::map<std::string,
        int32_t>()},
71         LocalPtr(0), LocalMemSize(0), MaxLocalMemSize(0),
72         LocalMemSizeRemembered(0), FunVals(Vs), CurrFuncPair{Name,
        CurrFunc} {
73         this->ScopeLocalIndicess.emplace_back();
74         this->ScopeTempss.emplace_back();
75         this->FramePtrIndex = 0;
76         this->LocalMemSize += this->FramePtrIndex * sizeof(int64_t);
77         this->MaxLocalMemSize =
78             (this->FramePtrIndex + 1 +
              this->FunVals->getFlags().size()) *
79             sizeof(int64_t);
80     }
81
82     void setIsStepFunction() { this->IsStepFunction = true; }
83
84     bool getIsStepFunction() { return this->IsStepFunction; }
85
86     const std::string &currFuncName() { return
        this->CurrFuncPair.first; }
87
88     llvm::Function *currFunc() { return this->CurrFuncPair.second; }
89
90     void setInsideLetDecl(bool B) { this->IsInsideLetDecl = B; }
91     bool getInsideLetDecl() { return this->IsInsideLetDecl; }
92
93     int32_t getFramePtrIndex() { return this->FramePtrIndex; }
94
95     void appendLocalIndex(size_t I) {
96         assert(this->ScopeLocalIndicess.size() > 0);
97         this->ScopeLocalIndicess.back().push_back(I);
98     }
99     void appendTemp(llvm::Value *Temp, size_t Idx = 0) {

```

```

100     if (this->ScopeTempss.size() == 1 && Idx == 1) {
101         this->ScopeTempss.back().push_back(Temp);
102     } else {
103         assert(this->ScopeTempss.size() > Idx);
104         this->ScopeTempss[this->ScopeTempss.size() - 1 -
            Idx].push_back(Temp);
105     }
106 }
107 std::vector<size_t> &getScopeLocalIndices() {
108     return this->ScopeLocalIndicess.back();
109 }
110 std::vector<llvm::Value *> &getScopeTemps() {
111     return this->ScopeTempss.back();
112 }
113
114 void func(llvm::Function *F) { this->Fun = F; }
115
116 llvm::Function *func() { return this->Fun; }
117
118 void setPrev(FuncContext *F) { this->Prev = F; }
119
120 FuncContext *getPrev() { return this->Prev; }
121
122 size_t getLocalMemSize() { return this->LocalMemSize; }
123
124 void enterScope();
125
126 void leaveScope();
127
128 void addLocal(const std::string &Name);
129
130 int32_t getLocalIndex(const std::string &Name);
131
132 void setLocalPtr(llvm::AllocaInst *In) { this->LocalPtr = In; }
133 llvm::Value *loadLocalPtr(llvm::BasicBlock *B) {
134 #ifdef SSML_VOLATILE_MEMORY
135     return new llvm::LoadInst(this->LocalPtr, "frame", true, B);
136 #else
137     return new llvm::LoadInst(this->LocalPtr, "frame", false, B);
138 #endif
139 }
140
141 llvm::AllocaInst *getPostTempsInst() { return this->LocalPtr; }
142
143 int32_t getNest() { return this->getNest(0); }
144
145 int32_t getTrueNest() { return this->getTrueNest(0); }
146
147 int32_t getMaxLocalMemSize() { return this->MaxLocalMemSize; }
148
149 void setFirstArg(llvm::Value *FirstArg) { this->FirstArg =
    FirstArg; }
150
151 void setSecondArg(llvm::Value *SecondArg) { this->SecondArg =
    SecondArg; }
152
153 llvm::Value *getFirstArg() { return this->FirstArg; }
154
155 llvm::Value *getSecondArg() { return this->SecondArg; }
156
157 void rememberContext();
158

```

```

159     void restoreRememberedContext();
160 };
161 } // End namespace codegen.
162 } // End namespace ssml.
163
164 namespace ssml {
165     namespace codegen {
166         class CodegenVisitor : public ssml::ast::Visitor {
167         private:
168             FuncContext *FuncContext = nullptr;
169             llvm::Module *Module;
170             NameMap Names;
171             TypeInstanceMap Instances;
172             std::shared_ptr<const std::string> DatatypeTypename;
173             llvm::BasicBlock *BasicBlock = nullptr;
174             llvm::BasicBlock *TrueBlock = nullptr;
175             llvm::BasicBlock *FalseBlock = nullptr;
176             llvm::Value *PatternExpr = nullptr;
177             llvm::Value *Temp = nullptr;
178             int64_t DatatypeInstanceValue = -1;
179             bool ExprWasTypeInstance = false;
180             bool InsideLetDecls = false;
181             bool LastExpr = false;
182
183         private:
184             void pushFunc(llvm::Function *F,
185                 std::shared_ptr<ssml::ast::FunctionVals>,
186                 const std::string &Name, llvm::Function *CurrFunc);
187             void pushFunc(llvm::Function *F,
188                 std::shared_ptr<ssml::ast::FunctionVals> V) {
189                 this->pushFunc(F, V, "", nullptr);
190             }
191             void changeFunc(llvm::Function *F);
192             FuncContext *peekFunc();
193             void popFunc();
194
195             llvm::Module *mod() { return this->Module; }
196
197             llvm::Value *loadLocalPtr() {
198                 return this->peekFunc()->loadLocalPtr(this->BasicBlock);
199             }
200
201             void beginMain(std::shared_ptr<ssml::ast::FunctionVals>);
202             void endMain();
203
204             void addFuncEntry(FuncContext *F, bool IsMain, bool
205                 PrevFrameIsLocals);
206             void fixupFuncEntry(FuncContext *);
207             llvm::CallInst *addAllocateCall(llvm::BasicBlock *B, llvm::Value
208                 *Size);
209             void addGCStore(llvm::Value *Source, llvm::Value *Dest,
210                 llvm::Value *Base);
211             llvm::Value *addGCLoad(llvm::Value *Source, llvm::Value *Base);
212             llvm::Value *addLocalLookup(llvm::BasicBlock *B,
213                 NameMap::ScopeValue SV);
214             void addLocalFuncVal(const std::string &Identifier, llvm::Value
215                 *Val);
216             void insertTypeInstances();
217
218             void enterScope();
219             void leaveScope(bool ZeroUnusedVals);
220

```

```

214     void doVisit(std::shared_ptr<ssml::ast::Node>);
215
216     bool tryBuiltinApplyExpr(ssml::ast::ApplyExpression *ApplyExpr);
217
218     void nextTemp();
219     void storeNextTemp(llvm::Value *);
220     llvm::Value *loadTemp(llvm::Value *Temp = nullptr);
221     void setExprResult(llvm::Value *V) { this->Temp = V; }
222     llvm::Value *getExprResult() { return this->Temp; }
223
224 public:
225     explicit CodegenVisitor();
226     ~CodegenVisitor();
227
228     void visit(ssml::ast::IntLiteral *) override;
229
230     void visit(ssml::ast::ShortIdentifier *) override;
231     void visit(ssml::ast::LongIdentifier *) override;
232     void visit(ssml::ast::SeqShortIdentifier *) override;
233     void visit(ssml::ast::SeqLongIdentifier *) override;
234
235     void visit(ssml::ast::Match *) override;
236     void visit(ssml::ast::SeqMatch *) override;
237
238     void visit(ssml::ast::SeqExpression *) override;
239     void visit(ssml::ast::LiteralExpression *) override;
240     void visit(ssml::ast::LongIdentifierExpression *) override;
241     void visit(ssml::ast::TupleExpression *) override;
242     void visit(ssml::ast::ListExpression *) override;
243     void visit(ssml::ast::ApplyExpression *) override;
244     void visit(ssml::ast::OrElseExpression *) override;
245     void visit(ssml::ast::AndAlsoExpression *) override;
246     void visit(ssml::ast::LetExpression *) override;
247     void visit(ssml::ast::IfExpression *) override;
248     void visit(ssml::ast::WhileExpression *) override;
249     void visit(ssml::ast::LambdaExpression *) override;
250
251     void visit(ssml::ast::SeqPattern *) override;
252     void visit(ssml::ast::LiteralPattern *) override;
253     void visit(ssml::ast::WildcardPattern *) override;
254     void visit(ssml::ast::LongIdentifierPattern *) override;
255     void visit(ssml::ast::TypePattern *) override;
256     void visit(ssml::ast::ApplyPattern *) override;
257     void visit(ssml::ast::ListPattern *) override;
258     void visit(ssml::ast::TuplePattern *) override;
259
260     void visit(ssml::ast::SeqDeclaration *) override;
261     void visit(ssml::ast::Root *) override;
262     void visit(ssml::ast::ValDeclaration *) override;
263     void visit(ssml::ast::NonfixDeclaration *) override;
264     void visit(ssml::ast::InfixDeclaration *) override;
265     void visit(ssml::ast::InfixRDeclaration *) override;
266     void visit(ssml::ast::FunPatternDeclaration *) override;
267     void visit(ssml::ast::FunDeclaration *) override;
268     void visit(ssml::ast::DatatypeBareInstanceDeclaration *)
269         override;
270     void visit(ssml::ast::DatatypeTypedInstanceDeclaration *)
271         override;
272     void visit(ssml::ast::BareDatatypeDeclaration *) override;
273     void visit(ssml::ast::TypedDatatypeDeclaration *) override;

```

```

274     void visit(ssml::ast::VariableType *) override;
275     void visit(ssml::ast::SeqType *) override;
276     void visit(ssml::ast::SeqVariableType *) override;
277     void visit(ssml::ast::TupleType *) override;
278     void visit(ssml::ast::ApplyType *) override;
279     void visit(ssml::ast::ProductType *) override;
280     void visit(ssml::ast::FunctionType *) override;
281 };
282 } // End namespace codegen.
283 } // End namespace ssml.
284
285 #endif // LLVM_TOOLS_SSML_LIB_CODEGEN_CODEGENVISITOR_H

```

Listing 41: lib/Codegen/TypeMap.h

```

1  #ifndef LLVM_TOOLS_SSML_LIB_CODEGEN_TYPEMAP_H
2  #define LLVM_TOOLS_SSML_LIB_CODEGEN_TYPEMAP_H
3
4  #include <vector>
5  #include <map>
6  #include <string>
7
8  namespace llvm {
9  class StructType;
10 } // End namespace llvm.
11
12 namespace ssml {
13 namespace codegen {
14 class TypeInstanceMap {
15 public:
16     using MapType = std::map<std::string, std::pair<int64_t, bool>>;
17
18 private:
19     std::vector<MapType> Maps;
20
21 public:
22     void insert(const std::string &Instance, int64_t Value, bool
                IsTyped);
23     std::pair<int64_t, bool> get(const std::string &Instance,
                                bool AllowNotInMap = false);
24     void enterScope();
25     void leaveScope();
26
27     int64_t getValue(const std::string &Instance) {
28         return this->get(Instance).first;
29     }
30     int64_t getIsTyped(const std::string &Instance) {
31         return this->get(Instance).second;
32     }
33 };
34 } // End namespace codegen.
35 } // End namespace ssml.
36
37 #endif // LLVM_TOOLS_SSML_LIB_CODEGEN_TYPEMAP_H

```

Listing 42: lib/Codegen/Codegen.cpp

```

1  #include "ssml/Codegen/Codegen.h"
2  #include "CodegenVisitor.h"
3
4  #include "ssml/AST/Declaration.h"

```

```

5 |
6 | void ssml::codegen::codegen(ssml::ast::Root *R) {
7 |     CodegenVisitor V;
8 |     R->accept(&V);
9 | }

```

Listing 43: lib/Codegen/CodegenVisitor.cpp

```

1 | #define DEBUG_TYPE "codegen"
2 |
3 | #include "CodegenVisitor.h"
4 |
5 | #include "ssml/AST/All.h"
6 | #include "ssml/Common/FatalExit.h"
7 |
8 | #include "llvm/IR/Verifier.h"
9 | #include "llvm/IR/Function.h"
10 | #include "llvm/IR/BasicBlock.h"
11 | #include "llvm/IR/Instructions.h"
12 | #include "llvm/IR/CallingConv.h"
13 | #include "llvm/IR/Module.h"
14 | #include "llvm/IR/LLVMContext.h"
15 | #include "llvm/IR/Constants.h"
16 | #include "llvm/Support/raw_ostream.h"
17 | #include "llvm/Support/ToolOutputFile.h"
18 | #include "llvm/Support/FileSystem.h"
19 | #include "llvm/Bitcode/ReaderWriter.h"
20 |
21 | #include "llvm/Support/raw_ostream.h"
22 | #include "llvm/Support/Debug.h"
23 |
24 | #include <system_error>
25 |
26 | #define CODEGEN_DLOG(msg) DEBUG(llvm::errs() << DEBUG_TYPE " :: "
27 |     << msg << '\n')
28 |
29 | using namespace ssml;
30 | using namespace ssml::codegen;
31 | using namespace ssml::ast;
32 |
33 | #if defined(SSML_SHADOW_STACK_GC)
34 | static const char gcstrategy[] = "shadow-stack";
35 | #elif !defined(SSML_DISABLE_GC)
36 | static const char gcstrategy[] = "ssml";
37 | #else
38 | static const char gcstrategy[] = "";
39 | #endif
40 |
41 | #if !defined(SSML_DISABLE_GC)
42 | static void setGC(llvm::Function *F) { F->setGC(gcstrategy); }
43 | #else // defined(SSML_DISABLE_GC)
44 | static void setGC(llvm::Function *F) {}
45 | #endif // !defined(SSML_DISABLE_GC)
46 |
47 | static const size_t alignment = sizeof(int64_t);
48 |
49 | static llvm::IntegerType *getInteger(llvm::Module *M, int32_t I) {
50 |     return llvm::IntegerType::get(M->getContext(), I);
51 | }
52 |
53 | static llvm::IntegerType *getInteger64(llvm::Module *M) {

```

```

53     return llvm::IntegerType::get(M->getContext(), 64);
54 }
55
56 static llvm::PointerType *getPointerTo(llvm::Type *V)
57     __attribute__((unused));
58 static llvm::PointerType *getPointerTo(llvm::Type *V) {
59     return llvm::PointerType::getUnqual(V);
60 }
61
62 static llvm::PointerType *getInteger64Pointer(llvm::Module *M) {
63     return llvm::PointerType::getUnqual(getInteger64(M));
64 }
65
66 static llvm::ConstantInt *getConst(llvm::Module *M, int64_t val,
67     int32_t Bits) {
68     return llvm::ConstantInt::getSigned(getInteger(M, Bits), val);
69 }
70
71 static llvm::ConstantInt *getConst(llvm::Module *M, int64_t val) {
72     return getConst(M, val, 64);
73 }
74
75 static llvm::ConstantInt *getZeroConst(llvm::Module *M) {
76     return getConst(M, 0);
77 }
78
79 static int64_t markValue(int64_t Val) { return Val * 2 | 1; }
80
81 static llvm::Value *markValue(llvm::Module *M, llvm::Value *V,
82     llvm::BasicBlock *B) {
83     auto Sh = llvm::BinaryOperator::Create(llvm::Instruction::Shl, V,
84         getConst(M, 1), "", B);
85     return llvm::BinaryOperator::CreateOr(Sh, getConst(M, 1), "", B);
86 }
87
88 static llvm::Value *unmarkValue(llvm::Module *M, llvm::Value *V,
89     llvm::BasicBlock *B) {
90     return llvm::BinaryOperator::Create(llvm::Instruction::AShr, V,
91         getConst(M, 1), "", B);
92 }
93
94 static llvm::Value *getSimpleBuiltinConst(llvm::Module *M,
95     const std::string &ID,
96     bool Marked) {
97     if (ID == "false")
98         return Marked ? getConst(M, markValue(0)) : getConst(M, 0);
99     else if (ID == "true")
100         return Marked ? getConst(M, markValue(1)) : getConst(M, 1);
101     return nullptr;
102 }
103
104 static llvm::Value *getTernaryBuiltinFunc(llvm::Module *M,
105     const std::string &ID) {
106     llvm::Value *Ret = nullptr;
107     if (ID == "Array.update") {
108         Ret = M->getGlobalVariable("frameArrayupdate");
109         assert(!Ret);
110     }
111     return Ret;
112 }
113
114 static llvm::Value *getBinaryBuiltinFunc(llvm::Module *M,
115     const std::string &ID) {

```

```

112     llvm::Value *Ret = nullptr;
113     if (ID == "+") {
114         Ret = M->getGlobalVariable("framePlus");
115         assert(!Ret);
116     } else if (ID == "-") {
117         Ret = M->getGlobalVariable("frameMinus");
118         assert(!Ret);
119     } else if (ID == "*") {
120         Ret = M->getGlobalVariable("frameMultiply");
121         assert(!Ret);
122     } else if (ID == "div") {
123         Ret = M->getGlobalVariable("frameDivision");
124         assert(!Ret);
125     } else if (ID == "mod") {
126         Ret = M->getGlobalVariable("frameModulo");
127         assert(!Ret);
128     } else if (ID == ":=") {
129         Ret = M->getGlobalVariable("frameRefassign");
130         assert(!Ret);
131     } else if (ID == "Array.array") {
132         Ret = M->getGlobalVariable("frameArray");
133         assert(!Ret);
134     } else if (ID == "Array.get") {
135         Ret = M->getGlobalVariable("frameArrayget");
136         assert(!Ret);
137     } else if (ID == "=") {
138         Ret = M->getGlobalVariable("frameEquals");
139         assert(!Ret);
140     } else if (ID == "<>") {
141         Ret = M->getGlobalVariable("frameNotEquals");
142         assert(!Ret);
143     } else if (ID == "<") {
144         Ret = M->getGlobalVariable("frameLess");
145         assert(!Ret);
146     } else if (ID == ">") {
147         Ret = M->getGlobalVariable("frameGreater");
148         assert(!Ret);
149     } else if (ID == "<=") {
150         Ret = M->getGlobalVariable("frameLessEquals");
151         assert(!Ret);
152     } else if (ID == ">=") {
153         Ret = M->getGlobalVariable("frameGreaterEquals");
154         assert(!Ret);
155     }
156     return Ret;
157 }
158
159 static llvm::Value *getUnaryBuiltinFunc(llvm::Module *M,
160                                         const std::string &ID) {
161     llvm::Value *Ret = nullptr;
162     if (ID == "print") {
163         Ret = M->getGlobalVariable("framePrintVal");
164         assert(!Ret);
165     } else if (ID == "~") {
166         Ret = M->getGlobalVariable("frameNegate");
167         assert(!Ret);
168     } else if (ID == "not") {
169         Ret = M->getGlobalVariable("frameNot");
170         assert(!Ret);
171     } else if (ID == "ref") {
172         Ret = M->getGlobalVariable("frameRef");
173         assert(!Ret);

```

```

174     } else if (ID == "Array.length") {
175         Ret = M->getGlobalVariable("frameArraylength");
176         assert(!Ret);
177     } else if (ID == "!") {
178         Ret = M->getGlobalVariable("frameDeref");
179         assert(!Ret);
180     }
181     return Ret;
182 }
183
184 static llvm::Value *getBuiltinValue(llvm::Module *M, const
    std::string &ID) {
185     llvm::Value *Ret = nullptr;
186     if (ID == "Array.empty") {
187         Ret = M->getGlobalVariable("arrayempty");
188         assert(!Ret);
189     }
190     return Ret;
191 }
192
193 static llvm::Value *getBuiltin(llvm::Module *M, const std::string
    &ID) {
194     llvm::Value *Ret = getTernaryBuiltinFunc(M, ID);
195     if (!Ret)
196         Ret = getBinaryBuiltinFunc(M, ID);
197     if (!Ret)
198         Ret = getUnaryBuiltinFunc(M, ID);
199     if (!Ret)
200         Ret = getBuiltinValue(M, ID);
201     return Ret;
202 }
203
204 static llvm::FunctionType *getStandardFuncType(llvm::Module *M) {
205     auto RetType = getInteger64(M);
206     llvm::SmallVector<llvm::Type *, 2> Arg;
207     Arg.push_back(getInteger64(M));
208     Arg.push_back(getInteger64(M));
209     auto FunType = llvm::FunctionType::get(RetType, Arg, false);
210     return FunType;
211 }
212
213 int32_t FuncContext::getNest(int32_t Depth) {
214     if (this->Prev)
215         return this->Prev->getNest(Depth + 1);
216     return Depth;
217 }
218
219 int32_t FuncContext::getTrueNest(int32_t Depth) {
220     if (this->Prev) {
221         if (this->currFunc())
222             return this->Prev->Prev->getNest(Depth + 1);
223         else
224             return this->Prev->getNest(Depth + 1);
225     }
226     return Depth;
227 }
228
229 void FuncContext::enterScope() {
230     this->ScopeLocalIndicess.emplace_back();
231     this->ScopeTempss.emplace_back();
232     this->LocalOffsets.emplace_back();
233 }

```

```

234
235 void FuncContext::leaveScope() {
236     this->LocalOffsets.pop_back();
237     this->ScopeLocalIndicess.pop_back();
238     this->ScopeTempss.pop_back();
239 }
240
241 void FuncContext::addLocal(const std::string &Name) {
242     this->LocalOffsets.back()[Name] = this->LocalMemSize;
243     this->LocalMemSize += alignment;
244     if (this->LocalMemSize > this->MaxLocalMemSize)
245         fatalExit("too many local variables in function");
246 }
247
248 int32_t FuncContext::getLocalIndex(const std::string &Name) {
249     for (auto It = LocalOffsets.rbegin(), End = LocalOffsets.rend();
250          It != End;
251          ++It) {
252         if (It->count(Name))
253             return It->find(Name)->second / alignment;
254     }
255     assert(0 && "local val name not found");
256     return 0;
257 }
258
259 void FuncContext::rememberContext() {
260     this->LocalMemSizeRemembered = this->LocalMemSize;
261     this->LocalOffsetsRemembered = this->LocalOffsets;
262 }
263
264 void FuncContext::restoreRememberedContext() {
265     this->LocalMemSize = this->LocalMemSizeRemembered;
266     this->LocalOffsets = this->LocalOffsetsRemembered;
267 }
268
269 static void addGCRootDecl(llvm::Module *M) {
270     auto FunRet = llvm::Type::getVoidTy(llvm::getGlobalContext());
271     llvm::SmallVector<llvm::Type *, 2> Arg;
272     Arg.push_back(llvm::PointerType::getUnqual(llvm::PointerType::getUnqual(
273         llvm::IntegerType::get(M->getContext(), 8))));
274     Arg.push_back(
275         llvm::PointerType::getUnqual(llvm::IntegerType::get(M->getContext(),
276             8)));
277     auto FunType = llvm::FunctionType::get(FunRet, Arg, false);
278     auto F = llvm::Function::Create(FunType,
279         llvm::GlobalValue::ExternalLinkage,
280         "llvm.gcroot", M);
281     F->setCallingConv(llvm::CallingConv::C);
282     setGC(F);
283 }
284
285 static void addPrintDecl(llvm::Module *M) {
286     auto FunRet = getInteger64Pointer(M);
287     llvm::SmallVector<llvm::Type *, 1> Arg;
288     Arg.push_back(getInteger64(M));
289     auto FunType = llvm::FunctionType::get(FunRet, Arg, false);
290     auto F = llvm::Function::Create(FunType,
291         llvm::GlobalValue::ExternalLinkage,
292         "print", M);
293     F->setCallingConv(llvm::CallingConv::C);
294     setGC(F);
295 }

```

```

292
293 #if 0
294 static void addPatternCheckDecl(llvm::Module *M) {
295     auto FunRet = llvm::Type::getVoidTy(llvm::getGlobalContext());
296     llvm::SmallVector<llvm::Type *, 1> Arg;
297     Arg.push_back(getInteger64(M));
298     auto FunType = llvm::FunctionType::get(FunRet, Arg, false);
299     auto F = llvm::Function::Create(FunType,
300                                     llvm::GlobalValue::ExternalLinkage,
301                                     "patternCheck", M);
302     F->setCallingConv(llvm::CallingConv::C);
303 }
304 static void callPatternCheck(llvm::Module *M, llvm::BasicBlock *B,
305                             llvm::Value *A) {
306     llvm::SmallVector<llvm::Value *, 1> Arg;
307     Arg.push_back(A);
308     auto F = M->getFunction("patternCheck");
309     assert(F);
310     llvm::CallInst::Create(F, Arg, "", B);
311 }
312 #endif
313
314 static void addFirstGenBeginDecl(llvm::Module *M) {
315 #ifdef SSML_GENERATIONAL_GC
316     auto Type = getInteger64(M);
317     new llvm::GlobalVariable(*M, Type, true,
318                             llvm::GlobalValue::ExternalLinkage,
319                             nullptr, "FirstGenBegin", nullptr,
320                             llvm::GlobalValue::NotThreadLocal, 0,
321                             true);
322 #endif
323 }
324 static void addFirstGenEndDecl(llvm::Module *M) {
325 #ifdef SSML_GENERATIONAL_GC
326     auto Type = getInteger64(M);
327     new llvm::GlobalVariable(*M, Type, true,
328                             llvm::GlobalValue::ExternalLinkage,
329                             nullptr, "FirstGenEnd", nullptr,
330                             llvm::GlobalValue::NotThreadLocal, 0,
331                             true);
332 #endif
333 }
334 static void addMemoryBarrierDecl(llvm::Module *M) {
335 #ifdef SSML_GENERATIONAL_GC
336     auto FunRet = llvm::Type::getVoidTy(llvm::getGlobalContext());
337     llvm::SmallVector<llvm::Type *, 1> Arg;
338     Arg.push_back(getInteger64Pointer(M));
339     // Arg.push_back(getInteger64(M));
340     auto FunType = llvm::FunctionType::get(FunRet, Arg, false);
341     auto F = llvm::Function::Create(FunType,
342                                     llvm::GlobalValue::ExternalLinkage,
343                                     "rememberset_insert", M);
344     F->setCallingConv(llvm::CallingConv::C);
345 #endif
346 }
347 static void addMemoryBarrierDef(llvm::Module *M) {
348 #ifdef SSML_GENERATIONAL_GC
349     auto FunRet = llvm::Type::getVoidTy(llvm::getGlobalContext());

```

```

348     llvm::SmallVector<llvm::Type *, 2> Arg;
349     Arg.push_back(getInteger64Pointer(M));
350     Arg.push_back(getInteger64(M));
351     auto FunType = llvm::FunctionType::get(FunRet, Arg, false);
352     auto F = llvm::Function::Create(FunType,
        llvm::GlobalValue::InternalLinkage,
        "ssmlMemoryBarrier", M);
353     F->setCallingConv(llvm::CallingConv::C);
354
355
356     auto Entry = llvm::BasicBlock::Create(M->getContext(),
        "entrylabel", F, 0);
357
358     auto ArgIt = F->arg_begin();
359     llvm::Value *Base = ArgIt;
360     llvm::Value *Source = ++ArgIt;
361
362     auto TB1 = llvm::BasicBlock::Create(llvm::getContext());
363     auto TB2 = llvm::BasicBlock::Create(llvm::getContext());
364     auto TB3 = llvm::BasicBlock::Create(llvm::getContext());
365     auto FB = llvm::BasicBlock::Create(llvm::getContext());
366     auto And = llvm::BinaryOperator::CreateAnd(Source, getConst(M,
        1), "", Entry);
367     auto Cond =
368         new llvm::ICmpInst(*Entry, llvm::CmpInst::ICMP_NE, And,
            getConst(M, 1));
369     llvm::BranchInst::Create(TB1, FB, Cond, Entry);
370     TB1->insertInto(F);
371
372     auto BaseInt = new llvm::PtrToIntInst(Base, getInteger64(M), "",
        TB1);
373
374     llvm::Value *Fen = M->getGlobalVariable("FirstGenEnd");
375     assert(Fen);
376     Fen = new llvm::LoadInst(Fen, "", TB1);
377
378     Cond = new llvm::ICmpInst(*TB1, llvm::CmpInst::ICMP_UGE,
        BaseInt, Fen);
379     llvm::BranchInst::Create(TB3, TB2, Cond, TB1);
380     TB2->insertInto(F);
381
382     llvm::Value *Fbe = M->getGlobalVariable("FirstGenBegin");
383     assert(Fbe);
384     Fbe = new llvm::LoadInst(Fbe, "", TB2);
385
386     Cond = new llvm::ICmpInst(*TB2, llvm::CmpInst::ICMP_ULT,
        BaseInt, Fbe);
387     llvm::BranchInst::Create(TB3, FB, Cond, TB2);
388     TB3->insertInto(F);
389
390     llvm::SmallVector<llvm::Value *, 1> Args;
391     Args.push_back(Base);
392
393     auto Barrier = M->getFunction("rememberset_insert");
394     auto Call = llvm::CallInst::Create(Barrier, Args, "", TB3);
395     Call->setTailCall();
396     llvm::BranchInst::Create(FB, TB3);
397
398     FB->insertInto(F);
399     llvm::ReturnInst::Create(M->getContext(), nullptr, FB);
400 #endif
401 }
402

```

```

403 static void addGCWriteDecl(llvm::Module *M) {
404     auto FunRet = llvm::Type::getVoidTy(llvm::getGlobalContext());
405     llvm::SmallVector<llvm::Type *, 3> Arg;
406     Arg.push_back(
407         llvm::PointerType::getUnqual(llvm::IntegerType::get(M->getContext(),
408             8)));
409     Arg.push_back(
410         llvm::PointerType::getUnqual(llvm::IntegerType::get(M->getContext(),
411             8)));
412     Arg.push_back(llvm::PointerType::getUnqual(llvm::PointerType::getUnqual(
413         llvm::IntegerType::get(M->getContext(), 8))));
414     auto FunType = llvm::FunctionType::get(FunRet, Arg, false);
415     auto F = llvm::Function::Create(FunType,
416         llvm::GlobalValue::ExternalLinkage,
417         "llvm.gcwrite", M);
418     F->setCallingConv(llvm::CallingConv::C);
419     setGC(F);
420 }
421
422 static void addGCReadDecl(llvm::Module *M) {
423     auto FunRet =
424         llvm::PointerType::getUnqual(llvm::IntegerType::get(M->getContext(),
425             8));
426     llvm::SmallVector<llvm::Type *, 2> Arg;
427     Arg.push_back(
428         llvm::PointerType::getUnqual(llvm::IntegerType::get(M->getContext(),
429             8)));
430     Arg.push_back(llvm::PointerType::getUnqual(llvm::PointerType::getUnqual(
431         llvm::IntegerType::get(M->getContext(), 8))));
432     auto FunType = llvm::FunctionType::get(FunRet, Arg, false);
433     auto F = llvm::Function::Create(FunType,
434         llvm::GlobalValue::ExternalLinkage,
435         "llvm.gcread", M);
436     F->setCallingConv(llvm::CallingConv::C);
437     setGC(F);
438 }
439
440 static void addAllocateDecl(llvm::Module *M) {
441     auto FunRet = getInteger64Pointer(M);
442     llvm::SmallVector<llvm::Type *, 1> Arg;
443     Arg.push_back(getInteger64(M));
444     auto FunType = llvm::FunctionType::get(FunRet, Arg, false);
445     auto F = llvm::Function::Create(FunType,
446         llvm::GlobalValue::ExternalLinkage,
447         "allocate", M);
448     F->setCallingConv(llvm::CallingConv::C);
449     setGC(F);
450 }
451
452 static void addDoarrayDecl(llvm::Module *M) {
453     auto FunRet = getInteger64(M);
454     llvm::SmallVector<llvm::Type *, 2> Args;
455     Args.push_back(getInteger64(M));
456     Args.push_back(getInteger64(M));
457     auto FunType = llvm::FunctionType::get(FunRet, Args, false);
458     auto F = llvm::Function::Create(FunType,
459         llvm::GlobalValue::ExternalLinkage,
460         "doarray", M);
461     F->setCallingConv(llvm::CallingConv::C);
462     setGC(F);
463 }

```

```

457 static void addDoarraygetDecl(llvm::Module *M) {
458     auto FunRet = getInteger64(M);
459     llvm::SmallVector<llvm::Type *, 2> Args;
460     Args.push_back(getInteger64(M));
461     Args.push_back(getInteger64(M));
462     auto FunType = llvm::FunctionType::get(FunRet, Args, false);
463     auto F = llvm::Function::Create(FunType,
        llvm::GlobalValue::ExternalLinkage,
        "doarrayget", M);
464     F->setCallingConv(llvm::CallingConv::C);
465     setGC(F);
466 }
467
468 static void addDoarrayupdateDecl(llvm::Module *M) {
469     auto FunRet = getInteger64(M);
470     llvm::SmallVector<llvm::Type *, 3> Args;
471     Args.push_back(getInteger64(M));
472     Args.push_back(getInteger64(M));
473     Args.push_back(getInteger64(M));
474     auto FunType = llvm::FunctionType::get(FunRet, Args, false);
475     auto F = llvm::Function::Create(FunType,
        llvm::GlobalValue::ExternalLinkage,
        "doarrayupdate", M);
476     F->setCallingConv(llvm::CallingConv::C);
477     setGC(F);
478 }
479
480 static void addArrayemptyDecl(llvm::Module *M) {
481     auto Type = getInteger64(M);
482     new llvm::GlobalVariable(*M, Type, true,
        llvm::GlobalValue::ExternalLinkage,
        nullptr, "arrayempty", nullptr,
        llvm::GlobalValue::NotThreadLocal, 0,
        true);
483 }
484
485 static void addFrameArrayupdateDecl(llvm::Module *M) {
486     auto Type = getInteger64Pointer(M);
487     new llvm::GlobalVariable(*M, Type, true,
        llvm::GlobalValue::ExternalLinkage,
        nullptr, "frameArrayupdate", nullptr,
        llvm::GlobalValue::NotThreadLocal, 0,
        true);
488 }
489
490 static void addFramePrintDecl(llvm::Module *M) {
491     auto Type = getInteger64Pointer(M);
492     new llvm::GlobalVariable(*M, Type, true,
        llvm::GlobalValue::ExternalLinkage,
        nullptr, "framePrintVal", nullptr,
        llvm::GlobalValue::NotThreadLocal, 0,
        true);
493 }
494
495 static void addFramePlusDecl(llvm::Module *M) {
496     auto Type = getInteger64Pointer(M);
497     new llvm::GlobalVariable(*M, Type, true,
        llvm::GlobalValue::ExternalLinkage,
        nullptr, "framePlus", nullptr,
        llvm::GlobalValue::NotThreadLocal, 0,
        true);
498 }
499
500 static void addFramePlusDecl(llvm::Module *M) {
501     auto Type = getInteger64Pointer(M);
502     new llvm::GlobalVariable(*M, Type, true,
        llvm::GlobalValue::ExternalLinkage,
        nullptr, "framePlus", nullptr,
        llvm::GlobalValue::NotThreadLocal, 0,
        true);
503 }
504
505 static void addFramePlusDecl(llvm::Module *M) {
506     auto Type = getInteger64Pointer(M);
507     new llvm::GlobalVariable(*M, Type, true,
        llvm::GlobalValue::ExternalLinkage,
        nullptr, "framePlus", nullptr,
        llvm::GlobalValue::NotThreadLocal, 0,
        true);
508 }

```

```

509
510 static void addFrameMinusDecl(llvm::Module *M) {
511     auto Type = getInteger64Pointer(M);
512     new llvm::GlobalVariable(*M, Type, true,
513                             llvm::GlobalValue::ExternalLinkage,
514                             nullptr, "frameMinus", nullptr,
515                             llvm::GlobalValue::NotThreadLocal, 0,
516                             true);
517 }
518
519 static void addFrameMultiplyDecl(llvm::Module *M) {
520     auto Type = getInteger64Pointer(M);
521     new llvm::GlobalVariable(*M, Type, true,
522                             llvm::GlobalValue::ExternalLinkage,
523                             nullptr, "frameMultiply", nullptr,
524                             llvm::GlobalValue::NotThreadLocal, 0,
525                             true);
526 }
527
528 static void addFrameDivisionDecl(llvm::Module *M) {
529     auto Type = getInteger64Pointer(M);
530     new llvm::GlobalVariable(*M, Type, true,
531                             llvm::GlobalValue::ExternalLinkage,
532                             nullptr, "frameDivision", nullptr,
533                             llvm::GlobalValue::NotThreadLocal, 0,
534                             true);
535 }
536
537 static void addFrameModuloDecl(llvm::Module *M) {
538     auto Type = getInteger64Pointer(M);
539     new llvm::GlobalVariable(*M, Type, true,
540                             llvm::GlobalValue::ExternalLinkage,
541                             nullptr, "frameModulo", nullptr,
542                             llvm::GlobalValue::NotThreadLocal, 0,
543                             true);
544 }
545
546 static void addFrameRefassignDecl(llvm::Module *M) {
547     auto Type = getInteger64Pointer(M);
548     new llvm::GlobalVariable(*M, Type, true,
549                             llvm::GlobalValue::ExternalLinkage,
550                             nullptr, "frameRefassign", nullptr,
551                             llvm::GlobalValue::NotThreadLocal, 0,
552                             true);
553 }
554
555 static void addFrameArrayDecl(llvm::Module *M) {
556     auto Type = getInteger64Pointer(M);
557     new llvm::GlobalVariable(*M, Type, true,
558                             llvm::GlobalValue::ExternalLinkage,
559                             nullptr, "frameArray", nullptr,
560                             llvm::GlobalValue::NotThreadLocal, 0,
561                             true);
562 }
563
564 static void addFrameArraygetDecl(llvm::Module *M) {
565     auto Type = getInteger64Pointer(M);
566     new llvm::GlobalVariable(*M, Type, true,
567                             llvm::GlobalValue::ExternalLinkage,
568                             nullptr, "frameArrayget", nullptr,
569                             llvm::GlobalValue::NotThreadLocal, 0,
570                             true);
571 }

```

```

557 }
558
559 static void addFrameNegateDecl(llvm::Module *M) {
560     auto Type = getInteger64Pointer(M);
561     new llvm::GlobalVariable(*M, Type, true,
562                             llvm::GlobalValue::ExternalLinkage,
563                             nullptr, "frameNegate", nullptr,
564                             llvm::GlobalValue::NotThreadLocal, 0,
565                             true);
566 }
567
568 static void addFrameNotDecl(llvm::Module *M) {
569     auto Type = getInteger64Pointer(M);
570     new llvm::GlobalVariable(*M, Type, true,
571                             llvm::GlobalValue::ExternalLinkage,
572                             nullptr, "frameNot", nullptr,
573                             llvm::GlobalValue::NotThreadLocal, 0,
574                             true);
575 }
576
577 static void addFrameRefDecl(llvm::Module *M) {
578     auto Type = getInteger64Pointer(M);
579     new llvm::GlobalVariable(*M, Type, true,
580                             llvm::GlobalValue::ExternalLinkage,
581                             nullptr, "frameRef", nullptr,
582                             llvm::GlobalValue::NotThreadLocal, 0,
583                             true);
584 }
585
586 static void addFrameDerefDecl(llvm::Module *M) {
587     auto Type = getInteger64Pointer(M);
588     new llvm::GlobalVariable(*M, Type, true,
589                             llvm::GlobalValue::ExternalLinkage,
590                             nullptr, "frameDeref", nullptr,
591                             llvm::GlobalValue::NotThreadLocal, 0,
592                             true);
593 }
594
595 static void addFrameArraylengthDecl(llvm::Module *M) {
596     auto Type = getInteger64Pointer(M);
597     new llvm::GlobalVariable(*M, Type, true,
598                             llvm::GlobalValue::ExternalLinkage,
599                             nullptr, "frameArraylength", nullptr,
600                             llvm::GlobalValue::NotThreadLocal, 0,
601                             true);
602 }
603
604 static void addFrameEqualsDecl(llvm::Module *M) {
605     auto Type = getInteger64Pointer(M);
606     new llvm::GlobalVariable(*M, Type, true,
607                             llvm::GlobalValue::ExternalLinkage,
608                             nullptr, "frameEquals", nullptr,
609                             llvm::GlobalValue::NotThreadLocal, 0,
610                             true);
611 }
612
613 static void addFrameNotEqualsDecl(llvm::Module *M) {
614     auto Type = getInteger64Pointer(M);
615     new llvm::GlobalVariable(*M, Type, true,
616                             llvm::GlobalValue::ExternalLinkage,
617                             nullptr, "frameNotEquals", nullptr,
618                             llvm::GlobalValue::NotThreadLocal, 0,
619                             true);
620 }

```



```

        true);
606 }
607
608 static void addFrameLessDecl(llvm::Module *M) {
609     auto Type = getInteger64Pointer(M);
610     new llvm::GlobalVariable(*M, Type, true,
        llvm::GlobalValue::ExternalLinkage,
611         nullptr, "frameLess", nullptr,
612         llvm::GlobalValue::NotThreadLocal, 0,
        true);
613 }
614
615 static void addFrameGreaterDecl(llvm::Module *M) {
616     auto Type = getInteger64Pointer(M);
617     new llvm::GlobalVariable(*M, Type, true,
        llvm::GlobalValue::ExternalLinkage,
618         nullptr, "frameGreater", nullptr,
619         llvm::GlobalValue::NotThreadLocal, 0,
        true);
620 }
621
622 static void addFrameLessEqualsDecl(llvm::Module *M) {
623     auto Type = getInteger64Pointer(M);
624     new llvm::GlobalVariable(*M, Type, true,
        llvm::GlobalValue::ExternalLinkage,
625         nullptr, "frameLessEquals", nullptr,
626         llvm::GlobalValue::NotThreadLocal, 0,
        true);
627 }
628
629 static void addFrameGreaterEqualsDecl(llvm::Module *M) {
630     auto Type = getInteger64Pointer(M);
631     new llvm::GlobalVariable(*M, Type, true,
        llvm::GlobalValue::ExternalLinkage,
632         nullptr, "frameGreaterEquals", nullptr,
633         llvm::GlobalValue::NotThreadLocal, 0,
        true);
634 }
635
636 static void addMatchErrorDecl(llvm::Module *M) {
637     auto FunRet = llvm::Type::getVoidTy(llvm::getGlobalContext());
638     llvm::SmallVector<llvm::Type *, 0> Arg;
639     auto FunType = llvm::FunctionType::get(FunRet, Arg, false);
640     auto F = llvm::Function::Create(FunType,
        llvm::GlobalValue::ExternalLinkage,
641         "matchError", M);
642     F->setCallingConv(llvm::CallingConv::C);
643     llvm::AttrBuilder B;
644     B.addAttribute(llvm::Attribute::NoReturn);
645     F->setAttributes(llvm::AttributeSet::get(
646         llvm::getGlobalContext(), llvm::AttributeSet::FunctionIndex,
        B));
647 }
648
649 static llvm::BasicBlock *
650 getMatchErrorBasicBlock(llvm::Module *M, const char *matchstr =
    "match_error") {
651     auto Ret = llvm::BasicBlock::Create(llvm::getGlobalContext(),
        matchstr);
652     auto ErrorFunc = M->getFunction("matchError");
653     assert(ErrorFunc);
654     llvm::SmallVector<llvm::Value *, 0> Arg;

```

```

655     auto Call = llvm::CallInst::Create(ErrorFunc, Arg, "", Ret);
656     Call->setTailCall();
657     llvm::ReturnInst::Create(M->getContext(), getZeroConst(M), Ret);
658     return Ret;
659 }
660
661 static llvm::LoadInst *addVolatileLoad(llvm::Value *Ptr,
662                                       const llvm::Twine &NameStr,
663                                       llvm::BasicBlock *B) {
664     #ifdef SSML_VOLATILE_MEMORY
665         return new llvm::LoadInst(Ptr, NameStr, true, B);
666     #else
667         return new llvm::LoadInst(Ptr, NameStr, false, B);
668     #endif
669 }
670
671 static llvm::StoreInst *addVolatileStore(llvm::Value *Src,
672                                         llvm::Value *Dst,
673                                         llvm::BasicBlock *B) {
674     #ifdef SSML_VOLATILE_MEMORY
675         return new llvm::StoreInst(Src, Dst, true, B);
676     #else
677         return new llvm::StoreInst(Src, Dst, false, B);
678     #endif
679 }
680
681 static llvm::Module *makeModule() {
682     auto Ret = new llvm::Module("llvm-ir.ll",
683                                llvm::getContext());
684     Ret->setDataLayout("e-m:e-i64:64-f80:128-n8:16:32:64-S128");
685     Ret->setTargetTriple("x86_64-unknown-linux-gnu");
686
687     // Add runtime forward declarations.
688     addGCRootDecl(Ret);
689     addGCWriteDecl(Ret);
690     addGCReadDecl(Ret);
691
692     addAllocateDecl(Ret);
693     addPrintDecl(Ret);
694     addDoarrayDecl(Ret);
695     addDoarraygetDecl(Ret);
696     addDoarrayupdateDecl(Ret);
697     addArrayemptyDecl(Ret);
698     addMatchErrorDecl(Ret);
699
700     addFrameArrayupdateDecl(Ret);
701
702     // addPatternCheckDecl(Ret);
703     addFirstGenBeginDecl(Ret);
704     addFirstGenEndDecl(Ret);
705     addMemoryBarrierDecl(Ret);
706     addMemoryBarrierDef(Ret);
707
708     addFramePrintDecl(Ret);
709     addFramePlusDecl(Ret);
710     addFrameMinusDecl(Ret);
711     addFrameMultiplyDecl(Ret);
712     addFrameDivisionDecl(Ret);
713     addFrameModuloDecl(Ret);
714     addFrameRefassignDecl(Ret);
715     addFrameArrayDecl(Ret);
716     addFrameArraygetDecl(Ret);

```

```

715     addFrameNegateDecl(Ret);
716     addFrameNotDecl(Ret);
717     addFrameRefDecl(Ret);
718     addFrameDerefDecl(Ret);
719     addFrameArraylengthDecl(Ret);
720
721     addFrameEqualsDecl(Ret);
722     addFrameNotEqualsDecl(Ret);
723     addFrameLessDecl(Ret);
724     addFrameGreaterDecl(Ret);
725     addFrameLessEqualsDecl(Ret);
726     addFrameGreaterEqualsDecl(Ret);
727
728     return Ret;
729 }
730
731 CodegenVisitor::CodegenVisitor() : Module(makeModule()) {}
732
733 CodegenVisitor::~CodegenVisitor() { delete this->mod(); }
734
735 void CodegenVisitor::doVisit(std::shared_ptr<ssml::ast::Node> N) {
736     N->accept(this);
737 }
738
739 void CodegenVisitor::enterScope() {
740     this->Names.enterScope();
741     this->Instances.enterScope();
742     this->peekFunc()->enterScope();
743 }
744
745 void CodegenVisitor::leaveScope(bool ZeroUnused) {
746     if (ZeroUnused) {
747         auto &Loc = this->peekFunc()->getScopeLocalIndices();
748
749         #if 0
750         auto FramePtr = this->peekFunc()->getLocalPtr();
751         auto Frame = new llvm::LoadInst(FramePtr, "",
752             this->BasicBlock);
753         #else
754         auto Frame = this->loadLocalPtr();
755         #endif
756         for (auto I : Loc) {
757             llvm::SmallVector<llvm::Value *, 1> Indices;
758             Indices.push_back(getConst(this->mod(), I));
759             auto Next = llvm::GetElementPtrInst::Create(
760                 getInteger64(this->mod()), Frame, Indices, "",
761                 this->BasicBlock);
762             #if 0
763             auto S = new llvm::StoreInst(getZeroConst(this->mod()),
764                 Next, false,
765                 this->BasicBlock);
766             S->setAlignment(alignment);
767             #else
768             this->addGCStore(getZeroConst(this->mod()), Next, Frame);
769             #endif
770         }
771         auto Temps = this->peekFunc()->getScopeTemps();
772         for (auto T : Temps) {
773             auto S = addVolatileStore(getZeroConst(this->mod()), T,
774                 this->BasicBlock);

```

```

773     S->setAlignment(alignment);
774 }
775 }
776
777 this->Names.leaveScope();
778 this->Instances.leaveScope();
779 this->peekFunc()->leaveScope();
780 }
781
782 void CodegenVisitor::nextTemp() {
783     auto F = this->peekFunc();
784
785     auto TAlloc = new
786         llvm::AllocaInst(getInteger64Pointer(this->mod()),
787             "temp_ptr",
788             F->getPostTempsInst());
789
790     TAlloc->setAlignment(alignment);
791 #if 0
792     auto Zstore = new llvm::StoreInst(
793         llvm::ConstantPointerNull::get(getInteger64Pointer(this->mod())),
794         TAlloc,
795         false, F->getPostTempsInst());
796     Zstore->setAlignment(alignment);
797 #endif
798
799 #if !defined(SSML_DISABLE_GC)
800     auto GC = this->mod()->getFunction("llvm.gcroot");
801     assert(GC);
802     llvm::SmallVector<llvm::Value *, 2> Args;
803     Args.push_back(new llvm::BitCastInst(
804         TAlloc, getPointerTo(getInteger(this->mod(),
805             8)), "",
806         F->getPostTempsInst()));
807     Args.push_back(
808         llvm::ConstantPointerNull::get(getPointerTo(getInteger(this->mod(),
809             8))));
810     auto Call = llvm::CallInst::Create(GC, Args, "",
811         F->getPostTempsInst());
812     Call->setTailCall();
813 #endif
814
815     this->Temp = new llvm::BitCastInst(TAlloc,
816         getInteger64Pointer(this->mod()),
817         "temp",
818         F->getPostTempsInst());
819
820     if (this->InsideLetDecls)
821         this->peekFunc()->appendTemp(this->Temp);
822     else
823         this->peekFunc()->appendTemp(this->Temp, 1);
824 }
825
826 void CodegenVisitor::storeNextTemp(llvm::Value *V) {
827     if (!this->LastExpr) {
828         this->nextTemp();
829         auto S = addVolatileStore(V, this->Temp, this->BasicBlock);
830         S->setAlignment(alignment);
831     } else {
832         this->Temp = V;
833     }
834 }

```

```

827 llvm::Value *CodegenVisitor::loadTemp(llvm::Value *Temp) {
828     if (!Temp)
829         Temp = this->Temp;
830     return addVolatileLoad(Temp, "", this->BasicBlock);
831 }
832
833 llvm::CallInst *CodegenVisitor::addAllocateCall(llvm::BasicBlock
834     *B,
835     llvm::Value *Size)
836     {
837     auto AllocFunc = this->mod()->getFunction("allocate");
838     assert(AllocFunc);
839     llvm::SmallVector<llvm::Value *, 1> Arg;
840     Arg.push_back(Size);
841     auto Ret = llvm::CallInst::Create(AllocFunc, Arg, "", B);
842     Ret->setTailCall();
843     return Ret;
844 }
845
846 llvm::Value *CodegenVisitor::addGCLoad(llvm::Value *Source,
847     llvm::Value *Base) {
848     if (Source->getType()->isPointerTy())
849         Source = new llvm::PtrToIntInst(Source,
850             getInteger64(this->mod()), "",
851             this->BasicBlock);
852
853     auto L = addVolatileLoad(
854         new llvm::IntToPtrInst(Source,
855             getInteger64Pointer(this->mod()), "",
856             this->BasicBlock),
857         "", this->BasicBlock);
858     return L;
859 }
860
861 #if 0
862     if (Base->getType()->isPointerTy())
863         Base = new llvm::PtrToIntInst(Base, getInteger64(this->mod()),
864             "",
865             this->BasicBlock);
866
867     auto F = this->mod()->getFunction("llvm.gcread");
868     llvm::SmallVector<llvm::Value *, 2> Args;
869     Args.push_back(new llvm::IntToPtrInst(
870         Base, getPointerTo(getInteger(this->mod(), 8)), "",
871         this->BasicBlock));
872     Args.push_back(new llvm::IntToPtrInst(
873         Source, getPointerTo(getInteger(this->mod(),
874             8)), "",
875         this->BasicBlock));
876     auto Call = llvm::CallInst::Create(F, Args, "",
877         this->BasicBlock);
878     Call->setTailCall();
879     return new llvm::PtrToIntInst(Call, getInteger64(this->mod()),
880         "gcread",
881         this->BasicBlock);
882 #endif
883 }
884
885 void CodegenVisitor::addGCStore(llvm::Value *Source, llvm::Value
886     *Dest,
887     llvm::Value *Base) {
888     if (Dest->getType()->isPointerTy())
889         Dest = new llvm::PtrToIntInst(Dest, getInteger64(this->mod()),
890             "",

```

```

877         this->BasicBlock);
878
879 #ifdef SSML_GENERATIONAL_GC
880     if (!Base->getType()->isPointerTy())
881         Base = new llvm::IntToPtrInst(Base,
882             getInteger64Pointer(this->mod()), "",
883             this->BasicBlock);
884     llvm::SmallVector<llvm::Value *, 1> Args;
885     Args.push_back(Base);
886     Args.push_back(Source);
887     auto F = this->mod()->getFunction("ssmlMemoryBarrier");
888     auto Call = llvm::CallInst::Create(F, Args, "",
889         this->BasicBlock);
890     Call->setTailCall();
891 #endif
892
893     auto D = new llvm::IntToPtrInst(Dest,
894         getInteger64Pointer(this->mod()), "",
895         this->BasicBlock);
896     auto S = addVolatileStore(Source, D, this->BasicBlock);
897     S->setAlignment(alignment);
898 }
899
900 void CodegenVisitor::addFuncEntry(FuncContext *F, bool IsMain,
901     bool PrevFrameIsLocals) {
902     auto FrameSize = F->getMaxLocalMemSize();
903     auto &EntryBB = F->func()->getEntryBlock();
904
905     auto Alloc = new
906         llvm::AllocaInst(getInteger64Pointer(this->mod()), "locals",
907             &EntryBB);
908     Alloc->setAlignment(alignment);
909     F->setLocalPtr(Alloc);
910
911     if (!IsMain) {
912 #ifdef SSML_GENERATIONAL_GC
913         this->nextTemp();
914 #endif
915         auto ArgIt = F->func()->arg_begin();
916         this->storeNextTemp(ArgIt);
917         F->setFirstArg(this->getExprResult());
918         this->storeNextTemp(++ArgIt);
919         F->setSecondArg(this->getExprResult());
920     } else {
921 #ifdef SSML_GENERATIONAL_GC
922         // Mark main as last stack frame to search for roots.
923         this->storeNextTemp(getConst(this->mod(), 1));
924 #endif
925     }
926
927 #if !defined(SSML_DISABLE_GC)
928     auto GC = this->mod()->getFunction("llvm.gcroot");
929     assert(GC);
930     llvm::SmallVector<llvm::Value *, 2> Args;
931     Args.push_back(new llvm::BitCastInst(
932         Alloc, getPointerTo(getPointerTo(getInteger(this->mod(),
933             8))), "",
934             this->BasicBlock));
935     Args.push_back(
936         llvm::ConstantPointerNull::get(getPointerTo(getInteger(this->mod(),
937             8))));

```

```

933     auto Call = llvm::CallInst::Create(GC, Args, "",
934         this->BasicBlock);
935     Call->setTailCall();
936 #endif
937     llvm::Value *LocalPtr;
938     if (!PrevFrameIsLocals) {
939         LocalPtr =
940             this->addAllocateCall(&EntryBB, getConst(this->mod(),
941                 FrameSize));
942         auto Store = addVolatileStore(LocalPtr, Alloc, &EntryBB);
943         Store->setAlignment(alignment);
944     } else {
945         llvm::Value *PrevFrame = this->loadTemp(F->getFirstArg());
946         LocalPtr = PrevFrame = new llvm::IntToPtrInst(
947             PrevFrame, getInteger64Pointer(this->mod()), "", &EntryBB);
948         auto Store = addVolatileStore(PrevFrame, Alloc, &EntryBB);
949         Store->setAlignment(alignment);
950     }
951 #if 0
952     {
953         llvm::SmallVector<llvm::Value *, 1> Arg;
954         Arg.push_back(new llvm::PtrToIntInst(Temps,
955             getInteger64(this->mod()), "",
956                 this->BasicBlock));
957         auto Fun = this->mod()->getFunction("print");
958         assert(Fun);
959         llvm::CallInst::Create(Fun, Arg, "", this->BasicBlock);
960     }
961     {
962         llvm::SmallVector<llvm::Value *, 1> Arg;
963         Arg.push_back(new llvm::PtrToIntInst(LocalPtr,
964             getInteger64(this->mod()),
965                 "", this->BasicBlock));
966         auto Fun = this->mod()->getFunction("print");
967         assert(Fun);
968         llvm::CallInst::Create(Fun, Arg, "", this->BasicBlock);
969     }
970 #endif
971 }
972 void CodegenVisitor::pushFunc(llvm::Function *F,
973     std::shared_ptr<FunctionVals> Vs,
974     const std::string &Name,
975     llvm::Function *CurrFunc) {
976     auto Next = new FuncContext(F, Vs, Name, CurrFunc);
977     Next->setPrev(this->FunContext);
978     this->FunContext = Next;
979     Next->setInsideLetDecl(this->InsideLetDecls);
980     this->InsideLetDecls = false;
981 }
982 void CodegenVisitor::changeFunc(llvm::Function *F) {
983     this->FunContext->func(F);
984 }
985 FuncContext *CodegenVisitor::peekFunc() { return this->FunContext;
986 }
987 void CodegenVisitor::popFunc() {
988     this->InsideLetDecls = this->peekFunc()->getInsideLetDecl();
989 }

```

```

990     auto Prev = this->peekFunc()->getPrev();
991     delete this->peekFunc();
992     this->FunContext = Prev;
993 }
994
995 void CodegenVisitor::beginMain(std::shared_ptr<FunctionVals> Vs) {
996     llvm::Module *M = this->mod();
997     auto FunRet = getInteger64(M);
998     auto T = llvm::FunctionType::get(FunRet, false);
999     auto F =
1000         llvm::Function::Create(T,
1001             llvm::GlobalValue::ExternalLinkage, "entry", M);
1002     F->setCallingConv(llvm::CallingConv::C);
1003     setGC(F);
1004     this->BasicBlock =
1005         llvm::BasicBlock::Create(M->getContext(), "entrylabel", F,
1006             0);
1007     this->pushFunc(F, Vs);
1008     this->addFuncEntry(this->peekFunc(), true, false);
1009 }
1010
1011 void CodegenVisitor::endMain() {
1012     auto M = this->mod();
1013     auto Main = this->peekFunc()->func();
1014     assert(!(Main->getName() == "entry"));
1015     (void)Main;
1016     auto Zero = getZeroConst(M);
1017     llvm::ReturnInst::Create(M->getContext(), Zero,
1018         this->BasicBlock);
1019     this->popFunc();
1020 }
1021
1022 llvm::Value *CodegenVisitor::addLocalLookup(llvm::BasicBlock *B,
1023     NameMap::ScopeValue
1024     SV) {
1025     assert(!SV.isTypeInstance());
1026     auto CurrScope = this->peekFunc()->getNest();
1027     auto LookupCount = CurrScope - SV.Scope;
1028     #if 0
1029     llvm::Value *Frame = this->peekFunc()->getLocalPtr();
1030     Frame = new llvm::LoadInst(Frame, "", B);
1031     #endif
1032     auto Frame = this->loadLocalPtr();
1033
1034     auto CurrFunc = this->peekFunc();
1035     for (int32_t I = 0; I < LookupCount; ++I) {
1036         llvm::SmallVector<llvm::Value *, 1> Indices1;
1037         Indices1.push_back(getConst(this->mod(),
1038             CurrFunc->getFramePtrIndex()));
1039         llvm::Value *Tmp = llvm::GetElementPtrInst::Create(
1040             getInteger64(this->mod()), Frame, Indices1, "", B);
1041         #if 0
1042         Tmp = new llvm::LoadInst(Tmp, "", B);
1043         #else
1044         Tmp = this->addGCLoad(Tmp, Frame);
1045         #endif
1046         Frame =
1047             new llvm::IntToPtrInst(Tmp,
1048                 getInteger64Pointer(this->mod()), "", B);
1049         CurrFunc = CurrFunc->getPrev();
1050     }
1051     // Should store Frame in temp storage, but a GC cannot be

```

```

1046         triggered here.
1047         llvm::SmallVector<llvm::Value *, 1> Indices;
1048         Indices.push_back(getConst(this->mod(), SV.Index));
1049         auto Ptr =
1050             llvm::GetElementPtrInst::Create(getInteger64(this->mod()),
1051                 Frame,
1052                 Indices, "", B);
1053     #if 0
1054         return new llvm::LoadInst(Ptr, "", B);
1055     #else
1056         return this->addGCLoad(Ptr, Frame);
1057     #endif
1058 }
1059
1060 void CodegenVisitor::visit(IntLiteral *N) {
1061     fatalExit("CodegenVisitor visit IntLiteral");
1062 }
1063
1064 void CodegenVisitor::visit(ShortIdentifier *N) {
1065     fatalExit("CodegenVisitor visit ShortIdentifier");
1066 }
1067
1068 void CodegenVisitor::visit(LongIdentifier *N) {
1069     fatalExit("CodegenVisitor visit LongIdentifier");
1070 }
1071
1072 void CodegenVisitor::visit(SeqShortIdentifier *N) {
1073     fatalExit("CodegenVisitor visit SeqShortIdentifier");
1074 }
1075
1076 void CodegenVisitor::visit(SeqLongIdentifier *N) {
1077     fatalExit("CodegenVisitor visit SeqLongIdentifier");
1078 }
1079
1080 void CodegenVisitor::visit(SeqExpression *N) {
1081     bool IsLast = this->LastExpr;
1082     this->LastExpr = false;
1083
1084     auto TB = this->TrueBlock;
1085     auto FB = this->FalseBlock;
1086     auto Size = N->size();
1087     for (size_t I = 0; I < Size; ++I) {
1088         if (I < Size - 1) {
1089             this->TrueBlock = nullptr;
1090             this->FalseBlock = nullptr;
1091         } else {
1092             this->LastExpr = IsLast;
1093             this->TrueBlock = TB;
1094             this->FalseBlock = FB;
1095         }
1096         this->doVisit((*N)[I]);
1097     }
1098     this->ExprWasTypeInstance = false;
1099 }
1100
1101 void CodegenVisitor::visit(LiteralExpression *N) {
1102     auto Val = N->getValue()->toInt();
1103     auto ExprResultTmp = getConst(this->mod(), markValue(Val));
1104     this->storeNextTemp(ExprResultTmp);
1105     this->ExprWasTypeInstance = false;
1106 }

```

```

1105 void CodegenVisitor::visit(LongIdentifierExpression *N) {
1106     NameMap::ScopeValue SV =
1107         this->Names.get(N->getIDs()->toString());
1108     if (SV.isTypeInstance()) {
1109         auto P = this->Instances.get(N->getIDs()->toString());
1110         llvm::Value *ExprResultTmp;
1111         if (P.second)
1112             ExprResultTmp = this->addAllocateCall(
1113                 this->BasicBlock, getConst(this->mod(), alignment * 2));
1114         else
1115             ExprResultTmp = this->addAllocateCall(this->BasicBlock,
1116                                                     getConst(this->mod(),
1117                                                         alignment));
1118         ExprResultTmp = new llvm::PtrToIntInst(
1119             ExprResultTmp, getInteger64(this->mod()), "",
1120             this->BasicBlock);
1121         this->storeNextTemp(ExprResultTmp);
1122     #if 0
1123         auto S = new llvm::StoreInst(getConst(this->mod(),
1124             markValue(P.first)),
1125                                     ExprResultTmp, false,
1126                                     this->BasicBlock);
1127         S->setAlignment(alignment);
1128     #else
1129         this->addGCStore(getConst(this->mod(), markValue(P.first)),
1130                         ExprResultTmp,
1131                         ExprResultTmp);
1132     #endif
1133     this->ExprWasTypeInstance = true;
1134 } else if (SV.isExisting()) {
1135     auto ExprResultTmp = this->addLocalLookup(this->BasicBlock,
1136         SV);
1137     this->storeNextTemp(ExprResultTmp);
1138     this->ExprWasTypeInstance = false;
1139 } else {
1140     auto ID = N->getIDs()->toString();
1141     llvm::Value *Val = getSimpleBuiltinConst(this->mod(), ID,
1142         true);
1143     if (Val) {
1144         this->storeNextTemp(Val);
1145         goto out;
1146     }
1147     Val = getBuiltin(this->mod(), ID);
1148     if (Val) {
1149         auto ExprResultTmp = new llvm::PtrToIntInst(
1150             Val, getInteger64(this->mod()), "", this->BasicBlock);
1151         this->storeNextTemp(ExprResultTmp);
1152         goto out;
1153     }
1154 }
1155
1156 NameMap::ScopeValue ScopeVal = {this->peekFunc()->getNest() +
1157     1, -1};
1158 auto Context = this->peekFunc();
1159 llvm::Value *CurrFunc = Context->currFunc();
1160 do {
1161     if (Context->currFuncName() == ID) {
1162         if (Context->getPrev() &&
1163             Context->getPrev()->getIsStepFunction()) {
1164             --ScopeVal.Scope;
1165             ScopeVal.Index = Context->getFramePtrIndex();
1166             Context = Context->getPrev();
1167         }

```

```

1157         break;
1158     }
1159     --ScopeVal.Scope;
1160     ScopeVal.Index = Context->getFramePtrIndex();
1161     Context = Context->getPrev();
1162     CurrFunc = Context->currFunc();
1163 } while (Context);
1164
1165 if (Context) {
1166     auto Result = this->addAllocateCall(this->BasicBlock,
1167                                         getConst(this->mod(), 2
1168                                                     * alignment));
1169
1170     auto ExprResultTmp = new llvm::PtrToIntInst(
1171         Result, getInteger64(this->mod()), "", this->BasicBlock);
1172     this->storeNextTemp(ExprResultTmp);
1173
1174     llvm::Value *Frame;
1175     if (ScopeVal.Index != -1) {
1176         Frame = this->addLocalLookup(this->BasicBlock, ScopeVal);
1177         Frame = new llvm::IntToPtrInst(Frame,
1178                                         getInteger64Pointer(this->mod()),
1179                                         "", this->BasicBlock);
1180     } else {
1181 #if 0
1182         Frame = new
1183             llvm::LoadInst(this->peekFunc()->getLocalPtr(), "",
1184                             this->BasicBlock);
1185 #else
1186         Frame = this->loadLocalPtr();
1187 #endif
1188     }
1189
1190     llvm::SmallVector<llvm::Value *, 1> Indices;
1191     Indices.push_back(getConst(this->mod(),
1192                                Context->getFramePtrIndex()));
1193     auto PrevFrameOffset = llvm::GetElementPtrInst::Create(
1194         getInteger64(this->mod()), Frame, Indices, "",
1195         this->BasicBlock);
1196
1197     // Should put frame in temp storage, but GC cannot be triggered
1198     // here.
1199 #if 0
1200     auto PrevFrameInt =
1201         new llvm::LoadInst(PrevFrameOffset, "",
1202                             this->BasicBlock);
1203 #else
1204     auto PrevFrameInt = this->addGCLoad(PrevFrameOffset, Frame);
1205 #endif
1206     auto Fun = new llvm::PtrToIntInst(CurrFunc,
1207                                       getInteger64(this->mod()), "",
1208                                       this->BasicBlock);
1209 #if 0
1210     auto Store = new llvm::StoreInst(Fun, Result, false,
1211                                       this->BasicBlock);
1212     Store->setAlignment(alignment);
1213 #else
1214     this->addGCStore(Fun, Result, Result);
1215 #endif
1216     Indices.clear();
1217     Indices.push_back(getConst(this->mod(), 1));
1218     auto Snd = llvm::GetElementPtrInst::Create(
1219         getInteger64(this->mod()), Result, Indices, "",
1220         this->BasicBlock);

```

```

1209 #if 0
1210     Store = new llvm::StoreInst(PrevFrameInt, Snd, false,
1211                                this->BasicBlock);
1212     Store->setAlignment(alignment);
1213 #else
1214     this->addGCStore(PrevFrameInt, Snd, Result);
1215 #endif
1216     goto out;
1217 }
1218 fatalExit("unexpected LongIdentifierExpression: " + ID);
1219 }
1220 out::;
1221 }
1222 void CodegenVisitor::visit(TupleExpression *N) {
1223     auto IsLast = this->LastExpr;
1224     this->LastExpr = false;
1225
1226     auto FB = this->FalseBlock;
1227     auto TB = this->TrueBlock;
1228
1229     auto Seq = N->getExprs();
1230
1231     llvm::Value *ExprResultTmp;
1232     llvm::Value *Ptr = nullptr;
1233     if (Seq->size()) {
1234         Ptr = addAllocateCall(this->BasicBlock,
1235                              getConst(this->mod(), Seq->size() *
1236                                       alignment));
1237         ExprResultTmp = new llvm::PtrToIntInst(Ptr,
1238         getInteger64(this->mod()), "",
1239                     this->BasicBlock);
1240     } else {
1241         ExprResultTmp = getZeroConst(this->mod());
1242     }
1243     this->storeNextTemp(ExprResultTmp);
1244     llvm::Value *Result = this->getExprResult();
1245
1246     size_t Idx = 0;
1247     for (auto E : *Seq) {
1248         this->TrueBlock = nullptr;
1249         this->FalseBlock = nullptr;
1250         this->doVisit(E);
1251
1252         llvm::SmallVector<llvm::Value *, 1> Indices;
1253         Indices.push_back(getConst(this->mod(), Idx));
1254         llvm::Value *Dst = this->loadTemp(Result);
1255         Dst = new llvm::IntToPtrInst(Dst,
1256                                     getInteger64Pointer(this->mod()), "",
1257                                     this->BasicBlock);
1258         auto Loc =
1259             llvm::GetElementPtrInst::Create(getInteger64(this->mod()),
1260             Dst,
1261             Indices, "",
1262             this->BasicBlock);
1263         this->addGCStore(this->loadTemp(), Loc, Dst);
1264         ++Idx;
1265     }
1266     this->LastExpr = IsLast;
1267     if (!IsLast)

```

```

1264     this->setExprResult(Result);
1265 else
1266     this->setExprResult(this->loadTemp(Result));
1267
1268     this->ExprWasTypeInstance = false;
1269     this->TrueBlock = TB;
1270     this->FalseBlock = FB;
1271 }
1272
1273 void CodegenVisitor::visit(ListExpression *N) {
1274     bool IsLast = this->LastExpr;
1275     this->LastExpr = false;
1276
1277     auto FB = this->FalseBlock;
1278     auto TB = this->TrueBlock;
1279
1280     llvm::SmallVector<llvm::Value *, 1> Indices;
1281     Indices.push_back(getConst(this->mod(), 1));
1282
1283     auto Seq = N->getExprs();
1284     llvm::Value *Result = nullptr;
1285     llvm::Value *First = nullptr;
1286     llvm::Value *Prev = nullptr;
1287     size_t Idx = 0;
1288     if (Seq->size()) {
1289         for (auto E : *Seq) {
1290             this->TrueBlock = nullptr;
1291             this->FalseBlock = nullptr;
1292             this->doVisit(E);
1293             auto Expr = this->loadTemp();
1294
1295             auto List = addAllocateCall(this->BasicBlock,
1296                                         getConst(this->mod(), 2 *
1297                                                     alignment));
1298
1299             auto ListInt = new llvm::PtrToIntInst(List,
1300                                                    getInteger64(this->mod(), ""),
1301                                                    this->BasicBlock);
1302
1303             this->storeNextTemp(ListInt);
1304             if (!First) {
1305                 First = ListInt;
1306                 Result = this->getExprResult();
1307             }
1308         }
1309     }
1310     #if 0
1311     auto Store = new llvm::StoreInst(getConst(this->mod(),
1312                                                markValue(0)),
1313                                     List, false,
1314                                     this->BasicBlock);
1315     Store->setAlignment(alignment);
1316 #else
1317     this->addGCStore(getConst(this->mod(), markValue(0)),
1318                     ListInt, ListInt);
1319 #endif
1320     auto Tup = addAllocateCall(this->BasicBlock,
1321                                getConst(this->mod(), 2 *
1322                                            alignment));
1323     auto TupInt = new llvm::PtrToIntInst(Tup,
1324                                           getInteger64(this->mod(), ""),
1325                                           this->BasicBlock);
1326     this->storeNextTemp(TupInt);
1327     auto NextPrev = this->getExprResult();
1328
1329     auto Snd = llvm::GetElementPtrInst::Create(

```

```

1319         getInteger64(this->mod()), List, Indices, "",
1320                 this->BasicBlock);
1321     #if 0
1322         Store = new llvm::StoreInst(TupInt, Snd, false,
1323                 this->BasicBlock);
1324         Store->setAlignment(alignment);
1325     #else
1326         this->addGCStore(TupInt, Snd, ListInt);
1327     #endif
1328     #if 0
1329         Store =
1330             new llvm::StoreInst(this->loadTemp(), Tup, false,
1331                 this->BasicBlock);
1332         Store->setAlignment(alignment);
1333     #else
1334         this->addGCStore(Expr, TupInt, TupInt);
1335     #endif
1336     if (Prev) {
1337         llvm::Value *Ptr = this->loadTemp(Prev);
1338         Ptr = new llvm::IntToPtrInst(Ptr,
1339             getInteger64Pointer(this->mod()), "",
1340             this->BasicBlock);
1341         Snd =
1342             llvm::GetElementPtrInst::Create(getInteger64(this->mod()),
1343                 Ptr,
1344                 Indices, "",
1345                 this->BasicBlock);
1346     #if 0
1347         Store = new llvm::StoreInst(ListInt, Snd, false,
1348                 this->BasicBlock);
1349         Store->setAlignment(alignment);
1350     #else
1351         this->addGCStore(ListInt, Snd, Ptr);
1352     #endif
1353     }
1354     if (Idx == Seq->size() - 1) {
1355         auto End =
1356             addAllocateCall(this->BasicBlock,
1357                 getConst(this->mod(), alignment));
1358         auto EndInt = new llvm::PtrToIntInst(End,
1359             getInteger64(this->mod()), "",
1360             this->BasicBlock);
1361         this->storeNextTemp(EndInt);
1362     #if 0
1363         Store = new llvm::StoreInst(getConst(this->mod(),
1364             markValue(1)), End,
1365             false, this->BasicBlock);
1366         Store->setAlignment(alignment);
1367     #else
1368         this->addGCStore(getConst(this->mod(), markValue(1)),
1369             EndInt, EndInt);
1370     #endif
1371     Snd =
1372         llvm::GetElementPtrInst::Create(getInteger64(this->mod()),
1373             Tup,
1374             Indices, "",
1375             this->BasicBlock);

```

```

1366 #if 0
1367     Store = new llvm::StoreInst(EndInt, Snd, false,
1368                               this->BasicBlock);
1369     Store->setAlignment(alignment);
1370 #else
1371     this->addGCStore(EndInt, Snd, TupInt);
1372 #endif
1373 }
1374     Prev = NextPrev;
1375     ++Idx;
1376 }
1377 } else {
1378     auto End =
1379         addAllocateCall(this->BasicBlock, getConst(this->mod(),
1380                                                    alignment));
1381     First = new llvm::PtrToIntInst(End, getInteger64(this->mod()),
1382                                   "",
1383                                   this->BasicBlock);
1384     this->storeNextTemp(First);
1385     Result = this->getExprResult();
1386 #if 0
1387     auto Store = new llvm::StoreInst(getConst(this->mod(),
1388                                                markValue(1)), End,
1389                                     false, this->BasicBlock);
1390     Store->setAlignment(alignment);
1391 #else
1392     this->addGCStore(getConst(this->mod(), markValue(1)), First,
1393                     First);
1394 #endif
1395 }
1396     if (!IsLast)
1397         this->setExprResult(Result);
1398     else
1399         this->setExprResult(this->loadTemp(Result));
1400     this->ExprWasTypeInstance = false;
1401     this->TrueBlock = TB;
1402     this->FalseBlock = FB;
1403 }
1404
1405 bool CodegenVisitor::tryBuiltinApplyExpr(ApplyExpression
1406                                          *ApplyExpr) {
1407     if (ApplyExpr->size() == 1) {
1408         auto IDExpr = (*ApplyExpr)[0]->asLongIdentifierExpression();
1409         if (!IDExpr)
1410             return false;
1411         auto ID = IDExpr->getIDs()->toString();
1412         if (getBuiltinValue(this->mod(), ID)) {
1413             bool IsLast = this->LastExpr;
1414             this->LastExpr = false;
1415             if (ID == "Array.empty") {
1416                 llvm::Value *ExprResultTmp =
1417                     this->mod()->getGlobalVariable("arrayempty");
1418                 ExprResultTmp = new llvm::PtrToIntInst(
1419                     ExprResultTmp, getInteger64(this->mod()), "",
1420                     this->BasicBlock);
1421                 this->LastExpr = IsLast;
1422                 this->storeNextTemp(ExprResultTmp);
1423             } else {

```

```

1421         fatalError("unexpected buintin value: " + ID);
1422     }
1423     return true;
1424 }
1425 return false;
1426 }
1427
1428 if (ApplyExpr->size() != 2)
1429     return false;
1430
1431 auto Left = (*ApplyExpr)[0];
1432 auto Right = (*ApplyExpr)[1];
1433 auto IDExpr = Left->asLongIdentifierExpression();
1434 if (!IDExpr)
1435     return false;
1436
1437 auto ID = IDExpr->getIDs()->toString();
1438
1439 auto SV = this->Names.get(ID);
1440 if (SV.isExisting())
1441     return false;
1442
1443 if (getTernaryBuintinFunc(this->mod(), ID)) {
1444     bool IsLast = this->LastExpr;
1445     this->LastExpr = false;
1446
1447     auto Tup = Right->asTupleExpression();
1448     assert(Tup);
1449     auto Seq = Tup->getExprs();
1450     assert(Seq->size() == 3);
1451
1452     this->doVisit((*Seq)[0]);
1453     auto LeftRes = this->getExprResult();
1454     this->doVisit((*Seq)[1]);
1455     auto MidRes = this->getExprResult();
1456     this->doVisit((*Seq)[2]);
1457     auto RightRes = this->getExprResult();
1458     if (ID == "Array.update") {
1459         auto Fun = this->mod()->getFunction("doarrayupdate");
1460         assert(Fun);
1461         llvm::SmallVector<llvm::Value *, 3> Args;
1462         Args.push_back(this->loadTemp(LeftRes));
1463         Args.push_back(this->loadTemp(MidRes));
1464         Args.push_back(this->loadTemp(RightRes));
1465         auto ExprResultTmp =
1466             llvm::CallInst::Create(Fun, Args, "", this->BasicBlock);
1467         ExprResultTmp->setTailCall();
1468         this->LastExpr = IsLast;
1469         this->storeNextTemp(ExprResultTmp);
1470     } else {
1471         fatalError("unexpected ternary buintin function: " + ID);
1472     }
1473 } else if (getBinaryBuintinFunc(this->mod(), ID)) {
1474     bool IsLast = this->LastExpr;
1475     this->LastExpr = false;
1476
1477     auto Tup = Right->asTupleExpression();
1478     assert(Tup);
1479     auto Seq = Tup->getExprs();
1480     assert(Seq->size() == 2);
1481
1482     this->doVisit((*Seq)[0]);

```

```

1483 auto LeftRes = this->getExprResult();
1484 this->doVisit((*Seq)[1]);
1485 auto RightRes = this->getExprResult();
1486 if (ID == "+") {
1487     LeftRes =
1488         unmarkValue(this->mod(), this->loadTemp(LeftRes),
1489                     this->BasicBlock);
1489     RightRes =
1490         unmarkValue(this->mod(), this->loadTemp(RightRes),
1491                     this->BasicBlock);
1491     llvm::Value *ExprResultTmp = llvm::BinaryOperator::CreateAdd(
1492         LeftRes, RightRes, "", this->BasicBlock);
1493     ExprResultTmp = markValue(this->mod(), ExprResultTmp,
1494                               this->BasicBlock);
1494     this->LastExpr = IsLast;
1495     this->storeNextTemp(ExprResultTmp);
1496 } else if (ID == "-") {
1497     LeftRes =
1498         unmarkValue(this->mod(), this->loadTemp(LeftRes),
1499                     this->BasicBlock);
1499     RightRes =
1500         unmarkValue(this->mod(), this->loadTemp(RightRes),
1501                     this->BasicBlock);
1501     llvm::Value *ExprResultTmp = llvm::BinaryOperator::CreateSub(
1502         LeftRes, RightRes, "", this->BasicBlock);
1503     ExprResultTmp = markValue(this->mod(), ExprResultTmp,
1504                               this->BasicBlock);
1504     this->LastExpr = IsLast;
1505     this->storeNextTemp(ExprResultTmp);
1506 } else if (ID == "*") {
1507     LeftRes =
1508         unmarkValue(this->mod(), this->loadTemp(LeftRes),
1509                     this->BasicBlock);
1509     RightRes =
1510         unmarkValue(this->mod(), this->loadTemp(RightRes),
1511                     this->BasicBlock);
1511     llvm::Value *ExprResultTmp = llvm::BinaryOperator::CreateMul(
1512         LeftRes, RightRes, "", this->BasicBlock);
1513     ExprResultTmp = markValue(this->mod(), ExprResultTmp,
1514                               this->BasicBlock);
1514     this->LastExpr = IsLast;
1515     this->storeNextTemp(ExprResultTmp);
1516 } else if (ID == "div") {
1517     LeftRes =
1518         unmarkValue(this->mod(), this->loadTemp(LeftRes),
1519                     this->BasicBlock);
1519     RightRes =
1520         unmarkValue(this->mod(), this->loadTemp(RightRes),
1521                     this->BasicBlock);
1521     llvm::Value *ExprResultTmp =
1522         llvm::BinaryOperator::CreateSDiv(
1523         LeftRes, RightRes, "", this->BasicBlock);
1523     ExprResultTmp = markValue(this->mod(), ExprResultTmp,
1524                               this->BasicBlock);
1524     this->LastExpr = IsLast;
1525     this->storeNextTemp(ExprResultTmp);
1526 } else if (ID == "mod") {
1527     LeftRes =
1528         unmarkValue(this->mod(), this->loadTemp(LeftRes),
1529                     this->BasicBlock);
1529     RightRes =
1530         unmarkValue(this->mod(), this->loadTemp(RightRes),

```

```

1531         this->BasicBlock);
1532     llvm::Value *ExprResultTmp =
1533         llvm::BinaryOperator::CreateSRem(
1534             LeftRes, RightRes, "", this->BasicBlock);
1535     ExprResultTmp = markValue(this->mod(), ExprResultTmp,
1536         this->BasicBlock);
1537     this->LastExpr = IsLast;
1538     this->storeNextTemp(ExprResultTmp);
1539 } else if (ID == "=") {
1540     auto B =
1541         new llvm::ICmpInst(*this->BasicBlock,
1542             llvm::CmpInst::ICMP_EQ,
1543             this->loadTemp(LeftRes),
1544             this->loadTemp(RightRes));
1545     llvm::Value *ExprResultTmp =
1546         llvm::CastInst::CreateIntegerCast(
1547             B, getInteger64(this->mod()), false, "",
1548             this->BasicBlock);
1549     ExprResultTmp = markValue(this->mod(), ExprResultTmp,
1550         this->BasicBlock);
1551     this->LastExpr = IsLast;
1552     this->storeNextTemp(ExprResultTmp);
1553 } else if (ID == "<") {
1554     auto B =
1555         new llvm::ICmpInst(*this->BasicBlock,
1556             llvm::CmpInst::ICMP_NE,
1557             this->loadTemp(LeftRes),
1558             this->loadTemp(RightRes));
1559     llvm::Value *ExprResultTmp =
1560         llvm::CastInst::CreateIntegerCast(
1561             B, getInteger64(this->mod()), false, "",
1562             this->BasicBlock);
1563     ExprResultTmp = markValue(this->mod(), ExprResultTmp,
1564         this->BasicBlock);
1565     this->LastExpr = IsLast;
1566     this->storeNextTemp(ExprResultTmp);
1567 } else if (ID == "<=") {
1568     auto B =
1569         new llvm::ICmpInst(*this->BasicBlock,
1570             llvm::CmpInst::ICMP_SLT,
1571             this->loadTemp(LeftRes),
1572             this->loadTemp(RightRes));
1573     llvm::Value *ExprResultTmp =
1574         llvm::CastInst::CreateIntegerCast(
1575             B, getInteger64(this->mod()), false, "",
1576             this->BasicBlock);
1577     ExprResultTmp = markValue(this->mod(), ExprResultTmp,
1578         this->BasicBlock);
1579     this->LastExpr = IsLast;
1580     this->storeNextTemp(ExprResultTmp);
1581 } else if (ID == ">") {
1582     auto B =
1583         new llvm::ICmpInst(*this->BasicBlock,
1584             llvm::CmpInst::ICMP_SGT,
1585             this->loadTemp(LeftRes),
1586             this->loadTemp(RightRes));
1587     llvm::Value *ExprResultTmp =
1588         llvm::CastInst::CreateIntegerCast(
1589             B, getInteger64(this->mod()), false, "",
1590             this->BasicBlock);
1591     ExprResultTmp = markValue(this->mod(), ExprResultTmp,
1592         this->BasicBlock);
1593     this->LastExpr = IsLast;
1594     this->storeNextTemp(ExprResultTmp);
1595 } else if (ID == ">=") {
1596     auto B =
1597         new llvm::ICmpInst(*this->BasicBlock,
1598             llvm::CmpInst::ICMP_SLE,
1599             this->loadTemp(LeftRes),
1600             this->loadTemp(RightRes));
1601     llvm::Value *ExprResultTmp =
1602         llvm::CastInst::CreateIntegerCast(
1603             B, getInteger64(this->mod()), false, "",
1604             this->BasicBlock);
1605     ExprResultTmp = markValue(this->mod(), ExprResultTmp,
1606         this->BasicBlock);
1607     this->LastExpr = IsLast;
1608     this->storeNextTemp(ExprResultTmp);
1609 }

```

```

1570     this->LastExpr = IsLast;
1571     this->storeNextTemp(ExprResultTmp);
1572 } else if (ID == "<=") {
1573     auto B =
1574         new llvm::ICmpInst(*this->BasicBlock,
1575                             llvm::CmpInst::ICMP_SLE,
1576                             this->loadTemp(LeftRes),
1577                             this->loadTemp(RightRes));
1578     llvm::Value *ExprResultTmp =
1579         llvm::CastInst::CreateIntegerCast(
1580             B, getInteger64(this->mod()), false, "",
1581             this->BasicBlock);
1582     ExprResultTmp = markValue(this->mod(), ExprResultTmp,
1583                               this->BasicBlock);
1584     this->LastExpr = IsLast;
1585     this->storeNextTemp(ExprResultTmp);
1586 } else if (ID == ">=") {
1587     auto B =
1588         new llvm::ICmpInst(*this->BasicBlock,
1589                             llvm::CmpInst::ICMP_SGE,
1590                             this->loadTemp(LeftRes),
1591                             this->loadTemp(RightRes));
1592     llvm::Value *ExprResultTmp =
1593         llvm::CastInst::CreateIntegerCast(
1594             B, getInteger64(this->mod()), false, "",
1595             this->BasicBlock);
1596     ExprResultTmp = markValue(this->mod(), ExprResultTmp,
1597                               this->BasicBlock);
1598     this->LastExpr = IsLast;
1599     this->storeNextTemp(ExprResultTmp);
1600 } else if (ID == "!=") {
1601     auto Ref = new llvm::IntToPtrInst(this->loadTemp(LeftRes),
1602                                       getInteger64Pointer(this->mod()),
1603                                       "",
1604                                       this->BasicBlock);
1605     #if 0
1606     auto S = new llvm::StoreInst(this->loadTemp(RightRes), Ref,
1607                                  false,
1608                                  this->BasicBlock);
1609     S->setAlignment(alignment);
1610     #else
1611     this->addGCStore(this->loadTemp(RightRes), Ref, Ref);
1612     #endif
1613     llvm::Value *ExprResultTmp = getZeroConst(this->mod());
1614     this->LastExpr = IsLast;
1615     this->storeNextTemp(ExprResultTmp);
1616 } else if (ID == "Array.array") {
1617     auto Fun = this->mod()->getFunction("doarray");
1618     assert(Fun);
1619     llvm::SmallVector<llvm::Value *, 2> Args;
1620     Args.push_back(this->loadTemp(LeftRes));
1621     Args.push_back(this->loadTemp(RightRes));
1622     auto ExprResultTmp =
1623         llvm::CallInst::Create(Fun, Args, "", this->BasicBlock);
1624     ExprResultTmp->setTailCall();
1625     this->LastExpr = IsLast;
1626     this->storeNextTemp(ExprResultTmp);
1627 } else if (ID == "Array.get") {
1628     auto Fun = this->mod()->getFunction("doarrayget");
1629     assert(Fun);
1630     llvm::SmallVector<llvm::Value *, 2> Args;
1631     Args.push_back(this->loadTemp(LeftRes));

```

```

1620     Args.push_back(this->loadTemp(RightRes));
1621     auto ExprResultTmp =
1622         llvm::CallInst::Create(Fun, Args, "", this->BasicBlock);
1623     ExprResultTmp->setTailCall();
1624     this->LastExpr = IsLast;
1625     this->storeNextTemp(ExprResultTmp);
1626 } else {
1627     fatalExit("unexpected binary buintin function: " + ID);
1628 }
1629 return true;
1630 } else if (getUnaryBuintinFunc(this->mod(), ID)) {
1631     bool IsLast = this->LastExpr;
1632     this->LastExpr = false;
1633
1634     this->doVisit(Right);
1635     auto Res = this->getExprResult();
1636     if (ID == "print") {
1637         llvm::SmallVector<llvm::Value *, 1> Args;
1638         Args.push_back(this->loadTemp(Res));
1639
1640         auto Fun = this->mod()->getFunction("print");
1641         assert(Fun);
1642         auto Call = llvm::CallInst::Create(Fun, Args, "",
1643             this->BasicBlock);
1644         Call->setTailCall();
1645         auto ExprResultTmp = new llvm::PtrToIntInst(
1646             Call, getInteger64(this->mod()), "", this->BasicBlock);
1647         this->LastExpr = IsLast;
1648         this->storeNextTemp(ExprResultTmp);
1649     } else if (ID == "~") {
1650         Res = unmarkValue(this->mod(), this->loadTemp(Res),
1651             this->BasicBlock);
1652         llvm::Value *ExprResultTmp =
1653             llvm::BinaryOperator::CreateNeg(Res, "",
1654                 this->BasicBlock);
1655         ExprResultTmp = markValue(this->mod(), ExprResultTmp,
1656             this->BasicBlock);
1657         this->LastExpr = IsLast;
1658         this->storeNextTemp(ExprResultTmp);
1659     } else if (ID == "not") {
1660         Res = unmarkValue(this->mod(), this->loadTemp(Res),
1661             this->BasicBlock);
1662         this->LastExpr = IsLast;
1663         this->nextTemp();
1664         this->LastExpr = false;
1665         auto Ptr = this->getExprResult();
1666         auto TB = llvm::BasicBlock::Create(llvm::getGlobalContext());
1667         auto FB = llvm::BasicBlock::Create(llvm::getGlobalContext());
1668         auto End =
1669             llvm::BasicBlock::Create(llvm::getGlobalContext());
1670         auto Cond = new llvm::ICmpInst(*this->BasicBlock,
1671             llvm::CmpInst::ICMP_EQ,
1672             Res, getConst(this->mod(),
1673                 1));
1674         llvm::BranchInst::Create(TB, FB, Cond, this->BasicBlock);
1675         TB->insertInto(this->peekFunc()->func());
1676         auto S = addVolatileStore(getConst(this->mod(),
1677             markValue(0), Ptr, TB);
1678         S->setAlignment(alignment);
1679         llvm::BranchInst::Create(End, TB);
1680         FB->insertInto(this->peekFunc()->func());

```

```

1673         S = addVolatileStore(getConst(this->mod(), markValue(1)),
1674                               Ptr, FB);
1675         S->setAlignment(alignment);
1676         llvm::BranchInst::Create(End, FB);
1677
1678         this->BasicBlock = End;
1679         this->BasicBlock->insertInto(this->peekFunc()->func());
1680         // this->nextTemp() is called and result is stored.
1681         this->LastExpr = IsLast;
1682     } else if (ID == "ref") {
1683         auto Ptr = this->addAllocateCall(this->BasicBlock,
1684                                         getConst(this->mod(),
1685                                                   alignment));
1686
1687         auto ExprResultTmp = new llvm::PtrToIntInst(
1688             Ptr, getInteger64(this->mod(), "", this->BasicBlock);
1689         this->LastExpr = IsLast;
1690         this->storeNextTemp(ExprResultTmp);
1691         this->LastExpr = false;
1692     #if 0
1693         auto S = new llvm::StoreInst(this->loadTemp(Res), Ptr, false,
1694                                     this->BasicBlock);
1695         S->setAlignment(alignment);
1696     #else
1697         this->addGCStore(this->loadTemp(Res), ExprResultTmp,
1698                         ExprResultTmp);
1699     #endif
1700     this->LastExpr = IsLast;
1701     } else if (ID == "Array.length") {
1702         auto Ptr = new llvm::IntToPtrInst(this->loadTemp(Res),
1703                                           getInteger64Pointer(this->mod(),
1704                                                             "",
1705                                                             this->BasicBlock);
1706     #if 0
1707         auto ExprResultTmp = new llvm::LoadInst(Ptr, "",
1708                                                  this->BasicBlock);
1709     #else
1710         auto ExprResultTmp = this->addGCLoad(Ptr, Ptr);
1711     #endif
1712     this->LastExpr = IsLast;
1713     this->storeNextTemp(ExprResultTmp);
1714     } else if (ID == "!") {
1715         auto Ptr = new llvm::IntToPtrInst(this->loadTemp(Res),
1716                                           getInteger64Pointer(this->mod(),
1717                                                             "",
1718                                                             this->BasicBlock);
1719     #if 0
1720         auto ExprResultTmp = new llvm::LoadInst(Ptr, "",
1721                                                  this->BasicBlock);
1722     #else
1723         auto ExprResultTmp = this->addGCLoad(Ptr, Ptr);
1724     #endif
1725     this->LastExpr = IsLast;
1726     this->storeNextTemp(ExprResultTmp);
1727     } else {
1728         fatalExit("unexpected unary builtin function: " + ID);
1729     }
1730     return true;
1731 }
1732
1733 return false;
1734 }

```

```

1728 void CodegenVisitor::visit(ApplyExpression *N) {
1729     if (this->tryBuiltinApplyExpr(N))
1730         return;
1731
1732     auto FB = this->FalseBlock;
1733     auto TB = this->TrueBlock;
1734
1735     bool IsLast = this->LastExpr;
1736     this->LastExpr = false;
1737
1738     size_t Count = 0;
1739     llvm::Value *First = nullptr;
1740     llvm::Value *Prev = nullptr;
1741     bool FirstWasInstance = false;
1742     for (size_t I = 0; I < N->size(); ++I) {
1743         auto C = (*N)[I];
1744
1745         if (N->size() == 1)
1746             this->LastExpr = IsLast;
1747
1748         this->TrueBlock = nullptr;
1749         this->FalseBlock = nullptr;
1750         this->ExprWasTypeInstance = false;
1751         this->doVisit(C);
1752         if (!Count) {
1753             First = this->getExprResult();
1754             FirstWasInstance = this->ExprWasTypeInstance;
1755         }
1756
1757         if (Count == 1 && FirstWasInstance) {
1758             auto Expr = new llvm::IntToPtrInst(this->loadTemp(First),
1759                                                 getInteger64Pointer(this->mod()),
1760                                                 "",
1761                                                 this->BasicBlock);
1762             llvm::SmallVector<llvm::Value *, 1> Indices;
1763             Indices.push_back(getConst(this->mod(), 1));
1764             auto Ptr = llvm::GetElementPtrInst::Create(
1765                 getInteger64(this->mod()), Expr, Indices, "",
1766                 this->BasicBlock);
1767
1768             #if 0
1769             auto S =
1770                 new llvm::StoreInst(this->loadTemp(), Ptr, false,
1771                                     this->BasicBlock);
1772             S->setAlignment(alignment);
1773             #else
1774             this->addGCStore(this->loadTemp(), Ptr, Expr);
1775             #endif
1776             this->setExprResult(First);
1777             assert(I == N->size() - 1);
1778             if (IsLast)
1779                 this->setExprResult(this->loadTemp());
1780         } else if (Count) {
1781             // Prev is a pointer to a tuple (i64, i64) where the first
1782             // is a pointer
1783             // to a function and the second is the pointer to the prev
1784             // environment.
1785             // The prev environment should be the first argument to the
1786             // function and
1787             // ExprResult should be second argument to the function.
1788             auto TupPtr = new llvm::IntToPtrInst(this->loadTemp(Prev),
1789                                                 getInteger64Pointer(this->mod()),
1790                                                 "",

```

```

1783                                     this->BasicBlock);
1784
1785 #if 0
1786     auto FunInt = new llvm::LoadInst(TupPtr, "",
1787                                     this->BasicBlock);
1788 #else
1789     auto FunInt = this->addGCLoad(TupPtr, TupPtr);
1790 #endif
1791     auto Fun = new llvm::IntToPtrInst(
1792         FunInt,
1793         llvm::PointerType::getUnqual(getStandardFuncType(this->mod())),
1794         "",
1795         this->BasicBlock);
1796
1797     llvm::SmallVector<llvm::Value *, 1> Indices;
1798     Indices.push_back(getConst(this->mod(), 1));
1799     auto EnvPtr = llvm::GetElementPtrInst::Create(
1800         getInteger64(this->mod()), TupPtr, Indices, "",
1801         this->BasicBlock);
1802
1803 #if 0
1804     auto Env = new llvm::LoadInst(EnvPtr, "", this->BasicBlock);
1805 #else
1806     auto Env = this->addGCLoad(EnvPtr, TupPtr);
1807 #endif
1808
1809     llvm::SmallVector<llvm::Value *, 2> Args;
1810     Args.push_back(Env);
1811     Args.push_back(this->loadTemp());
1812
1813     auto ExprResultTmp =
1814         llvm::CallInst::Create(Fun, Args, "", this->BasicBlock);
1815     ExprResultTmp->setCallingConv(llvm::CallingConv::C);
1816     ExprResultTmp->setTailCall();
1817     if (I == N->size() - 1)
1818         this->LastExpr = IsLast;
1819     this->storeNextTemp(ExprResultTmp);
1820 }
1821
1822     Prev = this->getExprResult();
1823     ++Count;
1824 }
1825
1826     if (Count > 1)
1827     {
1828         this->ExprWasTypeInstance = false;
1829         // else this->ExprWasTypeInstance is set.
1830         this->LastExpr = IsLast;
1831         this->TrueBlock = TB;
1832         this->FalseBlock = FB;
1833     }
1834 }
1835
1836 void CodegenVisitor::visit(OrElseExpression *N) {
1837     assert(!this->TrueBlock == !this->FalseBlock);
1838
1839     bool IsLast = this->LastExpr;
1840     this->LastExpr = false;
1841
1842     llvm::BasicBlock *TB = nullptr;
1843     llvm::BasicBlock *FB = nullptr;
1844     llvm::Value *Res = nullptr;
1845     if (!this->TrueBlock) {
1846         FB = llvm::BasicBlock::Create(llvm::getGlobalContext());
1847         TB = llvm::BasicBlock::Create(llvm::getGlobalContext());
1848         this->TrueBlock = TB;

```

```

1842     this->FalseBlock = FB;
1843     Res = new llvm::AllocaInst(getInteger64(this->mod()), "orres",
1844                                this->BasicBlock);
1845 }
1846 auto OrigT = this->TrueBlock;
1847 auto OrigF = this->FalseBlock;
1848
1849 auto TmpFB = llvm::BasicBlock::Create(llvm::getGlobalContext());
1850 this->FalseBlock = TmpFB;
1851 this->doVisit(N->getLeftExpr());
1852 auto Cond =
1853     new llvm::ICmpInst(*this->BasicBlock, llvm::CmpInst::ICMP_EQ,
1854                        this->loadTemp(), getConst(this->mod(),
1855                                                    markValue(1)));
1856
1857 llvm::BranchInst::Create(OrigT, TmpFB, Cond, this->BasicBlock);
1858
1859 this->BasicBlock = TmpFB;
1860 this->BasicBlock->insertInto(this->peekFunc()->func());
1861
1862 this->TrueBlock = OrigT;
1863 this->FalseBlock = OrigF;
1864 this->doVisit(N->getRightExpr());
1865 if (Res) {
1866     Cond = new llvm::ICmpInst(*this->BasicBlock,
1867                               llvm::CmpInst::ICMP_EQ,
1868                               this->loadTemp(),
1869                               getConst(this->mod(), markValue(1)));
1870     llvm::BranchInst::Create(OrigT, OrigF, Cond, this->BasicBlock);
1871
1872     auto Merge =
1873         llvm::BasicBlock::Create(llvm::getGlobalContext());
1874
1875     TB->insertInto(this->peekFunc()->func());
1876     auto Store = new llvm::StoreInst(getConst(this->mod(),
1877                                                markValue(1)), Res,
1878                                     false, TB);
1879     Store->setAlignment(alignment);
1880     llvm::BranchInst::Create(Merge, TB);
1881
1882     FB->insertInto(this->peekFunc()->func());
1883     Store = new llvm::StoreInst(getConst(this->mod(),
1884                                         markValue(0)), Res, false,
1885                                FB);
1886     Store->setAlignment(alignment);
1887     llvm::BranchInst::Create(Merge, FB);
1888
1889     this->BasicBlock = Merge;
1890     this->BasicBlock->insertInto(this->peekFunc()->func());
1891     this->LastExpr = IsLast;
1892     this->storeNextTemp(new llvm::LoadInst(Res, "",
1893                                           this->BasicBlock));
1894 }
1895
1896 this->LastExpr = IsLast;
1897 this->ExprWasTypeInstance = false;
1898 }
1899
1900 void CodegenVisitor::visit(AndAlsoExpression *N) {
1901     assert(!this->TrueBlock == !this->FalseBlock);
1902
1903     bool IsLast = this->LastExpr;
1904     this->LastExpr = false;

```

```

1898
1899     llvm::BasicBlock *TB = nullptr;
1900     llvm::BasicBlock *FB = nullptr;
1901     llvm::Value *Res = nullptr;
1902     if (!this->TrueBlock) {
1903         FB = llvm::BasicBlock::Create(llvm::getGlobalContext());
1904         TB = llvm::BasicBlock::Create(llvm::getGlobalContext());
1905         this->TrueBlock = TB;
1906         this->FalseBlock = FB;
1907         Res = new llvm::AllocaInst(getInteger64(this->mod(), "andres",
1908                                                this->BasicBlock));
1909     }
1910     auto OrigT = this->TrueBlock;
1911     auto OrigF = this->FalseBlock;
1912
1913     auto TmpTB = llvm::BasicBlock::Create(llvm::getGlobalContext());
1914     this->TrueBlock = TmpTB;
1915     this->doVisit(N->getLeftExpr());
1916     auto Cond =
1917         new llvm::ICmpInst(*this->BasicBlock, llvm::CmpInst::ICMP_EQ,
1918                            this->loadTemp(), getConst(this->mod(),
1919                                                         markValue(1)));
1919     llvm::BranchInst::Create(TmpTB, OrigF, Cond, this->BasicBlock);
1920
1921     this->BasicBlock = TmpTB;
1922     this->BasicBlock->insertInto(this->peekFunc()->func());
1923
1924     this->TrueBlock = OrigT;
1925     this->FalseBlock = OrigF;
1926     this->doVisit(N->getRightExpr());
1927     if (Res) {
1928         Cond = new llvm::ICmpInst(*this->BasicBlock,
1929                                    llvm::CmpInst::ICMP_EQ,
1930                                    this->loadTemp(),
1931                                    getConst(this->mod(), markValue(1)));
1932         llvm::BranchInst::Create(OrigT, OrigF, Cond, this->BasicBlock);
1933
1934         auto Merge =
1935             llvm::BasicBlock::Create(llvm::getGlobalContext());
1936
1937         TB->insertInto(this->peekFunc()->func());
1938         auto Store = new llvm::StoreInst(getConst(this->mod(),
1939                                                    markValue(1)), Res,
1940                                          false, TB);
1941         Store->setAlignment(alignment);
1942         llvm::BranchInst::Create(Merge, TB);
1943
1944         FB->insertInto(this->peekFunc()->func());
1945         Store = new llvm::StoreInst(getConst(this->mod(),
1946                                                    markValue(0)), Res, false,
1947                                     FB);
1948         Store->setAlignment(alignment);
1949         llvm::BranchInst::Create(Merge, FB);
1950
1951         this->BasicBlock = Merge;
1952         this->BasicBlock->insertInto(this->peekFunc()->func());
1953         this->LastExpr = IsLast;
1954         this->storeNextTemp(new llvm::LoadInst(Res, "",
1955                                                this->BasicBlock));
1956     }
1957     this->LastExpr = IsLast;

```

```

1954     this->ExprWasTypeInstance = false;
1955 }
1956
1957 void CodegenVisitor::visit(LetExpression *N) {
1958     auto FB = this->FalseBlock;
1959     auto TB = this->TrueBlock;
1960     this->enterScope();
1961
1962     bool IsLast = this->LastExpr;
1963     this->LastExpr = false;
1964
1965     bool PrevInLet = this->InsideLetDecls;
1966     this->InsideLetDecls = true;
1967     this->doVisit(N->getDecls());
1968     this->InsideLetDecls = PrevInLet;
1969
1970     this->LastExpr = IsLast;
1971     this->TrueBlock = TB;
1972     this->FalseBlock = FB;
1973     this->doVisit(N->getExprs());
1974
1975     this->leaveScope(!IsLast);
1976     this->ExprWasTypeInstance = false;
1977 }
1978
1979 void CodegenVisitor::visit(IfExpression *N) {
1980     auto FB = this->FalseBlock;
1981     auto TB = this->TrueBlock;
1982
1983     bool IsLast = this->LastExpr;
1984
1985     auto NTB = llvm::BasicBlock::Create(llvm::getGlobalContext());
1986     auto NFB = llvm::BasicBlock::Create(llvm::getGlobalContext());
1987     auto End = llvm::BasicBlock::Create(llvm::getGlobalContext());
1988
1989     auto Res = new llvm::AllocaInst(getInteger64(this->mod()),
1990                                     "ifres",
1991                                     this->BasicBlock);
1992
1993     this->LastExpr = false;
1994     this->TrueBlock = nullptr;
1995     this->FalseBlock = nullptr;
1996     this->doVisit(N->getCondExpr());
1997     auto Cond =
1998         new llvm::ICmpInst(*this->BasicBlock, llvm::CmpInst::ICMP_EQ,
1999                             this->loadTemp(), getConst(this->mod(),
2000                                                         markValue(1)));
2001     llvm::BranchInst::Create(NTB, NFB, Cond, this->BasicBlock);
2002
2003     this->TrueBlock = nullptr;
2004     this->FalseBlock = nullptr;
2005     this->BasicBlock = NTB;
2006     this->BasicBlock->insertInto(this->peekFunc()->func());
2007     this->LastExpr = IsLast;
2008     this->doVisit(N->getThenExpr());
2009     llvm::StoreInst *TStore;
2010     if (!IsLast)
2011         TStore =
2012             new llvm::StoreInst(this->loadTemp(), Res, false,
2013                                 this->BasicBlock);
2014     else
2015         TStore = new llvm::StoreInst(this->getExprResult(), Res, false,

```

```

2013         this->BasicBlock);
2014 TStore->setAlignment(alignment);
2015 llvm::BranchInst::Create(End, this->BasicBlock);
2016
2017 this->TrueBlock = nullptr;
2018 this->FalseBlock = nullptr;
2019 this->BasicBlock = NFB;
2020 this->BasicBlock->insertInto(this->peekFunc()->func());
2021 this->LastExpr = IsLast;
2022 this->doVisit(N->getElseExpr());
2023 llvm::StoreInst *FStore;
2024 if (!IsLast)
2025     FStore =
2026         new llvm::StoreInst(this->loadTemp(), Res, false,
                             this->BasicBlock);
2027 else
2028     FStore = new llvm::StoreInst(this->getExprResult(), Res, false,
2029                                 this->BasicBlock);
2030 FStore->setAlignment(alignment);
2031 llvm::BranchInst::Create(End, this->BasicBlock);
2032
2033 this->BasicBlock = End;
2034 this->BasicBlock->insertInto(this->peekFunc()->func());
2035
2036 this->LastExpr = IsLast;
2037 this->storeNextTemp(new llvm::LoadInst(Res, "",
2038                                         this->BasicBlock));
2039 this->ExprWasTypeInstance = false;
2040 this->TrueBlock = TB;
2041 this->FalseBlock = FB;
2042 }
2043
2044 void CodegenVisitor::visit(WhileExpression *N) {
2045     auto FB = this->FalseBlock;
2046     auto TB = this->TrueBlock;
2047
2048     bool IsLast = this->LastExpr;
2049     this->LastExpr = false;
2050
2051     auto Start = llvm::BasicBlock::Create(llvm::getGlobalContext());
2052     auto NTB = llvm::BasicBlock::Create(llvm::getGlobalContext());
2053     auto NFB = llvm::BasicBlock::Create(llvm::getGlobalContext());
2054
2055     llvm::BranchInst::Create(Start, this->BasicBlock);
2056
2057     this->TrueBlock = nullptr;
2058     this->FalseBlock = nullptr;
2059     this->BasicBlock = Start;
2060     this->BasicBlock->insertInto(this->peekFunc()->func());
2061     this->doVisit(N->getCondExpr());
2062     auto Cond =
2063         new llvm::ICmpInst(*this->BasicBlock, llvm::CmpInst::ICMP_EQ,
2064                             this->loadTemp(), getConst(this->mod(),
2065                                                         markValue(1)));
2066
2067     llvm::BranchInst::Create(NTB, NFB, Cond, this->BasicBlock);
2068
2069     this->TrueBlock = nullptr;
2070     this->FalseBlock = nullptr;
2071     this->BasicBlock = NTB;
2072     this->BasicBlock->insertInto(this->peekFunc()->func());
2073     this->doVisit(N->getBodyExpr());
2074     llvm::BranchInst::Create(Start, this->BasicBlock);

```

```

2072
2073     this->BasicBlock = NFB;
2074     this->BasicBlock->insertInto(this->peekFunc()->func());
2075
2076     this->LastExpr = IsLast;
2077     this->storeNextTemp(getZeroConst(this->mod()));
2078     this->ExprWasTypeInstance = false;
2079     this->TrueBlock = TB;
2080     this->FalseBlock = FB;
2081 }
2082
2083 void CodegenVisitor::visit(Match *N) {
2084     auto FB = this->FalseBlock;
2085
2086     this->peekFunc()->restoreRememberedContext();
2087     this->PatternExpr =
2088         this->loadTemp(this->peekFunc()->getSecondArg());
2089     this->TrueBlock =
2090         llvm::BasicBlock::Create(llvm::getGlobalContext());
2091     this->doVisit(N->getPattern());
2092     this->TrueBlock->insertInto(this->peekFunc()->func());
2093     this->BasicBlock = this->TrueBlock;
2094
2095     bool PrevLast = this->LastExpr;
2096     this->LastExpr = true;
2097     this->TrueBlock = nullptr;
2098     this->FalseBlock = nullptr;
2099     this->doVisit(N->getExpr());
2100     this->LastExpr = PrevLast;
2101
2102 #if 0
2103     {
2104         llvm::SmallVector<llvm::Value *, 1> Arg;
2105         Arg.push_back(this->ExprResultTmp);
2106         auto Fun = this->mod()->getFunction("print");
2107         assert(Fun);
2108         llvm::CallInst::Create(Fun, Arg, "", this->BasicBlock);
2109     }
2110 #endif
2111     llvm::ReturnInst::Create(this->mod()->getContext(),
2112                             this->getExprResult(),
2113                             this->BasicBlock);
2114     this->FalseBlock = FB;
2115 }
2116
2117 void CodegenVisitor::visit(SeqMatch *N) {
2118     auto Size = N->size();
2119     for (size_t I = 0; I < Size; ++I) {
2120         if (I == Size - 1)
2121             this->FalseBlock =
2122                 getMatchErrorBasicBlock(this->mod(),
2123                                         "lambda_match_fail");
2124         else
2125             this->FalseBlock =
2126                 llvm::BasicBlock::Create(llvm::getGlobalContext());
2127         this->doVisit((*N)[I]);
2128         this->FalseBlock->insertInto(this->peekFunc()->func());
2129         this->BasicBlock = this->FalseBlock;
2130     }
2131 #if 0
2132     llvm::ReturnInst::Create(this->mod()->getContext(),

```

```

2129         getZeroConst(this->mod()),
2130                     this->BasicBlock);
2131 #endif
2132 }
2133 void CodegenVisitor::visit(LambdaExpression *N) {
2134     static size_t NextLambdaNumber = 0;
2135     ++NextLambdaNumber;
2136
2137     bool IsLast = this->LastExpr;
2138     this->LastExpr = false;
2139
2140     auto OrigBB = this->BasicBlock;
2141     auto OrigTrueBB = this->TrueBlock;
2142     auto OrigFalseBB = this->FalseBlock;
2143
2144     // 1. Insert new function definition in module.
2145     // 2. Restore BBs.
2146     // 3. Allocate memory for function pointer and current frame.
2147     // 4. Put function pointer and frame in allocated memory.
2148     // 5. Let ExprResult = allocated memory.
2149
2150     auto FunType = getStandardFuncType(this->mod());
2151     auto Lambda = llvm::Function::Create(
2152         FunType, llvm::GlobalValue::ExternalLinkage,
2153         "_" + std::to_string(NextLambdaNumber), this->mod());
2154     Lambda->setCallingConv(llvm::CallingConv::C);
2155     setGC(Lambda);
2156
2157     auto EntryBB =
2158         llvm::BasicBlock::Create(this->mod()->getContext(),
2159                                 "entrylabel", Lambda);
2160     this->BasicBlock = EntryBB;
2161     this->TrueBlock = nullptr;
2162     this->FalseBlock = nullptr;
2163
2164     this->pushFunc(Lambda, N->getFunctionVals());
2165     this->addFuncEntry(this->peekFunc(), false, false);
2166     this->addLocalFuncVal("context",
2167                          this->loadTemp(this->peekFunc()->getFirstArg()));
2168     this->enterScope();
2169     this->doVisit(N->getCases());
2170     this->leaveScope(false);
2171     this->popFunc();
2172
2173     this->BasicBlock = OrigBB;
2174     this->TrueBlock = OrigTrueBB;
2175     this->FalseBlock = OrigFalseBB;
2176
2177     auto Result =
2178         this->addAllocateCall(this->BasicBlock,
2179                             getConst(this->mod(), 16));
2180     auto ExprResultTmp = new llvm::PtrToIntInst(Result,
2181         getInteger64(this->mod()),
2182         "",
2183         this->BasicBlock);
2184
2185     this->LastExpr = IsLast;
2186     this->storeNextTemp(ExprResultTmp);
2187     this->LastExpr = false;
2188
2189     auto Val = new llvm::PtrToIntInst(Lambda,
2190         getInteger64(this->mod()), "",

```

```

2185                                     this->BasicBlock);
2186
2187     auto Store = new llvm::StoreInst(Val, Result, false,
2188                                     this->BasicBlock);
2189     Store->setAlignment(alignment);
2190 #else
2191     this->addGCStore(Val, ExprResultTmp, ExprResultTmp);
2192 #endif
2193
2194     llvm::SmallVector<llvm::Value *, 1> Indices;
2195     Indices.push_back(getConst(this->mod(), 1));
2196     auto Ptr =
2197         llvm::GetElementPtrInst::Create(getInteger64(this->mod()),
2198                                     Result,
2199                                     Indices, "",
2200                                     this->BasicBlock);
2201
2202 #if 0
2203     llvm::Value *Locals =
2204         new llvm::LoadInst(this->peekFunc()->getLocalPtr(), "",
2205                             this->BasicBlock);
2206 #else
2207     llvm::Value *Locals = this->loadLocalPtr();
2208 #endif
2209 #endif
2210     Val = new llvm::PtrToIntInst(Locals, getInteger64(this->mod()),
2211                                 "",
2212                                 this->BasicBlock);
2213
2214 #if 0
2215     Store = new llvm::StoreInst(Val, Ptr, false, this->BasicBlock);
2216     Store->setAlingment(alignment);
2217 #else
2218     this->addGCStore(Val, Ptr, ExprResultTmp);
2219 #endif
2220
2221     // this->Temp is set from prev call to
2222     this->storeNextTemp(ExprResultTmp).
2223     this->LastExpr = IsLast;
2224     this->ExprWasTypeInstance = false;
2225 }
2226
2227 void CodegenVisitor::visit(SeqPattern *N) {
2228     for (auto &&C : *N)
2229         this->doVisit(C);
2230 }
2231
2232 void CodegenVisitor::visit(LiteralPattern *N) {
2233     auto Val = markValue(N->getValue()->toInt());
2234     // callPatternCheck(this->mod(), this->BasicBlock,
2235                         this->PatternExpr);
2236     auto Cond = new llvm::ICmpInst(*this->BasicBlock,
2237                                   llvm::CmpInst::ICMP_EQ,
2238                                   this->PatternExpr,
2239                                   getConst(this->mod(), Val));
2240     llvm::BranchInst::Create(this->TrueBlock, this->FalseBlock, Cond,
2241                             this->BasicBlock);
2242 }
2243
2244 void CodegenVisitor::visit(WildcardPattern *N) {
2245     llvm::BranchInst::Create(this->TrueBlock, this->BasicBlock);
2246 }
2247
2248 void CodegenVisitor::addLocalFuncVal(const std::string &Identifier,
2249                                     llvm::Value *Val) {

```

```

2237     this->peekFunc()->addLocal(Identifier);
2238     auto idx = this->peekFunc()->getLocalIndex(Identifier);
2239     llvm::SmallVector<llvm::Value *, 1> Indices;
2240     Indices.push_back(getConst(this->mod(), idx));
2241     #if 0
2242         llvm::Value *Loc =
2243             new llvm::LoadInst(this->peekFunc()->getLocalPtr(), "",
2244                               this->BasicBlock);
2245     #else
2246         llvm::Value *Loc = this->loadLocalPtr();
2247     #endif
2248     auto Ptr =
2249         llvm::GetElementPtrInst::Create(getInteger64(this->mod()),
2250                                         Loc,
2251                                         Indices, "",
2252                                         this->BasicBlock);
2253     // No need to store Loc in temp storage since GC cannot be
2254     // triggered here.
2255     #if 0
2256         auto Store = new llvm::StoreInst(Val, Ptr, false,
2257                                           this->BasicBlock);
2258         Store->setAlignment(alignment);
2259     #else
2260         this->addGCStore(Val, Ptr, Loc);
2261     #endif
2262     this->Names.insert(Identifier, {this->peekFunc()->getNest(),
2263                                   idx});
2264     this->peekFunc()->rememberContext();
2265     this->peekFunc()->appendLocalIndex(idx);
2266 }
2267
2268 void CodegenVisitor::insertTypeInstances() {
2269     this->Instances.insert(":", 0, true);
2270     this->Names.insert(":", {this->peekFunc()->getNest(), -1});
2271     this->Instances.insert("nil", 1, false);
2272     this->Names.insert("nil", {this->peekFunc()->getNest(), -1});
2273 }
2274
2275 void CodegenVisitor::visit(LongIdentifierPattern *N) {
2276     auto Identifier = N->getIDs()->toString();
2277     auto Name = this->Names.get(Identifier);
2278     if (Name.isTypeInstance()) {
2279         auto Struct = new llvm::IntToPtrInst(this->PatternExpr,
2280                                              getInteger64Pointer(this->mod()),
2281                                              "",
2282                                              this->BasicBlock);
2283     #if 0
2284         llvm::Value *Val = new llvm::LoadInst(Struct, "",
2285                                               this->BasicBlock);
2286     #else
2287         llvm::Value *Val = this->addGCLoad(Struct, Struct);
2288     #endif
2289     Val = unmarkValue(this->mod(), Val, this->BasicBlock);
2290     // callPatternCheck(this->mod(), this->BasicBlock, Val);
2291
2292     auto ConstValue = this->Instances.getValue(Identifier);
2293     auto Cond = new llvm::ICmpInst(*this->BasicBlock,
2294                                   llvm::CmpInst::ICMP_EQ,
2295                                   Val, getConst(this->mod(),
2296                                                ConstValue));
2297     llvm::BranchInst::Create(this->TrueBlock, this->FalseBlock,
2298                             Cond,

```

```

2287         this->BasicBlock);
2288     } else {
2289         auto Builtin = getSimpleBuiltinConst(this->mod(), Identifier,
2290             true);
2291         if (Builtin) {
2292             auto Cond = new llvm::ICmpInst(*this->BasicBlock,
2293                 llvm::CmpInst::ICMP_EQ,
2294                 this->PatternExpr, Builtin);
2295             llvm::BranchInst::Create(this->TrueBlock, this->FalseBlock,
2296                 Cond,
2297                 this->BasicBlock);
2298         } else {
2299             this->addLocalFuncVal(Identifier, this->PatternExpr);
2300             llvm::BranchInst::Create(this->TrueBlock, this->BasicBlock);
2301         }
2302     }
2303 }
2304
2305 void CodegenVisitor::visit(TypePattern *N) {
2306     this->doVisit(N->getPattern()); }
2307
2308 void CodegenVisitor::visit(ApplyPattern *N) {
2309     size_t Count = 0;
2310     auto OrigResult = this->PatternExpr;
2311     for (auto &&C : *N) {
2312         if (Count == 1) {
2313             this->BasicBlock = this->TrueBlock;
2314             this->BasicBlock->insertInto(this->peekFunc()->func());
2315             this->TrueBlock =
2316                 llvm::BasicBlock::Create(llvm::getGlobalContext());
2317             auto Expr = new llvm::IntToPtrInst(
2318                 OrigResult, getInteger64Pointer(this->mod()), "",
2319                 this->BasicBlock);
2320             llvm::SmallVector<llvm::Value *, 1> Indices;
2321             Indices.push_back(getConst(this->mod(), 1));
2322             auto Ptr = llvm::GetElementPtrInst::Create(
2323                 getInteger64(this->mod()), Expr, Indices, "",
2324                 this->BasicBlock);
2325
2326             #if 0
2327             this->PatternExpr = new llvm::LoadInst(Ptr, "",
2328                 this->BasicBlock);
2329             #else
2330             this->PatternExpr = this->addGCLoad(Ptr, Expr);
2331             #endif
2332         } else if (Count > 1) {
2333             fatalExit("too many apply pattern arguments");
2334         }
2335         this->doVisit(C);
2336         ++Count;
2337     }
2338 }
2339
2340 void CodegenVisitor::visit(ListPattern *N) {
2341     llvm::Value *ListExpr = new llvm::IntToPtrInst(
2342         this->PatternExpr, getInteger64Pointer(this->mod()), "",
2343         this->BasicBlock);
2344
2345     llvm::SmallVector<llvm::Value *, 1> Indices;
2346     Indices.push_back(getConst(this->mod(), 1));
2347
2348     size_t Idx = 0;
2349     auto OrigTrueBB = this->TrueBlock;

```



```

2341     auto Seq = N->getPatterns();
2342     for (auto P : *Seq) {
2343     #if 0
2344         llvm::Value *First = new llvm::LoadInst(ListExpr, "",
2345             this->BasicBlock);
2346     #else
2347         llvm::Value *First = this->addGCLoad(ListExpr, ListExpr);
2348     #endif
2349         First = unmarkValue(this->mod(), First, this->BasicBlock);
2350         // callPatternCheck(this->mod(), this->BasicBlock, First);
2351         auto Cond = new llvm::ICmpInst(*this->BasicBlock,
2352             llvm::CmpInst::ICMP_EQ,
2353             First, getConst(this->mod(),
2354                 0));
2355
2356         auto NextBlock =
2357             llvm::BasicBlock::Create(llvm::getGlobalContext());
2358
2359         llvm::BranchInst::Create(NextBlock, this->FalseBlock, Cond,
2360             this->BasicBlock);
2361
2362         NextBlock->insertInto(this->peekFunc()->func());
2363         this->BasicBlock = NextBlock;
2364
2365         llvm::SmallVector<llvm::Value *, 1> Indices;
2366         Indices.push_back(getConst(this->mod(), 1));
2367         llvm::Value *Tup = llvm::GetElementPtrInst::Create(
2368             getInteger64(this->mod()), ListExpr, Indices, "",
2369             this->BasicBlock);
2370
2371     #if 0
2372         Tup = new llvm::LoadInst(Tup, "", this->BasicBlock);
2373     #else
2374         Tup = this->addGCLoad(Tup, ListExpr);
2375     #endif
2376         Tup = new llvm::IntToPtrInst(Tup,
2377             getInteger64Pointer(this->mod()), "",
2378             this->BasicBlock);
2379
2380     #if 0
2381         this->PatternExpr = new llvm::LoadInst(Tup, "",
2382             this->BasicBlock);
2383     #else
2384         this->PatternExpr = this->addGCLoad(Tup, Tup);
2385     #endif
2386
2387         this->TrueBlock =
2388             llvm::BasicBlock::Create(llvm::getGlobalContext());
2389         this->doVisit(P);
2390         this->BasicBlock = this->TrueBlock;
2391         this->BasicBlock->insertInto(this->peekFunc()->func());
2392
2393         ListExpr =
2394             llvm::GetElementPtrInst::Create(getInteger64(this->mod()),
2395                 Tup,
2396                 Indices, "",
2397                 this->BasicBlock);
2398
2399     #if 0
2400         ListExpr = new llvm::LoadInst(ListExpr, "", this->BasicBlock);
2401     #else
2402         ListExpr = this->addGCLoad(ListExpr, Tup);
2403     #endif
2404         ListExpr = new llvm::IntToPtrInst(
2405             ListExpr, getInteger64Pointer(this->mod()), "",

```

```

2392         this->BasicBlock);
2393     ++Idx;
2394 }
2395
2396 #if 0
2397     llvm::Value *End = new llvm::LoadInst(ListExpr, "",
2398         this->BasicBlock);
2399 #else
2400     llvm::Value *End = this->addGCLoad(ListExpr, ListExpr);
2401 #endif
2402     End = unmarkValue(this->mod(), End, this->BasicBlock);
2403     // callPatternCheck(this->mod(), this->BasicBlock, End);
2404     auto Cond = new llvm::ICmpInst(*this->BasicBlock,
2405         llvm::CmpInst::ICMP_EQ, End,
2406         getConst(this->mod(), 1));
2407     this->TrueBlock = OrigTrueBB;
2408     llvm::BranchInst::Create(this->TrueBlock, this->FalseBlock, Cond,
2409         this->BasicBlock);
2410 }
2411
2412 void CodegenVisitor::visit(TuplePattern *N) {
2413     auto Seq = N->getPatterns();
2414     llvm::Value *TupExpr = nullptr;
2415     if (Seq->size())
2416         TupExpr = new llvm::IntToPtrInst(this->PatternExpr,
2417             getInteger64Pointer(this->mod()),
2418             "",
2419             this->BasicBlock);
2420
2421     size_t Idx = 0;
2422     auto OrigTrueBB = this->TrueBlock;
2423     for (auto P : *Seq) {
2424         llvm::SmallVector<llvm::Value *, 1> Indices;
2425         Indices.push_back(getConst(this->mod(), Idx));
2426         auto Ptr = llvm::GetElementPtrInst::Create(
2427             getInteger64(this->mod()), TupExpr, Indices, "",
2428             this->BasicBlock);
2429
2430         if (Idx < Seq->size() - 1)
2431             this->TrueBlock =
2432                 llvm::BasicBlock::Create(llvm::getGlobalContext());
2433         else
2434             this->TrueBlock = OrigTrueBB;
2435     }
2436     #if 0
2437     this->PatternExpr = new llvm::LoadInst(Ptr, "",
2438         this->BasicBlock);
2439 #else
2440     this->PatternExpr = this->addGCLoad(Ptr, TupExpr);
2441 #endif
2442     this->doVisit(P);
2443     if (Idx < Seq->size() - 1) {
2444         this->BasicBlock = this->TrueBlock;
2445         this->BasicBlock->insertInto(this->peekFunc()->func());
2446     }
2447     ++Idx;
2448 }
2449
2450 if (!Seq->size())
2451     llvm::BranchInst::Create(this->TrueBlock, this->BasicBlock);
2452 }

```

```

2447 void CodegenVisitor::visit(SeqDeclaration *N) {
2448     for (auto &&C : *N)
2449         this->doVisit(C);
2450 }
2451
2452 void CodegenVisitor::visit(Root *N) {
2453     beginMain(N->getFunctionVals());
2454     this->enterScope();
2455     this->addLocalFuncVal("context", getZeroConst(this->mod()));
2456     this->insertTypeInstances();
2457     for (auto &&C : *N)
2458         this->doVisit(C);
2459     this->leaveScope(false);
2460     endMain();
2461
2462     if (llvm::verifyModule(*this->mod(), &llvm::errs()))
2463         fatalExit("internal code generation error");
2464
2465     std::error_code Err;
2466     llvm::tool_output_file Out("llvm-bitcode.bc", Err,
2467                               llvm::sys::fs::F_None);
2468     if (Err)
2469         fatalExit("error opening output file");
2470     llvm::WriteBitcodeToFile(this->mod(), Out.os());
2471     Out.keep();
2472 }
2473
2474 void CodegenVisitor::visit(ValDeclaration *N) {
2475     this->TrueBlock = nullptr;
2476     this->FalseBlock = nullptr;
2477     this->doVisit(N->getSource());
2478     this->PatternExpr = this->loadTemp();
2479     this->FalseBlock = getMatchErrorBasicBlock(this->mod(),
2480                                               "val_match_fail");
2481     this->TrueBlock =
2482         llvm::BasicBlock::Create(llvm::getGlobalContext(),
2483                                 "val_true");
2484     assert(this->TrueBlock != this->FalseBlock);
2485     this->doVisit(N->getDest());
2486     this->FalseBlock->insertInto(this->peekFunc()->func());
2487     this->TrueBlock->insertInto(this->peekFunc()->func());
2488     this->BasicBlock = this->TrueBlock;
2489 }
2490
2491 void CodegenVisitor::visit(NonfixDeclaration *N) {
2492     // Nop.
2493 }
2494
2495 void CodegenVisitor::visit(InfixDeclaration *N) {
2496     // Nop.
2497 }
2498
2499 void CodegenVisitor::visit(InfixRDeclaration *N) {
2500     // Nop.
2501 }
2502
2503 void CodegenVisitor::visit(FunPatternDeclaration *N) {
2504     fatalExit("CodegenVisitor visit FunPatternDeclaration");
2505 }
2506
2507 static std::string functionOperatorToID(const std::string &FunID) {
2508     std::string Ret = "userop_";

```

```

2506     for (size_t I = 0; I < FunID.size(); ++I) {
2507         switch (FunID[0]) {
2508             case '!':
2509                 Ret += "ban";
2510                 break;
2511             case '%':
2512                 Ret += "per";
2513                 break;
2514             case '&':
2515                 Ret += "amp";
2516                 break;
2517             case '$':
2518                 Ret += "dol";
2519                 break;
2520             case '#':
2521                 Ret += "has";
2522                 break;
2523             case '+':
2524                 Ret += "plu";
2525                 break;
2526             case '-':
2527                 Ret += "min";
2528                 break;
2529             case '/':
2530                 Ret += "div";
2531                 break;
2532             case ':':
2533                 Ret += "col";
2534                 break;
2535             case '<':
2536                 Ret += "les";
2537                 break;
2538             case '=':
2539                 Ret += "equ";
2540                 break;
2541             case '>':
2542                 Ret += "gre";
2543                 break;
2544             case '?':
2545                 Ret += "que";
2546                 break;
2547             case '@':
2548                 Ret += "ats";
2549                 break;
2550             case '\\':
2551                 Ret += "bac";
2552                 break;
2553             case '~':
2554                 Ret += "til";
2555                 break;
2556             case '^':
2557                 Ret += "quo";
2558                 break;
2559             case '^':
2560                 Ret += "pow";
2561                 break;
2562             case '|':
2563                 Ret += "bar";
2564                 break;
2565             case '*':
2566                 Ret += "mul";
2567                 break;

```

```

2568         default:
2569             return FunID;
2570     }
2571 }
2572 return Ret;
2573 }
2574
2575 static std::string extractFunctionID(std::shared_ptr<Pattern>
    IDParams) {
2576     auto TypePat = IDParams->asTypePattern();
2577     assert(TypePat);
2578     auto ApplyPat = TypePat->getPattern()->asApplyPattern();
2579     assert(ApplyPat);
2580     auto IDPat = ApplyPat->front()->asLongIdentifierPattern();
2581     assert(IDPat);
2582     return IDPat->getIDs()->toString();
2583 }
2584
2585 static std::vector<std::shared_ptr<Pattern>>
2586 extractFunctionParams(std::shared_ptr<Pattern> IDParams) {
2587     auto TypePat = IDParams->asTypePattern();
2588     assert(TypePat);
2589     auto ApplyPat = TypePat->getPattern()->asApplyPattern();
2590     assert(ApplyPat);
2591     std::vector<std::shared_ptr<Pattern>> Ret(ApplyPat->begin(),
        ApplyPat->end());
2592     Ret.erase(Ret.begin());
2593     return Ret;
2594 }
2595
2596 static std::string getFunParamID(size_t I, size_t J) {
2597 #if 0
2598     return "_" + std::to_string(I) + "_" + std::to_string(J);
2599 #else
2600     return "_" + std::to_string(J) + "_";
2601 #endif
2602 }
2603
2604 void CodegenVisitor::visit(FunDeclaration *N) {
2605     auto OrigBB = this->BasicBlock;
2606     auto OrigTrueBB = this->TrueBlock;
2607     auto OrigFalseBB = this->FalseBlock;
2608
2609     bool PrevLast = this->LastExpr;
2610     this->LastExpr = false;
2611
2612     auto RealFunID = extractFunctionID(N->front()->getIDParams());
2613     auto FunID = functionOperatorToID(RealFunID);
2614     std::vector<std::vector<std::shared_ptr<Pattern>>> Params;
2615     for (auto P : *N)
2616         Params.push_back(extractFunctionParams(P->getIDParams()));
2617     auto ParamCount = Params.front().size();
2618     assert(ParamCount);
2619
2620     std::shared_ptr<FunctionVals> FunVals(new FunctionVals());
2621     for (size_t I = 0; I < Params.size(); ++I) {
2622         for (size_t J = 0; J < Params[I].size() - 1; ++J) {
2623             auto &V = Params[I];
2624             bool Val = V[J]->hasSimpleType();
2625             FunVals->addVal(!Val);
2626         }
2627         break;

```

```

2628     }
2629
2630     static size_t NextFunNumber = 0;
2631     ++NextFunNumber;
2632
2633     auto FunType = getStandardFuncType(this->mod());
2634     auto OuterFun = llvm::Function::Create(
2635         FunType, llvm::GlobalValue::ExternalLinkage,
2636         "_" + std::to_string(NextFunNumber) + FunID, this->mod());
2637     OuterFun->setCallingConv(llvm::CallingConv::C);
2638     setGC(OuterFun);
2639
2640     auto Fun = OuterFun;
2641     if (ParamCount - 1) {
2642         this->pushFunc(Fun, FunVals);
2643         this->peekFunc()->setIsStepFunction();
2644     }
2645     for (size_t J = 0; J < ParamCount - 1; ++J) {
2646         this->changeFunc(Fun);
2647         this->TrueBlock = nullptr;
2648         this->FalseBlock = nullptr;
2649         auto EntryBB = this->BasicBlock =
2650             llvm::BasicBlock::Create(this->mod()->getContext(),
2651                                     "entrylabel", Fun);
2652
2653         this->addFuncEntry(this->peekFunc(), false, J != 0);
2654         if (J == 0)
2655             this->addLocalFuncVal("context",
2656                                  this->loadTemp(this->peekFunc()->getFirstArg()));
2657
2658         llvm::Value *Param =
2659             this->loadTemp(this->peekFunc()->getSecondArg());
2660     #if 0
2661         for (size_t I = 0; I < Params.size(); ++I) {
2662             this->addLocalFuncVal(getFunParamID(I, J), Param);
2663         }
2664     #else
2665         this->addLocalFuncVal(getFunParamID(0, J), Param);
2666     #endif
2667
2668     ++NextFunNumber;
2669     Fun = llvm::Function::Create(FunType,
2670                                 llvm::GlobalValue::ExternalLinkage,
2671                                 "_" +
2672                                     std::to_string(NextFunNumber)
2673                                     + FunID,
2674                                 this->mod());
2675     Fun->setCallingConv(llvm::CallingConv::C);
2676     setGC(Fun);
2677
2678     // TODO. Find out if necessary to use gcread and gcwrite here.
2679     auto RetVal = this->addAllocateCall(EntryBB,
2680                                         getConst(this->mod(), 16));
2681     auto FunInt =
2682         new llvm::PtrToIntInst(Fun, getInteger64(this->mod()), "",
2683                                EntryBB);
2684     auto Store = addVolatileStore(FunInt, RetVal, EntryBB);
2685     Store->setAlignment(alignment);
2686
2687     llvm::SmallVector<llvm::Value *, 1> Indices;
2688     Indices.push_back(getConst(this->mod(), 1));
2689     auto Snd =

```

```

2683         llvm::GetElementPtrInst::Create(getInteger64(this->mod()),
                                          RetVal, Indices,
                                           "", EntryBB);
2684 #if 0
2685     llvm::Value *Context =
2686         new llvm::LoadInst(this->peekFunc()->getLocalPtr(), "",
                             EntryBB);
2687 #else
2688     llvm::Value *Context = this->loadLocalPtr();
2689 #endif
2690     Context =
2691         new llvm::PtrToIntInst(Context, getInteger64(this->mod()),
                                 "", EntryBB);
2692     Store = addVolatileStore(Context, Snd, EntryBB);
2693     Store->setAlignment(alignment);
2694
2695     auto Ret =
2696         new llvm::PtrToIntInst(RetVal, getInteger64(this->mod()),
                                 "", EntryBB);
2697     llvm::ReturnInst::Create(this->mod()->getContext(), Ret,
                              EntryBB);
2698 }
2699
2700 this->pushFunc(Fun, N->getFunctionVals(), RealFunID, OuterFun);
2701 this->BasicBlock =
2702     llvm::BasicBlock::Create(this->mod()->getContext(),
                              "entrylabel", Fun);
2703 this->TrueBlock = nullptr;
2704 this->FalseBlock = nullptr;
2705
2706 this->addFuncEntry(this->peekFunc(), false, false);
2707 this->addLocalFuncVal("context",
2708                     this->loadTemp(this->peekFunc()->getFirstArg()));
2709 this->enterScope();
2710
2711 for (size_t I = 0; I < Params.size(); ++I) {
2712     if (I == Params.size() - 1)
2713         this->FalseBlock = getMatchErrorBasicBlock(this->mod(),
                                                    "fun_match_fail");
2714     else
2715         this->FalseBlock =
2716             llvm::BasicBlock::Create(llvm::getGlobalContext());
2717     auto &ParamList = Params[I];
2718     for (size_t J = 0; J < ParamList.size(); ++J) {
2719         this->TrueBlock =
2720             llvm::BasicBlock::Create(llvm::getGlobalContext());
2721         if (J == ParamList.size() - 1) {
2722             this->PatternExpr =
2723                 this->loadTemp(this->peekFunc()->getSecondArg());
2724         } else {
2725             NameMap::ScopeValue SV = this->Names.get(getFunParamID(I,
2726                                                         J));
2727             assert(!SV.isTypeInstance());
2728             assert(SV.isExisting());
2729             this->PatternExpr = this->addLocalLookup(this->BasicBlock,
2730                                                     SV);
2731         }
2732     }
2733     this->doVisit(ParamList[J]);
2734     this->TrueBlock->insertInto(this->peekFunc()->func());

```

```

2732         this->BasicBlock = this->TrueBlock;
2733
2734         if (J == ParamList.size() - 1) {
2735             auto FB = this->FalseBlock;
2736             auto Expr = (*N)[I]->getDef();
2737             this->FalseBlock = nullptr;
2738             this->TrueBlock = nullptr;
2739             this->setExprResult(nullptr);
2740             this->LastExpr = true;
2741             this->doVisit(Expr);
2742             llvm::ReturnInst::Create(this->mod()->getContext(),
2743                                     this->getExprResult(),
2744                                     this->BasicBlock);
2745
2746             this->FalseBlock = FB;
2747         }
2748     }
2749     this->FalseBlock->insertInto(this->peekFunc()->func());
2750     this->BasicBlock = this->FalseBlock;
2751 }
2752 #if 0
2753     llvm::ReturnInst::Create(this->mod()->getContext(),
2754                             getZeroConst(this->mod()),
2755                             this->BasicBlock);
2756 #endif
2757     this->leaveScope(false);
2758
2759     if (ParamCount - 1)
2760         this->popFunc(); // pop inner.
2761     this->popFunc(); // pop outer.
2762
2763     this->BasicBlock = OrigBB;
2764     this->TrueBlock = OrigTrueBB;
2765     this->FalseBlock = OrigFalseBB;
2766
2767     this->LastExpr = PrevLast;
2768
2769     auto Ptr = this->addAllocateCall(this->BasicBlock,
2770                                     getConst(this->mod(), 16));
2771     auto PtrAsInt = new llvm::PtrToIntInst(Ptr,
2772                                             getInteger64(this->mod()), "",
2773                                             this->BasicBlock);
2774     this->storeNextTemp(PtrAsInt);
2775
2776     auto FunInt = new llvm::PtrToIntInst(OuterFun,
2777                                         getInteger64(this->mod()), "",
2778                                         this->BasicBlock);
2779 #if 0
2780     auto Store = new llvm::StoreInst(FunInt, Ptr, false,
2781                                     this->BasicBlock);
2782     Store->setAlignment(alignment);
2783 #else
2784     this->addGCStore(FunInt, PtrAsInt, PtrAsInt);
2785 #endif
2786
2787     llvm::SmallVector<llvm::Value *, 1> Indices;
2788     Indices.push_back(getConst(this->mod(), 1));
2789     auto Snd =
2790         llvm::GetElementPtrInst::Create(getInteger64(this->mod()),
2791                                         Ptr,

```

```

2786                                     Indices, "",
2787                                     this->BasicBlock);
2788
2789 #if 0
2790     llvm::Value *Context =
2791         new llvm::LoadInst(this->peekFunc()->getLocalPtr(), "",
2792                             this->BasicBlock);
2793 #else
2794     llvm::Value *Context = this->loadLocalPtr();
2795 #endif
2796     Context = new llvm::PtrToIntInst(Context,
2797         getInteger64(this->mod()), "",
2798         this->BasicBlock);
2799 #if 0
2800     Store = new llvm::StoreInst(Context, Snd, false,
2801         this->BasicBlock);
2802     Store->setAlignment(alignment);
2803 #else
2804     this->addGCStore(Context, Snd, PtrAsInt);
2805 #endif
2806     this->addLocalFuncVal(RealFunID, PtrAsInt);
2807 }
2808
2809 void CodegenVisitor::visit(DatatypeBareInstanceDeclaration *N) {
2810     this->Instances.insert(*N->getID()->getID(),
2811         this->DatatypeInstanceValue,
2812         false);
2813     this->Names.insert(*N->getID()->getID(),
2814         {this->peekFunc()->getNest(), -1});
2815 }
2816
2817 void CodegenVisitor::visit(DatatypeTypedInstanceDeclaration *N) {
2818     this->Instances.insert(*N->getID()->getID(),
2819         this->DatatypeInstanceValue,
2820         true);
2821     this->Names.insert(*N->getID()->getID(),
2822         {this->peekFunc()->getNest(), -1});
2823 }
2824
2825 void CodegenVisitor::visit(BareDatatypeDeclaration *N) {
2826     this->DatatypeTypepname = N->getID()->getID();
2827     this->DatatypeInstanceValue = 0;
2828     for (auto &&C : *N->getDef()) {
2829         this->doVisit(C);
2830         ++this->DatatypeInstanceValue;
2831     }
2832 }
2833
2834 void CodegenVisitor::visit(TypedDatatypeDeclaration *N) {
2835     this->DatatypeTypepname = N->getID()->getID();
2836     this->DatatypeInstanceValue = 0;
2837     for (auto &&C : *N->getDef()) {
2838         this->doVisit(C);
2839         ++this->DatatypeInstanceValue;
2840     }
2841 }
2842
2843 void CodegenVisitor::visit(LongIdentifierType *N) {
2844     // Nop.
2845 }
2846
2847 void CodegenVisitor::visit(VariableType *N) {
2848     // Nop.
2849 }

```

```

2840 }
2841
2842 void CodegenVisitor::visit(SeqType *N) {
2843     // Nop.
2844 }
2845
2846 void CodegenVisitor::visit(SeqVariableType *N) {
2847     // Nop.
2848 }
2849
2850 void CodegenVisitor::visit(TupleType *N) {
2851     // Nop.
2852 }
2853
2854 void CodegenVisitor::visit(ApplyType *N) {
2855     // Nop.
2856 }
2857
2858 void CodegenVisitor::visit(ProductType *N) {
2859     // Nop.
2860 }
2861
2862 void CodegenVisitor::visit(FunctionType *N) {
2863     // Nop.
2864 }

```

Listing 44: lib/Codegen/NameMap.cpp

```

1  #include "NameMap.h"
2
3  #include <cassert>
4
5  using namespace ssml::codegen;
6
7  void NameMap::insert(const std::string &K, ScopeValue V) {
8      Maps.back()[K] = V;
9  }
10
11 NameMap::ScopeValue NameMap::get(const std::string &K) {
12     for (auto It = Maps.rbegin(), End = Maps.rend(); It != End; ++It)
13         if (It->count(K))
14             return It->find(K)->second;
15     return {-1, -1};
16 }
17
18 void NameMap::enterScope() {
19     Maps.push_back(MapType());
20 }
21
22 void NameMap::leaveScope() {
23     Maps.pop_back();
24 }

```

Listing 45: lib/Common/ErrorMessage.cpp

```

1  #include "ssml/Common/ErrorMessage.h"
2
3  #include "ssml/Common/CommandLine.h"
4
5  #include "llvm/Support/raw_ostream.h"
6  #include "llvm/ADT/StringRef.h"

```

```

7
8 #include <cstdlib>
9
10 using namespace ssml;
11 using namespace llvm;
12
13 void ssml::errorExit(StringRef FileName, SourceLocation Loc,
14                     llvm::StringRef Msg) {
15     errs() << FileName << ':' << Loc << ": error: " << Msg << '\n';
16     exit(1);
17 }
18
19 void ssml::errorExit(SourceLocation Loc, llvm::StringRef Msg) {
20     return errorExit(*Commandline.getFilename(), Loc, Msg);
21 }

```

Listing 46: lib/Common/Compare.cpp

```

1 #include "ssml/Common/Compare.h"
2
3 #include <string>
4
5 namespace ssml {
6     template struct SharedPtrLess<std::string>;
7 } // End namespace ssml.

```

Listing 47: lib/Common/Commandline.cpp

```

1 #include "ssml/Common/Commandline.h"
2
3 using namespace ssml;
4
5 CommandLineImpl ssml::Commandline;
6
7 static std::shared_ptr<std::string> StdinFilename(new
8     std::string("<stdin>"));
9
10 const std::shared_ptr<std::string> CommandLineImpl::getFilename()
11     const {
12     return StdinFilename;
13 }

```

Listing 48: lib/Common/ErrorExit.cpp

```

1 #include "ssml/Common/FatalExit.h"
2
3 #include "llvm/Support/raw_ostream.h"
4
5 #include <exception>
6
7 using namespace ssml;
8 using namespace llvm;
9
10 void ssml::fatalExit(Twine Msg) {
11     errs() << "fatal error: " << Msg << '\n';
12     std::terminate();
13 }
14
15 void ssml::fatalExitOutOfMem() {

```

```

16     fatalError("out of memory");
17 }

```

Listing 49: lib/Common/StringManipulation.cpp

```

1  #include "ssml/Common/StringManipulation.h"
2
3  #include "ssml/Common/FatalExit.h"
4
5  #include <cstring>
6
7  using namespace ssml;
8
9  char *ssml::duplicate(const char *str) {
10     char *dup = strdup(str);
11     if (!dup)
12         fatalErrorOutOfMem();
13     return dup;
14 }

```

Listing 50: lib/Common/SourceLocation.cpp

```

1  #include "ssml/Common/SourceLocation.h"
2
3  #include "llvm/Support/raw_ostream.h"
4
5  #include <sstream>
6
7  using namespace llvm;
8  using namespace ssml;
9
10 std::string SourceLocation::toString() const {
11     return std::to_string(this->Line) + ':' +
12         std::to_string(this->Column);
13 }
14 raw_ostream &llvm::operator<<(raw_ostream &Out, SourceLocation L) {
15     return Out << L.toString();
16 }

```

Listing 51: lib/AST/Type.cpp

```

1  #include "ssml/AST/Type.h"
2
3  #include "ssml/AST/Visitor.h"
4  #include "ssml/AST/Identifier.h"
5
6  #include <cassert>
7
8  using namespace ssml::ast;
9
10 Type::Type(SourceLocation L) : Node(L) {}
11
12 TupleType *Type::asTupleType() { return nullptr; }
13
14 LongIdentifierType::LongIdentifierType(SourceLocation L,
15                                         std::shared_ptr<LongIdentifier>
16                                             ID)
17     : Type(L), ID(ID) {}

```

```

17 void LongIdentifierType::accept(Visitor *V) { V->visit(this); }
18
19
20 VariableType::VariableType(SourceLocation L,
21                             std::shared_ptr<ShortIdentifier> ID)
22     : Type(L), ID(ID) {}
23
24 void VariableType::accept(Visitor *V) { V->visit(this); }
25
26 SeqType::SeqType(SourceLocation L, std::shared_ptr<Type> F)
27     : Type(L), std::vector<std::shared_ptr<Type>>{F} {}
28
29 void SeqType::accept(Visitor *V) { V->visit(this); }
30
31 SeqVariableType::SeqVariableType(SourceLocation L,
32                                   std::shared_ptr<VariableType> F)
33     : Type(L), std::vector<std::shared_ptr<VariableType>>{F} {}
34
35 void SeqVariableType::accept(Visitor *V) { V->visit(this); }
36
37 TupleType::TupleType(SourceLocation L, std::shared_ptr<SeqType>
38                       Types)
39     : Type(L), Types(Types) {}
40
41 void TupleType::accept(Visitor *V) { V->visit(this); }
42
43 TupleType *TupleType::asTupleType() { return this; }
44
45 IdentifierColonType::IdentifierColonType(SourceLocation L,
46                                           std::shared_ptr<Identifier>
47                                           ID,
48                                           std::shared_ptr<Type> T)
49     : Type(L), ID(ID), Typ(T) {}
50
51 void IdentifierColonType::accept(Visitor *V) { V->visit(this); }
52
53 RecordType::RecordType(SourceLocation L,
54                         std::shared_ptr<IdentifierColonType> F)
55     : Type(L),
56       std::vector<std::shared_ptr<IdentifierColonType>>{F} {}
57
58 void RecordType::accept(Visitor *V) { V->visit(this); }
59
60 ApplyType::ApplyType(SourceLocation L, std::shared_ptr<Type> F)
61     : Type(L), std::vector<std::shared_ptr<Type>>{F} {}
62
63 void ApplyType::accept(Visitor *V) { V->visit(this); }
64
65 ProductType::ProductType(SourceLocation L, std::shared_ptr<Type> F)
66     : Type(L), std::vector<std::shared_ptr<Type>>{F} {}
67
68 void ProductType::accept(Visitor *V) { V->visit(this); }
69
70 FunctionType::FunctionType(SourceLocation L, std::shared_ptr<Type>
71                             F)
72     : Type(L), std::vector<std::shared_ptr<Type>>{F} {}
73
74 void FunctionType::accept(Visitor *V) { V->visit(this); }

```

Listing 52: lib/AST/Literal.cpp

```

1  #include "ssml/AST/Literal.h"
2  #include "ssml/AST/Visitor.h"
3
4  #include "ssml/Common/ErrorMessage.h"
5  #include "ssml/Common/FatalExit.h"
6
7  #include "llvm/ADT/StringRef.h"
8
9  #include <cstdlib>
10 #include <cassert>
11 #include <cerrno>
12
13 using namespace ssml;
14 using namespace ssml::ast;
15
16 Literal::Literal(SourceLocation L, std::shared_ptr<std::string>
    Value)
17     : Node(L), Value(Value) {}
18
19 IntLiteral::IntLiteral(SourceLocation L,
    std::shared_ptr<std::string> Value)
20     : Literal(L, Value), IntValue(0) {
21     const char *RawVal = Value->c_str();
22     size_t Len = Value->size();
23     int Base;
24     if (Len > 2 && RawVal[1] == 'x')
25         Base = 16;
26     else
27         Base = 10;
28
29     errno = 0;
30     long long V = std::strtoll(RawVal, nullptr, Base);
31     if (errno == ERANGE) {
32         errorExit(this->getLocation(), "int literal out of bounds: " +
            *Value);
33     } else {
34         assert(!errno);
35         this->IntValue = V;
36     }
37 }
38
39 void IntLiteral::accept(Visitor *V) { V->visit(this); }
40
41 std::int64_t IntLiteral::getIntValue() const { return
    this->IntValue; }
42
43 int64_t IntLiteral::toInt() { return this->getIntValue(); }
44
45 RealLiteral::RealLiteral(SourceLocation L,
    std::shared_ptr<std::string> Value)
46     : Literal(L, Value) {}
47
48 void RealLiteral::accept(Visitor *V) { V->visit(this); }
49
50 int64_t RealLiteral::toInt() {
51     fatalExit("cannot convert real to int");
52     return 0;
53 }
54
55 CharLiteral::CharLiteral(SourceLocation L,
    std::shared_ptr<std::string> Value)
56     : Literal(L, Value) {}

```

```

57
58 void CharLiteral::accept(Visitor *V) { V->visit(this); }
59
60 int64_t CharLiteral::toInt() {
61     fatalExit("cannot convert char to int");
62     return 0;
63 }
64
65 StringLiteral::StringLiteral(SourceLocation L,
66                             std::shared_ptr<std::string> Value)
67     : Literal(L, Value) {}
68
69 void StringLiteral::accept(Visitor *V) { V->visit(this); }
70
71 int64_t StringLiteral::toInt() {
72     fatalExit("cannot convert string to int");
73     return 0;
74 }

```

Listing 53: lib/AST/Identifier.cpp

```

1  #include "ssml/AST/Identifier.h"
2
3  #include "ssml/AST/Visitor.h"
4  #include "ssml/AST/Literal.h"
5
6  #include <sstream>
7  #include <cassert>
8
9  using namespace ssml::ast;
10
11 Identifier::Identifier(SourceLocation L) : Node(L) {}
12
13 ShortIdentifier::ShortIdentifier(SourceLocation L,
14                                std::shared_ptr<std::string> ID)
15     : Identifier(L), ID(ID) {}
16
17 void ShortIdentifier::accept(Visitor *V) { V->visit(this); }
18
19 std::string ShortIdentifier::toString() { return *this->getID(); }
20
21 LongIdentifier::LongIdentifier(SourceLocation L,
22                               std::shared_ptr<ShortIdentifier>
23                               First)
24     : Identifier(L),
25       std::vector<std::shared_ptr<ShortIdentifier>>{First} {}
26
27 void LongIdentifier::accept(Visitor *V) { V->visit(this); }
28
29 std::string LongIdentifier::toString() {
30     std::stringstream S;
31
32     auto Stop = this->size();
33     assert(Stop);
34     --Stop;
35
36     size_t I;
37     for (I = 0; I < Stop; ++I)
38         S <<>(*this)[I]->getID() << ' ';
39     S <<(*this)[I]->getID();

```

```

39     return S.str();
40 }
41
42 IntIdentifier::IntIdentifier(SourceLocation L,
43     std::shared_ptr<IntLiteral> Lit)
44     : Identifier(L), Literal(Lit) {}
45
46 void IntIdentifier::accept(Visitor *V) { V->visit(this); }
47
48 std::string IntIdentifier::toString() {
49     return *this->getLiteral()->getValueString();
50 }
51
52 SeqShortIdentifier::SeqShortIdentifier(SourceLocation L,
53     std::shared_ptr<ShortIdentifier>
54         First)
55     : Identifier(L),
56       std::vector<std::shared_ptr<ShortIdentifier>>{First} {}
57
58 void SeqShortIdentifier::accept(Visitor *V) { V->visit(this); }
59
60 std::string SeqShortIdentifier::toString() {
61     std::stringstream S;
62
63     auto Stop = this->size();
64     assert(Stop);
65     --Stop;
66
67     size_t I;
68     for (I = 0; I < Stop; ++I)
69         S << (*this)[I]->getID() << ' ';
70     S << (*this)[I]->getID();
71
72     return S.str();
73 }
74
75 SeqLongIdentifier::SeqLongIdentifier(SourceLocation L,
76     std::shared_ptr<LongIdentifier>
77         First)
78     : Identifier(L),
79       std::vector<std::shared_ptr<LongIdentifier>>{First} {}
80
81 void SeqLongIdentifier::accept(Visitor *V) { V->visit(this); }
82
83 std::string SeqLongIdentifier::toString() {
84     std::stringstream S;
85
86     auto Stop = this->size();
87     assert(Stop);
88     --Stop;
89
90     size_t I;
91     for (I = 0; I < Stop; ++I)
92         S << (*this)[I]->toString() << ' ';
93     S << (*this)[I]->toString();
94
95     return S.str();
96 }

```

Listing 54: lib/AST/Match.cpp

```

1 #include "ssml/AST/Match.h"
2
3 #include "ssml/AST/Pattern.h"
4 #include "ssml/AST/Expression.h"
5 #include "ssml/AST/Visitor.h"
6
7 using namespace ssml::ast;
8
9 Match::Match(SourceLocation L, std::shared_ptr<Pattern> P,
10             std::shared_ptr<Expression> E)
11     : Node(L), Pat(P), Expr(E) {}
12
13 void Match::accept(Visitor *V) { V->visit(this); }
14
15 SeqMatch::SeqMatch(SourceLocation L, std::shared_ptr<Match> F)
16     : Node(L), std::vector<std::shared_ptr<Match> >{F} {}
17
18 void SeqMatch::accept(Visitor *V) { V->visit(this); }

```

Listing 55: lib/AST/Visitor.cpp

```

1 #include "ssml/AST/Visitor.h"
2
3 #include "ssml/Common/FatalExit.h"
4
5 using namespace ssml;
6 using namespace ssml::ast;
7
8 Visitor::~Visitor() {}
9
10 void Visitor::visit(RealLiteral *) {
11     fatalExit("unimplemented feature");
12 }
13
14 void Visitor::visit(CharLiteral *) {
15     fatalExit("unimplemented feature");
16 }
17
18 void Visitor::visit(StringLiteral *) {
19     fatalExit("unimplemented feature");
20 }
21
22 void Visitor::visit(IntIdentifier *) {
23     fatalExit("unimplemented feature");
24 }
25
26 void Visitor::visit(IdentifierEqualsExpression *) {
27     fatalExit("unimplemented feature");
28 }
29
30 void Visitor::visit(RecordExpression *) {
31     fatalExit("unimplemented feature");
32 }
33
34 void Visitor::visit(SelectorExpression *) {
35     fatalExit("unimplemented feature");
36 }
37
38 void Visitor::visit(TypeExpression *) {
39     fatalExit("unimplemented feature");
40 }

```

```

41
42 void Visitor::visit(CaseExpression *) {
43     fatalExit("unimplemented feature");
44 }
45
46 void Visitor::visit(ShortIdentifierPattern *) {
47     fatalExit("unimplemented feature");
48 }
49
50 void Visitor::visit(AsPattern *) {
51     fatalExit("unimplemented feature");
52 }
53
54 void Visitor::visit(IdentifierEqualsPattern *) {
55     fatalExit("unimplemented feature");
56 }
57
58 void Visitor::visit(RecordPattern *) {
59     fatalExit("unimplemented feature");
60 }
61
62 void Visitor::visit(OpenDeclaration *) {
63     fatalExit("unimplemented feature");
64 }
65
66 void Visitor::visit(RestrictedStructDeclaration *) {
67     fatalExit("unimplemented feature");
68 }
69
70 void Visitor::visit(BareStructDeclaration *) {
71     fatalExit("unimplemented feature");
72 }
73
74 void Visitor::visit(AbstractValDeclaration *) {
75     fatalExit("unimplemented feature");
76 }
77
78 void Visitor::visit(AbstractTypeDeclaration *) {
79     fatalExit("unimplemented feature");
80 }
81
82 void Visitor::visit(AbstractStructDeclaration *) {
83     fatalExit("unimplemented feature");
84 }
85
86 void Visitor::visit(SharingTypeDeclaration *) {
87     fatalExit("unimplemented feature");
88 }
89
90 void Visitor::visit(SigDeclaration *) {
91     fatalExit("unimplemented feature");
92 }
93
94 void Visitor::visit(TypeDeclaration *) {
95     fatalExit("unimplemented feature");
96 }
97
98 void Visitor::visit(ShortBareFunctorDeclaration *) {
99     fatalExit("unimplemented feature");
100 }
101
102 void Visitor::visit(ShortRestrictedFunctorDeclaration *) {

```

```

103     fatalError("unimplemented feature");
104 }
105
106 void Visitor::visit(LongBareFunctorDeclaration *) {
107     fatalError("unimplemented feature");
108 }
109
110 void Visitor::visit(LongRestrictedFunctorDeclaration *) {
111     fatalError("unimplemented feature");
112 }
113
114 void Visitor::visit(LongIdentifierDefinition *) {
115     fatalError("unimplemented feature");
116 }
117
118 void Visitor::visit(StructDefinition *) {
119     fatalError("unimplemented feature");
120 }
121
122 void Visitor::visit(AnnotationDefinition *) {
123     fatalError("unimplemented feature");
124 }
125
126 void Visitor::visit(ShortFunctorDefinition *) {
127     fatalError("unimplemented feature");
128 }
129
130 void Visitor::visit(LongFunctorDefinition *) {
131     fatalError("unimplemented feature");
132 }
133
134 void Visitor::visit(SigDefinition *) {
135     fatalError("unimplemented feature");
136 }
137
138 void Visitor::visit(IdentifierColonType *) {
139     fatalError("unimplemented feature");
140 }
141
142 void Visitor::visit(RecordType *) {
143     fatalError("unimplemented feature");
144 }

```

Listing 56: lib/AST/Fixup/Fixup.cpp

```

1  #include "ssml/AST/Fixup/Fixup.h"
2
3  #include "FixupVisitor.h"
4
5  #include "ssml/AST/Declaration.h"
6  #include "ssml/AST/Visitor.h"
7
8  using namespace ssml::ast::fix;
9
10 void ssml::ast::fixup(Root *R) {
11     FixupVisitor V;
12     V.visit(R);
13 }

```

Listing 57: lib/AST/Fixup/OperatorDefinition.cpp

```

1 #include "OperatorDefinition.h"
2
3 ssml::ast::fix::OperatorDefinition::OperatorDefinition(
4     std::shared_ptr<const std::string> ID, Associativity A,
5     Precedence P)
6     : Assoc(A), Prec(P) {}

```

Listing 58: lib/AST/Fixup/OperatorDefVisitor.cpp

```

1 #include "OperatorDefVisitor.h"
2
3 #include "OperatorDefMap.h"
4
5 #include "ssml/AST/Node.h"
6 #include "ssml/AST/Declaration.h"
7 #include "ssml/AST/Identifier.h"
8 #include "ssml/AST/Literal.h"
9 #include "ssml/AST/Match.h"
10 #include "ssml/AST/Pattern.h"
11 #include "ssml/AST/Expression.h"
12 #include "ssml/AST/Type.h"
13 #include "ssml/AST/Definition.h"
14
15 #include "ssml/Common/FatalExit.h"
16 #include "ssml/Common/ErrorMessage.h"
17
18 using namespace ssml::ast;
19 using namespace ssml::ast::fix;
20
21 OperatorDefVisitor::OperatorDefVisitor(OperatorDefMap &M) :
22     OpDefs(M) {}
23
24 OperatorDefinition
25 OperatorDefVisitor::getOperatorDefinition(Node &N, OperatorDefMap
26     &M) {
27     OperatorDefVisitor V(M);
28     N.accept(&V);
29     return V.OpDef;
30 }
31
32 OperatorDefinition
33 OperatorDefVisitor::getShortIdentifierOpDef(ShortIdentifier &ID) {
34     return this->OpDefs.get(ID.getID());
35 }
36
37 OperatorDefinition
38 OperatorDefVisitor::getLongIdentifierOpDef(
39     LongIdentifier &IDs, SourceLocation L, bool opKeyPrefixed) {
40     auto S = IDs.size();
41     assert(S >= 1);
42
43     if (opKeyPrefixed) {
44         #if 0
45         if (S > 1) {
46             errorExit(L, "invalid 'op' keyword before long identifier");
47         } else {
48             return OperatorDefinition();
49         }
50     }
51     #endif
52     return OperatorDefinition();
53 } else if (S > 1) {

```

```

50     return OperatorDefinition();
51 } else {
52     return this->OpDefs.get(IDs.back()->getID());
53 }
54 return OperatorDefinition();
55 }
56
57 void OperatorDefVisitor::visit(IntLiteral *N) {
58     fatalExit("internal error"); }
59 void OperatorDefVisitor::visit(Match *N) { fatalExit("internal
60     error"); }
61 void OperatorDefVisitor::visit(SeqMatch *N) { fatalExit("internal
62     error"); }
63 void OperatorDefVisitor::visit(ShortIdentifier *N) {
64     fatalExit("internal error");
65 }
66
67 void OperatorDefVisitor::visit(LongIdentifier *N) {
68     fatalExit("internal error");
69 }
70
71 void OperatorDefVisitor::visit(IntIdentifier *N) {
72     fatalExit("internal error");
73 }
74
75 void OperatorDefVisitor::visit(SeqShortIdentifier *N) {
76     fatalExit("internal error");
77 }
78
79 void OperatorDefVisitor::visit(SeqLongIdentifier *N) {
80     fatalExit("internal error");
81 }
82
83 void OperatorDefVisitor::visit(SeqExpression *N) {
84     for (auto C : *N)
85         C->accept(this);
86 }
87
88 void OperatorDefVisitor::visit(LiteralExpression *N) {
89     this->OpDef = OperatorDefinition();
90 }
91
92 void OperatorDefVisitor::visit(LongIdentifierExpression *N) {
93     this->OpDef = this->getLongIdentifierOpDef(*N->getIDs(),
94         N->getLocation(),
95         N->isPrefixedWithOpKey());
96 }
97 void OperatorDefVisitor::visit(TupleExpression *N) {
98     this->OpDef = OperatorDefinition();
99 }
100
101 void OperatorDefVisitor::visit(ListExpression *N) {
102     this->OpDef = OperatorDefinition();
103 }
104
105 void OperatorDefVisitor::visit(ApplyExpression *N) {
106     this->OpDef = OperatorDefinition();
107 }

```

```

108
109 void OperatorDefVisitor::visit(OrElseExpression *N) {
110     this->OpDef = OperatorDefinition();
111 }
112
113 void OperatorDefVisitor::visit(AndAlsoExpression *N) {
114     this->OpDef = OperatorDefinition();
115 }
116
117 void OperatorDefVisitor::visit(LetExpression *N) {
118     this->OpDef = OperatorDefinition();
119 }
120
121 void OperatorDefVisitor::visit(IfExpression *N) {
122     this->OpDef = OperatorDefinition();
123 }
124
125 void OperatorDefVisitor::visit(WhileExpression *N) {
126     this->OpDef = OperatorDefinition();
127 }
128
129 void OperatorDefVisitor::visit(CaseExpression *N) {
130     this->OpDef = OperatorDefinition();
131 }
132
133 void OperatorDefVisitor::visit(LambdaExpression *N) {
134     this->OpDef = OperatorDefinition();
135 }
136
137 void OperatorDefVisitor::visit(SeqPattern *N) {
138     assert(N->size() == 1);
139     N->back()->accept(this);
140 }
141
142 void OperatorDefVisitor::visit(LiteralPattern *N) {
143     this->OpDef = OperatorDefinition();
144 }
145
146 void OperatorDefVisitor::visit(WildcardPattern *N) {
147     this->OpDef = OperatorDefinition();
148 }
149
150 void OperatorDefVisitor::visit(ShortIdentifierPattern *N) {
151     this->OpDef = this->getShortIdentifierOpDef(*N->getID());
152 }
153
154 void OperatorDefVisitor::visit(LongIdentifierPattern *N) {
155     this->OpDef = this->getLongIdentifierOpDef(*N->getIDs(),
156                                               N->getLocation(),
157                                               N->isPrefixedWithOpKey());
158 }
159
160 void OperatorDefVisitor::visit(TypePattern *N) {
161     this->OpDef = OperatorDefinition();
162 }
163
164 void OperatorDefVisitor::visit(ApplyPattern *N) {
165     this->OpDef = OperatorDefinition();
166 }
167
168 void OperatorDefVisitor::visit(ListPattern *N) {
169     this->OpDef = OperatorDefinition();

```

```

169 }
170
171 void OperatorDefVisitor::visit(TuplePattern *N) {
172     this->OpDef = OperatorDefinition();
173 }
174
175 void OperatorDefVisitor::visit(SeqDeclaration *N) {
176     fatalExit("internal error");
177 }
178
179 void OperatorDefVisitor::visit(Root *N) { fatalExit("internal
    error"); }
180
181 void OperatorDefVisitor::visit(ValDeclaration *N) {
182     fatalExit("internal error");
183 }
184
185 void OperatorDefVisitor::visit(OpenDeclaration *N) {
186     fatalExit("internal error");
187 }
188
189 void OperatorDefVisitor::visit(NonfixDeclaration *N) {
190     fatalExit("internal error");
191 }
192
193 void OperatorDefVisitor::visit(InfixDeclaration *N) {
194     fatalExit("internal error");
195 }
196
197 void OperatorDefVisitor::visit(InfixRDeclaration *N) {
198     fatalExit("internal error");
199 }
200
201 void OperatorDefVisitor::visit(BareStructDeclaration *N) {
202     fatalExit("internal error");
203 }
204
205 void OperatorDefVisitor::visit(RestrictedStructDeclaration *N) {
206     fatalExit("internal error");
207 }
208
209 void OperatorDefVisitor::visit(AbstractValDeclaration *N) {
210     fatalExit("internal error");
211 }
212
213 void OperatorDefVisitor::visit(AbstractTypeDeclaration *N) {
214     fatalExit("internal error");
215 }
216
217 void OperatorDefVisitor::visit(AbstractStructDeclaration *N) {
218     fatalExit("internal error");
219 }
220
221 void OperatorDefVisitor::visit(SigDeclaration *N) {
222     fatalExit("internal error");
223 }
224
225 void OperatorDefVisitor::visit(FunPatternDeclaration *N) {
226     fatalExit("internal error");
227 }
228
229 void OperatorDefVisitor::visit(FunDeclaration *N) {

```

```

230     fatalError("internal error");
231 }
232
233 void OperatorDefVisitor::visit(TypeDeclaration *N) {
234     fatalError("internal error");
235 }
236
237 void OperatorDefVisitor::visit(DatatypeBareInstanceDeclaration *N)
238 {
239     fatalError("internal error");
240 }
241
242 void OperatorDefVisitor::visit(DatatypeTypedInstanceDeclaration
243     *N) {
244     fatalError("internal error");
245 }
246
247 void OperatorDefVisitor::visit(BareDatatypeDeclaration *N) {
248     fatalError("internal error");
249 }
250
251 void OperatorDefVisitor::visit(TypedDatatypeDeclaration *N) {
252     fatalError("internal error");
253 }
254
255 void OperatorDefVisitor::visit(ShortBareFunctorDeclaration *N) {
256     fatalError("internal error");
257 }
258
259 void OperatorDefVisitor::visit(ShortRestrictedFunctorDeclaration
260     *N) {
261     fatalError("internal error");
262 }
263
264 void OperatorDefVisitor::visit(LongBareFunctorDeclaration *N) {
265     fatalError("internal error");
266 }
267
268 void OperatorDefVisitor::visit(LongRestrictedFunctorDeclaration
269     *N) {
270     fatalError("internal error");
271 }
272
273 void OperatorDefVisitor::visit(StructDefinition *N) {
274     fatalError("internal error");
275 }
276
277 void OperatorDefVisitor::visit(AnnotationDefinition *N) {
278     fatalError("internal error");
279 }
280
281 void OperatorDefVisitor::visit(ShortFunctorDefinition *N) {
282     fatalError("internal error");
283 }
284
285 void OperatorDefVisitor::visit(LongFunctorDefinition *N) {
286     fatalError("internal error");
287 }

```

```

288
289 void OperatorDefVisitor::visit(SigDefinition *N) {
290     fatalExit("internal error");
291 }
292
293 void OperatorDefVisitor::visit(LongIdentifierType *N) {
294     fatalExit("internal error");
295 }
296
297 void OperatorDefVisitor::visit(VariableType *N) {
298     fatalExit("internal error"); }
299
300 void OperatorDefVisitor::visit(SeqType *N) { fatalExit("internal
301     error"); }
302
303 void OperatorDefVisitor::visit(SeqVariableType *N) {
304     fatalExit("internal error");
305 }
306
307 void OperatorDefVisitor::visit(TupleType *N) { fatalExit("internal
308     error"); }
309
310 void OperatorDefVisitor::visit(ApplyType *N) { fatalExit("internal
311     error"); }
312
313 void OperatorDefVisitor::visit(ProductType *N) {
314     fatalExit("internal error"); }
315
316 void OperatorDefVisitor::visit(FunctionType *N) {
317     fatalExit("internal error"); }

```

Listing 59: lib/AST/Fixup/OperatorDefMap.cpp

```

1  #include "OperatorDefMap.h"
2
3  using namespace ssml::ast::fix;
4
5  OperatorDefinition OperatorDefMap::get(std::shared_ptr<const
6      std::string> K) {
7      for (auto It = Maps.rbegin(), End = Maps.rend(); It != End;
8          ++It) {
9          auto &M = *It;
10         if (M.count(K))
11             return M[K];
12     }
13     return Maps.front()[K];
14 }
15
16 void OperatorDefMap::put(std::shared_ptr<const std::string> K,
17     OperatorDefinition Def) {
18     Maps.back()[K] = Def;
19 }
20
21 void OperatorDefMap::enterScope() {
22     this->Maps.push_back(MapType());
23 }
24
25 void OperatorDefMap::leaveScope() {
26     this->Maps.pop_back();
27 }

```

Listing 60: lib/AST/Fixup/FixupVisitor.cpp

```

1  #define DEBUG_TYPE "OperatorFixupParser"
2
3  #include "FixupVisitor.h"
4
5  #include "OperatorFixupParser.h"
6  #include "OperatorDefVisitor.h"
7
8  #include "ssml/AST/Node.h"
9  #include "ssml/AST/Declaration.h"
10 #include "ssml/AST/Identifier.h"
11 #include "ssml/AST/Literal.h"
12 #include "ssml/AST/Match.h"
13 #include "ssml/AST/Pattern.h"
14 #include "ssml/AST/Expression.h"
15 #include "ssml/AST/Type.h"
16 #include "ssml/AST/Definition.h"
17
18 #include "ssml/Common/ErrorMessage.h"
19
20 #include "llvm/Support/raw_ostream.h"
21 #include "llvm/Support/Debug.h"
22
23 using namespace ssml::ast;
24 using namespace ssml::ast::fix;
25
26 template <typename VectorT, typename ItemT, typename TupleT,
27           typename SeqT>
28 std::shared_ptr<VectorT> FixupVisitor::operatorFixup(VectorT
29   &Initial) {
30   OperatorFixupParser<VectorT, ItemT, TupleT, SeqT> Parser(Initial,
31   this->OpDefs);
32   return Parser.parse();
33 }
34
35 void FixupVisitor::visit(IntLiteral *N) {}
36
37 void FixupVisitor::visit(Match *N) {
38   N->getPattern()->accept(this);
39   N->getExpr()->accept(this);
40 }
41
42 void FixupVisitor::visit(SeqMatch *N) {
43   for (auto C : *N)
44     C->accept(this);
45 }
46
47 void FixupVisitor::visit(ShortIdentifier *N) {}
48
49 void FixupVisitor::visit(LongIdentifier *N) {}
50
51 void FixupVisitor::visit(IntIdentifier *N) {}
52
53 void FixupVisitor::visit(SeqShortIdentifier *N) {}
54
55 void FixupVisitor::visit(SeqLongIdentifier *N) {}
56
57 void FixupVisitor::visit(SeqExpression *N) {
58   for (auto C : *N)
59     C->accept(this);
60 }

```

```

60 void FixupVisitor::visit(LiteralExpression *N) {
61     this->CurrOpDef = OperatorDefinition();
62 }
63
64 void FixupVisitor::visit(LongIdentifierExpression *N) {
65     this->CurrOpDef = OperatorDefVisitor::getOperatorDefinition(*N,
66         this->OpDefs);
67 }
68
69 void FixupVisitor::visit(TupleExpression *N) {
70     N->getExprs()->accept(this);
71     this->CurrOpDef = OperatorDefinition();
72 }
73
74 void FixupVisitor::visit(ListExpression *N) {
75     N->getExprs()->accept(this);
76     this->CurrOpDef = OperatorDefinition();
77 }
78
79 void FixupVisitor::visit(ApplyExpression *N) {
80     DEBUG(llvm::errs() << "Fixup ApplyExpression\n");
81     for (auto C : *N)
82         C->accept(this);
83     auto V = this->operatorFixup<ApplyExpression, Expression,
84         TupleExpression,
85         SeqExpression>(*N);
86     N->clear();
87     N->insert(N->begin(), V->begin(), V->end());
88     this->CurrOpDef = OperatorDefinition();
89 }
90
91 void FixupVisitor::visit(OrElseExpression *N) {
92     N->getLeftExpr()->accept(this);
93     N->getRightExpr()->accept(this);
94     this->CurrOpDef = OperatorDefinition();
95 }
96
97 void FixupVisitor::visit(AndAlsoExpression *N) {
98     N->getLeftExpr()->accept(this);
99     N->getRightExpr()->accept(this);
100     this->CurrOpDef = OperatorDefinition();
101 }
102
103 void FixupVisitor::visit(LetExpression *N) {
104     this->OpDefs.enterScope();
105     N->getDecls()->accept(this);
106     N->getExprs()->accept(this);
107     this->CurrOpDef = OperatorDefinition();
108     this->OpDefs.leaveScope();
109 }
110
111 void FixupVisitor::visit(IfExpression *N) {
112     N->getCondExpr()->accept(this);
113     N->getThenExpr()->accept(this);
114     N->getElseExpr()->accept(this);
115     this->CurrOpDef = OperatorDefinition();
116 }
117
118 void FixupVisitor::visit(WhileExpression *N) {
119     N->getCondExpr()->accept(this);
120     N->getBodyExpr()->accept(this);
121     this->CurrOpDef = OperatorDefinition();

```

```

120 }
121
122 void FixupVisitor::visit(CaseExpression *N) {
123     N->getExpr()->accept(this);
124     N->getCases()->accept(this);
125     this->CurrOpDef = OperatorDefinition();
126 }
127
128 void FixupVisitor::visit(LambdaExpression *N) {
129     N->getCases()->accept(this);
130     this->CurrOpDef = OperatorDefinition();
131 }
132
133 void FixupVisitor::visit(SeqPattern *N) {
134     for (auto C : *N)
135         C->accept(this);
136 }
137
138 void FixupVisitor::visit(LiteralPattern *N) {
139     this->CurrOpDef = OperatorDefinition();
140 }
141
142 void FixupVisitor::visit(WildcardPattern *N) {
143     this->CurrOpDef = OperatorDefinition();
144 }
145
146 void FixupVisitor::visit(ShortIdentifierPattern *N) {
147     this->CurrOpDef = OperatorDefVisitor::getOperatorDefinition(*N,
148         this->OpDefs);
149 }
150
151 void FixupVisitor::visit(LongIdentifierPattern *N) {
152     this->CurrOpDef = OperatorDefVisitor::getOperatorDefinition(*N,
153         this->OpDefs);
154 }
155
156 void FixupVisitor::visit(TypePattern *N) {
157     N->getPattern()->accept(this);
158     N->getType()->accept(this);
159     this->CurrOpDef = OperatorDefinition();
160 }
161
162 void FixupVisitor::visit(ApplyPattern *N) {
163     DEBUG(llvm::errs() << "Fixup ApplyPattern\n");
164     for (auto C : *N)
165         C->accept(this);
166     auto V =
167         this->operatorFixup<ApplyPattern, Pattern, TuplePattern,
168             SeqPattern>(*N);
169     N->clear();
170     N->insert(N->begin(), V->begin(), V->end());
171     this->CurrOpDef = OperatorDefinition();
172 }
173
174 void FixupVisitor::visit(ListPattern *N) {
175     N->getPatterns()->accept(this);
176     this->CurrOpDef = OperatorDefinition();
177 }
178
179 void FixupVisitor::visit(TuplePattern *N) {
180     N->getPatterns()->accept(this);
181     this->CurrOpDef = OperatorDefinition();
182 }

```

```

179 }
180
181 void FixupVisitor::visit(SeqDeclaration *N) {
182     for (auto C : *N)
183         C->accept(this);
184 }
185
186 void FixupVisitor::visit(Root *N) {
187     this->OpDefs.enterScope();
188
189     std::shared_ptr<ShortIdentifier> Cons =
190         std::shared_ptr<ShortIdentifier>(new ShortIdentifier(
191             SourceLocation(0, 0),
192             std::make_shared<std::string>(":=")));
193
194     std::shared_ptr<ShortIdentifier> Plus =
195         std::shared_ptr<ShortIdentifier>(new ShortIdentifier(
196             SourceLocation(0, 0),
197             std::make_shared<std::string>("+")));
198
199     std::shared_ptr<ShortIdentifier> Minus =
200         std::shared_ptr<ShortIdentifier>(new ShortIdentifier(
201             SourceLocation(0, 0),
202             std::make_shared<std::string>("-")));
203
204     std::shared_ptr<ShortIdentifier> Multiply =
205         std::shared_ptr<ShortIdentifier>(new ShortIdentifier(
206             SourceLocation(0, 0),
207             std::make_shared<std::string>("*")));
208
209     std::shared_ptr<ShortIdentifier> Division =
210         std::shared_ptr<ShortIdentifier>(new ShortIdentifier(
211             SourceLocation(0, 0),
212             std::make_shared<std::string>("div")));
213
214     std::shared_ptr<ShortIdentifier> Modulo =
215         std::shared_ptr<ShortIdentifier>(new ShortIdentifier(
216             SourceLocation(0, 0),
217             std::make_shared<std::string>("mod")));
218
219     std::shared_ptr<ShortIdentifier> Refassign =
220         std::shared_ptr<ShortIdentifier>(new ShortIdentifier(
221             SourceLocation(0, 0),
222             std::make_shared<std::string>(":=")));
223
224     std::shared_ptr<ShortIdentifier> Equals =
225         std::shared_ptr<ShortIdentifier>(new ShortIdentifier(
226             SourceLocation(0, 0),
227             std::make_shared<std::string>("=")));
228
229     std::shared_ptr<ShortIdentifier> NotEquals =
230         std::shared_ptr<ShortIdentifier>(new ShortIdentifier(
231             SourceLocation(0, 0),
232             std::make_shared<std::string>("<>")));
233
234     std::shared_ptr<ShortIdentifier> Less =
235         std::shared_ptr<ShortIdentifier>(new ShortIdentifier(
236             SourceLocation(0, 0),
237             std::make_shared<std::string>("<")));
238
239     std::shared_ptr<ShortIdentifier> Greater =
240         std::shared_ptr<ShortIdentifier>(new ShortIdentifier(

```

```

231         SourceLocation(0, 0),
                std::make_shared<std::string>(">"));
232
233     std::shared_ptr<ShortIdentifier> LessEquals =
234         std::shared_ptr<ShortIdentifier>(new ShortIdentifier(
235             SourceLocation(0, 0),
                std::make_shared<std::string>("<=")));
236
237     std::shared_ptr<ShortIdentifier> GreaterEquals =
238         std::shared_ptr<ShortIdentifier>(new ShortIdentifier(
239             SourceLocation(0, 0),
                std::make_shared<std::string>(">=")));
240
241     SeqShortIdentifier Infixr3Seq(SourceLocation(0, 0), Refassign);
242     IntLiteral Infixr3Lit(SourceLocation(0, 0),
                std::make_shared<std::string>("3"));
243
244     SeqShortIdentifier Infix4Seq(SourceLocation(0, 0), Equals);
245     Infix4Seq.push_back(NotEquals);
246     Infix4Seq.push_back(Less);
247     Infix4Seq.push_back(Greater);
248     Infix4Seq.push_back(LessEquals);
249     Infix4Seq.push_back(GreaterEquals);
250     IntLiteral Infix4Lit(SourceLocation(0, 0),
                std::make_shared<std::string>("4"));
251
252     SeqShortIdentifier Infixr5Seq(SourceLocation(0, 0), Cons);
253     IntLiteral Infixr5Lit(SourceLocation(0, 0),
                std::make_shared<std::string>("5"));
254
255     SeqShortIdentifier Infix6Seq(SourceLocation(0, 0), Plus);
256     Infix6Seq.push_back(Minus);
257     IntLiteral Infix6Lit(SourceLocation(0, 0),
                std::make_shared<std::string>("6"));
258
259     SeqShortIdentifier Infix7Seq(SourceLocation(0, 0), Multiply);
260     Infix7Seq.push_back(Division);
261     Infix7Seq.push_back(Modulo);
262     IntLiteral Infix7Lit(SourceLocation(0, 0),
                std::make_shared<std::string>("7"));
263
264     setOpDef(Infixr3Seq, OperatorDefinition::INFIXR, Infixr3Lit);
265     setOpDef(Infix4Seq, OperatorDefinition::INFIX, Infix4Lit);
266     setOpDef(Infixr5Seq, OperatorDefinition::INFIXR, Infixr5Lit);
267     setOpDef(Infix6Seq, OperatorDefinition::INFIX, Infix6Lit);
268     setOpDef(Infix7Seq, OperatorDefinition::INFIX, Infix7Lit);
269
270     this->OpDefs.enterScope();
271     for (auto C : *N)
272         C->accept(this);
273     this->OpDefs.leaveScope();
274     this->OpDefs.leaveScope();
275 }
276
277 void FixupVisitor::visit(ValDeclaration *N) {
278     N->getDest()->accept(this);
279     N->getSource()->accept(this);
280 }
281
282 void FixupVisitor::setNonfixOpDef(SeqShortIdentifier &IDs) {
283     for (auto ID : IDs)
284         this->OpDefs.put(ID->getID(), OperatorDefinition());
285 }

```

```

290 }
291
292 void FixupVisitor::setOpDef(SeqShortIdentifier &IDs,
293                           OperatorDefinition::Associativity A,
294                           IntLiteral &Prec) {
295     auto P = Prec.getIntValue();
296     if (P < OperatorDefinition::minPrecedence() ||
297         P > OperatorDefinition::maxPrecedence()) {
298         errorExit(Prec.getLocation(),
299                 "invalid operator precedence: " +
300                 *Prec.getValueString());
301     }
302     for (auto ID : IDs)
303         this->OpDefs.put(ID->getID(), OperatorDefinition(ID->getID(),
304                 A, P));
305 }
306
307 void FixupVisitor::visit(NonfixDeclaration *N) {
308     DEBUG(llvm::errs() << "Nonfix declaration\n");
309     this->setNonfixOpDef(*N->getIDs());
310 }
311
312 void FixupVisitor::visit(InfixDeclaration *N) {
313     DEBUG(llvm::errs() << "Infix declaration\n");
314     setOpDef(*N->getIDs(), OperatorDefinition::INFIX,
315             *N->getPrecedence());
316 }
317
318 void FixupVisitor::visit(InfixRDeclaration *N) {
319     DEBUG(llvm::errs() << "Infixr declaration\n");
320     setOpDef(*N->getIDs(), OperatorDefinition::INFIXR,
321             *N->getPrecedence());
322 }
323
324 void FixupVisitor::visit(BareStructDeclaration *N) {
325     N->getID()->accept(this);
326     N->getStructDef()->accept(this);
327 }
328
329 void FixupVisitor::visit(RestrictedStructDeclaration *N) {
330     N->getID()->accept(this);
331     N->getSigDef()->accept(this);
332     N->getStructDef()->accept(this);
333 }
334
335 void FixupVisitor::visit(AbstractValDeclaration *N) {
336     N->getID()->accept(this);
337     N->getType()->accept(this);
338 }
339
340 void FixupVisitor::visit(AbstractTypeDeclaration *N) {
341     N->getID()->accept(this);
342 }
343
344 void FixupVisitor::visit(AbstractStructDeclaration *N) {
345     N->getID()->accept(this);
346     N->getSigDef()->accept(this);
347 }
348
349 void FixupVisitor::visit(SigDeclaration *N) {
350     N->getID()->accept(this);

```

```

348     N->getSigDef()->accept(this);
349 }
350
351 void FixupVisitor::visit(FunPatternDeclaration *N) {
352     N->getIDParams()->accept(this);
353     N->getDef()->accept(this);
354 }
355
356 void FixupVisitor::visit(FunDeclaration *N) {
357     for (auto C : *N)
358         C->accept(this);
359 }
360
361 void FixupVisitor::visit(TypeDeclaration *N) {
362     N->getID()->accept(this);
363     N->getType()->accept(this);
364 }
365
366 void FixupVisitor::visit(DatatypeBareInstanceDeclaration *N) {
367     N->getID()->accept(this);
368 }
369
370 void FixupVisitor::visit(DatatypeTypedInstanceDeclaration *N) {
371     N->getID()->accept(this);
372     N->getType()->accept(this);
373 }
374
375 void FixupVisitor::visit(BareDatatypeDeclaration *N) {
376     N->getID()->accept(this);
377     N->getDef()->accept(this);
378 }
379
380 void FixupVisitor::visit(TypedDatatypeDeclaration *N) {
381     N->getTypeParams()->accept(this);
382     N->getID()->accept(this);
383     N->getDef()->accept(this);
384 }
385
386 void FixupVisitor::visit(ShortBareFunctorDeclaration *N) {
387     N->getFunctorID()->accept(this);
388     N->getParamID()->accept(this);
389     N->getParamSigDef()->accept(this);
390     N->getFunctorStructDef()->accept(this);
391 }
392
393 void FixupVisitor::visit(ShortRestrictedFunctorDeclaration *N) {
394     N->getFunctorID()->accept(this);
395     N->getParamID()->accept(this);
396     N->getParamSigDef()->accept(this);
397     N->getFunctorSigDef()->accept(this);
398     N->getFunctorStructDef()->accept(this);
399 }
400
401 void FixupVisitor::visit(LongBareFunctorDeclaration *N) {
402     N->getFunctorID()->accept(this);
403     N->getParams()->accept(this);
404     N->getFunctorStructDef()->accept(this);
405 }
406
407 void FixupVisitor::visit(LongRestrictedFunctorDeclaration *N) {
408     N->getFunctorID()->accept(this);
409     N->getParams()->accept(this);

```

```

410     N->getFunctorSigDef()->accept(this);
411     N->getFunctorStructDef()->accept(this);
412 }
413
414 void FixupVisitor::visit(LongIdentifierDefinition *N) {}
415
416 void FixupVisitor::visit(StructDefinition *N) {
417     this->OpDefs.enterScope();
418     N->getDecls()->accept(this);
419     this->OpDefs.leaveScope();
420 }
421
422 void FixupVisitor::visit(AnnotationDefinition *N) {
423     N->getStructDef()->accept(this);
424     N->getSigDef()->accept(this);
425 }
426
427 void FixupVisitor::visit(ShortFunctorDefinition *N) {
428     N->getFunctorID()->accept(this);
429     N->getStructDef()->accept(this);
430 }
431
432 void FixupVisitor::visit(LongFunctorDefinition *N) {
433     N->getFunctorID()->accept(this);
434     N->getDecls()->accept(this);
435 }
436
437 void FixupVisitor::visit(SigDefinition *N) {}
438
439 void FixupVisitor::visit(LongIdentifierType *N) {}
440
441 void FixupVisitor::visit(VariableType *N) {}
442
443 void FixupVisitor::visit(SeqType *N) {}
444
445 void FixupVisitor::visit(SeqVariableType *N) {}
446
447 void FixupVisitor::visit(TupleType *N) {}
448
449 void FixupVisitor::visit(ApplyType *N) {}
450
451 void FixupVisitor::visit(ProductType *N) {}
452
453 void FixupVisitor::visit(FunctionType *N) {
454     if (N->size() > 2) {
455         std::vector<std::shared_ptr<Type>> Vec = *N;
456         N->clear();
457         FunctionType *Fun = N;
458         Fun->push_back(Vec[0]);
459         Vec.erase(Vec.begin());
460         while (Vec.size() >= 2) {
461             auto Next = std::shared_ptr<FunctionType>(
462                 new FunctionType(Vec[0]->getLocation(), Vec[0]));
463             Fun->push_back(Next);
464             Fun = Next.get();
465             Vec.erase(Vec.begin());
466         }
467         assert(Vec.size() == 1);
468         Fun->push_back(Vec[0]);
469     }
470 }

```

Listing 61: lib/AST/Fixup/OperatorFixupParser.cpp

```

1  #define DEBUG_TYPE "OperatorFixupParser"
2
3  #include "OperatorFixupParser.h"
4  #include "OperatorDefVisitor.h"
5  #include "OperatorDefMap.h"
6
7  #include "ssml/AST/Node.h"
8  #include "ssml/AST/Expression.h"
9  #include "ssml/AST/Pattern.h"
10
11 #include "ssml/Common/ErrorMessage.h"
12 #include "ssml/Common/SourceLocation.h"
13
14 #include "llvm/Support/raw_ostream.h"
15 #include "llvm/Support/ErrorHandler.h"
16 #include "llvm/Support/Debug.h"
17
18 #include <cassert>
19 #include <set>
20
21 #define FIXUP_DLOG(msg) DEBUG(llvm::errs() << DEBUG_TYPE " :: " <<
    msg << '\n')
22
23 using namespace llvm;
24 using namespace ssml;
25 using namespace ssml::ast;
26 using namespace ssml::ast::fix;
27
28 template <typename VectorT, typename ItemT, typename TupleT,
    typename SeqT>
29 OperatorFixupParser<VectorT, ItemT, TupleT,
    SeqT>::OperatorFixupParser(
30     VectorT &Operators, OperatorDefMap &OpDefs)
31     : Operators(Operators), Current(Operators.begin()),
    End(Operators.end()),
32     OpDefs(OpDefs) {}
33
34 template <typename VectorT, typename ItemT, typename TupleT,
    typename SeqT>
35 void OperatorFixupParser<VectorT, ItemT, TupleT,
    SeqT>::errorExitIfInfix(
36     SharedPtrT P) {
37     switch (getDef(P).getAssoc()) {
38     case OperatorDefinition::INFIX:
39         errorExit(P->getLocation(), "unexpected infix operator");
40     case OperatorDefinition::INFIXR:
41         errorExit(P->getLocation(), "unexpected infixr operator");
42     default:
43         break;
44     }
45 }
46
47 template <typename VectorT, typename ItemT, typename TupleT,
    typename SeqT>
48 auto OperatorFixupParser<VectorT, ItemT, TupleT,
    SeqT>::infixToApply(
49     Precedence OpPrec, SharedPtrVec LHS, SharedPtrT BinOp) ->
    SharedPtrVec {
50     FIXUP_DLOG("convert binary expression into application
    expression");
51

```

```

52     SharedPtrVec Apply(new VectorT(BinOp->getLocation(), BinOp));
53
54     auto RHS = this->next();
55     if (!RHS)
56         errorExit(BinOp->getLocation(), "expected operand after
57             operator");
58     errorExitIfInfix(RHS);
59
60     SharedPtrSeq Args(new SeqT(LHS->getLocation(), LHS));
61     SharedPtrVec Next(new VectorT(RHS->getLocation(), RHS));
62     if (OpPrec > OperatorDefinition::maxPrecedence())
63         Args->push_back(this->application(Next));
64     else
65         Args->push_back(this->binary(OpPrec, Next));
66
67     SharedPtrTup Tup(new TupleT(Args->getLocation(), Args));
68     Apply->push_back(Tup);
69
70     return Apply;
71 }
72
73 template <typename VectorT, typename ItemT, typename TupleT,
74     typename SeqT>
75 auto OperatorFixupParser<VectorT, ItemT, TupleT, SeqT>::binary(
76     Precedence OpPrec, SharedPtrVec LHS) -> SharedPtrVec {
77     FIXUP_DLOG("binary expression begin");
78
79     while (auto BinOp = this->peek()) {
80         FIXUP_DLOG("binary expression loop start");
81
82         auto BinOpDef = getDef(BinOp);
83
84         if (BinOpDef.getAssoc() == OperatorDefinition::NONFIX) {
85             FIXUP_DLOG("operator is NONFIX");
86             LHS = application(LHS);
87             continue;
88         }
89
90         if (BinOpDef.getPrec() < OpPrec)
91             break;
92
93         if (BinOpDef.getPrec() > OpPrec) {
94             FIXUP_DLOG("goto next operator precedence");
95             LHS = binary(BinOpDef.getPrec(), LHS);
96             continue;
97         }
98
99         this->next();
100         FIXUP_DLOG("found correct operator precedence");
101         if (BinOpDef.getAssoc() == OperatorDefinition::INFIX)
102             LHS = infixToApply(OpPrec + 1, LHS, BinOp);
103         else /* INFIXR */
104             LHS = infixToApply(OpPrec, LHS, BinOp);
105     }
106
107     FIXUP_DLOG("binary expression return");
108     return LHS;
109 }
110
111 template <typename VectorT, typename ItemT, typename TupleT,
112     typename SeqT>
113 auto OperatorFixupParser<VectorT, ItemT, TupleT,

```

```

111     SeqT>::application(
112         SharedPtrVec Op) -> SharedPtrVec {
113     FIXUP_DLOG("application expression begin");
114     SharedPtrT Next;
115     while ((Next = peek()) &&
116           getDef(Next).getAssoc() == OperatorDefinition::NONFIX) {
117         FIXUP_DLOG("application expression apply argument");
118         Op->push_back(this->next());
119     }
120     FIXUP_DLOG("application expression return");
121     return Op;
122 }
123 namespace {
124 struct PrecedenceLess {
125     bool operator()(OperatorDefinition L, OperatorDefinition R)
126     const {
127         return L.getPrec() < R.getPrec();
128     }
129 };
130 } // End anonymous namespace.
131 template <typename VectorT, typename ItemT, typename TupleT,
132           typename SeqT>
133 void OperatorFixupParser<VectorT, ItemT, TupleT,
134                           SeqT>::errorExitIfIllegalOperatorMix() {
135     std::set<OperatorDefinition, PrecedenceLess> OpSet;
136     for (auto X : this->Operators) {
137         auto Def = getDef(X);
138         if (Def.getAssoc() == OperatorDefinition::NONFIX)
139             continue;
140         if (OpSet.count(Def)) {
141             if (OpSet.find(Def)->getAssoc() != Def.getAssoc())
142                 errorExit(Operators.back()->getLocation(), "mix of infix
143                                     and infixr "
144                                     "operators with
145                                     same "
146                                     "precedence");
147         }
148         OpSet.insert(Def);
149     }
150 }
151 template <typename VectorT, typename ItemT, typename TupleT,
152           typename SeqT>
153 auto OperatorFixupParser<VectorT, ItemT, TupleT, SeqT>::parse()
154     -> SharedPtrVec {
155     FIXUP_DLOG("parse");
156     auto First = next();
157     this->errorExitIfInfix(First);
158     SharedPtrVec LHS(new VectorT(First->getLocation(), First));
159     auto Ret = this->binary(OperatorDefinition::minPrecedence(),
160                           LHS);
161     this->errorExitIfIllegalOperatorMix();
162     return Ret;
163 }
164
165 template <typename VectorT, typename ItemT, typename TupleT,

```

```

166     typename SeqT>
167     OperatorFixupParser<VectorT, ItemT, TupleT,
168         SeqT>::getDef(SharedPtrT P) {
169         return OperatorDefVisitor::getOperatorDefinition(*P,
170             this->OpDefs);
171     }
172
173     template <typename VectorT, typename ItemT, typename TupleT,
174         typename SeqT>
175     auto OperatorFixupParser<VectorT, ItemT, TupleT, SeqT>::peek() ->
176         SharedPtrT {
177         if (this->Current == this->End)
178             return nullptr;
179         return *this->Current;
180     }
181
182     template <typename VectorT, typename ItemT, typename TupleT,
183         typename SeqT>
184     auto OperatorFixupParser<VectorT, ItemT, TupleT, SeqT>::next() ->
185         SharedPtrT {
186         if (this->Current == this->End)
187             return nullptr;
188         SharedPtrT Ret = *this->Current;
189         ++Current;
190         return Ret;
191     }
192
193     template class ssml::ast::fix::OperatorFixupParser<
194         ApplyExpression, Expression, TupleExpression, SeqExpression>;
195     template class ssml::ast::fix::OperatorFixupParser<ApplyPattern,
196         Pattern,
197                                     TuplePattern,
198                                     SeqPattern>;

```

Listing 62: lib/AST/Fixup/OperatorDefVisitor.h

```

1  #ifndef LLVM_TOOLS_SSML_LIB_AST_FIXUP_OPERATORDEFVISITOR_H
2  #define LLVM_TOOLS_SSML_LIB_AST_FIXUP_OPERATORDEFVISITOR_H
3
4  #include "OperatorDefinition.h"
5
6  #include "ssml/AST/Visitor.h"
7
8  #include "ssml/Common/SourceLocation.h"
9
10 namespace ssml {
11     namespace ast {
12         class Node;
13     } // End namespace ast.
14 } // End namespace ssml.
15
16 namespace ssml {
17     namespace ast {
18         namespace fix {
19             class OperatorDefMap;
20         } // End namespace fix.
21     } // End namespace ast.
22 } // End namespace ssml.
23
24 namespace ssml {

```

```

25 namespace ast {
26 namespace fix {
27 class OperatorDefVisitor : public Visitor {
28 private:
29     OperatorDefinition OpDef;
30     OperatorDefMap &OpDefs;
31
32 private:
33     OperatorDefVisitor(OperatorDefMap &M);
34     OperatorDefinition getShortIdentifierOpDef(ShortIdentifier &ID);
35     OperatorDefinition getLongIdentifierOpDef(LongIdentifier &IDs,
36                                             SourceLocation L,
37                                             bool opKeyPrefixed);
38
39 public:
40     static OperatorDefinition getOperatorDefinition(ssml::ast::Node
41                                             &N,
42                                             OperatorDefMap
43                                             &M);
44
45     void visit(IntLiteral *) override;
46
47     void visit(ShortIdentifier *) override;
48     void visit(LongIdentifier *) override;
49     void visit(IntIdentifier *) override;
50     void visit(SeqShortIdentifier *) override;
51     void visit(SeqLongIdentifier *) override;
52
53     void visit(Match *) override;
54     void visit(SeqMatch *) override;
55
56     void visit(SeqExpression *) override;
57     void visit(LiteralExpression *) override;
58     void visit(LongIdentifierExpression *) override;
59     void visit(TupleExpression *) override;
60     void visit(ListExpression *) override;
61     void visit(ApplyExpression *) override;
62     void visit(OrElseExpression *) override;
63     void visit(AndAlsoExpression *) override;
64     void visit(LetExpression *) override;
65     void visit(IfExpression *) override;
66     void visit(WhileExpression *) override;
67     void visit(CaseExpression *) override;
68     void visit(LambdaExpression *) override;
69
70     void visit(SeqPattern *) override;
71     void visit(LiteralPattern *) override;
72     void visit(WildcardPattern *) override;
73     void visit(ShortIdentifierPattern *) override;
74     void visit(LongIdentifierPattern *) override;
75     void visit(TypePattern *) override;
76     void visit(ApplyPattern *) override;
77     void visit(ListPattern *) override;
78     void visit(TuplePattern *) override;
79
80     void visit(SeqDeclaration *) override;
81     void visit(Root *) override;
82     void visit(ValDeclaration *) override;
83     void visit(OpenDeclaration *) override;
84     void visit(NonfixDeclaration *) override;
85     void visit(InfixDeclaration *) override;
86     void visit(InfixRDeclaration *) override;

```

```

85     void visit(RestrictedStructDeclaration *) override;
86     void visit(BareStructDeclaration *) override;
87     void visit(AbstractValDeclaration *) override;
88     void visit(AbstractTypeDeclaration *) override;
89     void visit(AbstractStructDeclaration *) override;
90     void visit(SigDeclaration *) override;
91     void visit(FunPatternDeclaration *) override;
92     void visit(FunDeclaration *) override;
93     void visit(TypeDeclaration *) override;
94     void visit(DatatypeBareInstanceDeclaration *) override;
95     void visit(DatatypeTypedInstanceDeclaration *) override;
96     void visit(BareDatatypeDeclaration *) override;
97     void visit(TypedDatatypeDeclaration *) override;
98     void visit(ShortBareFunctorDeclaration *) override;
99     void visit(ShortRestrictedFunctorDeclaration *) override;
100    void visit(LongBareFunctorDeclaration *) override;
101    void visit(LongRestrictedFunctorDeclaration *) override;
102
103    void visit(LongIdentifierDefinition *) override;
104    void visit(StructDefinition *) override;
105    void visit(AnnotationDefinition *) override;
106    void visit(ShortFunctorDefinition *) override;
107    void visit(LongFunctorDefinition *) override;
108    void visit(SigDefinition *) override;
109
110    void visit(LongIdentifierType *) override;
111    void visit(VariableType *) override;
112    void visit(SeqType *) override;
113    void visit(SeqVariableType *) override;
114    void visit(TupleType *) override;
115    void visit(ApplyType *) override;
116    void visit(ProductType *) override;
117    void visit(FunctionType *) override;
118 };
119 } // End namespace fix.
120 } // End namespace ast.
121 } // End namespace ssml.
122
123 #endif // LLVM_TOOLS_SSML_LIB_AST_FIXUP_OPERATORDEFVISITOR_H

```

Listing 63: lib/AST/Fixup/OperatorDefMap.h

```

1  #ifndef LLVM_TOOLS_SSML_LIB_AST_FIXUP_OPERATORDEFMAP_H
2  #define LLVM_TOOLS_SSML_LIB_AST_FIXUP_OPERATORDEFMAP_H
3
4  #include "OperatorDefinition.h"
5
6  #include "ssml/Common/Compare.h"
7
8  #include <map>
9  #include <vector>
10 #include <string>
11
12 namespace ssml {
13 namespace ast {
14 namespace fix {
15 class OperatorDefMap {
16 private:
17     using MapType =
18         std::map<std::shared_ptr<const std::string>,
19                 OperatorDefinition,

```

```

19         ssml::SharedPtrLess<std::string>>;
20
21 private:
22     std::vector<MapType> Maps;
23
24 public:
25     OperatorDefinition get(std::shared_ptr<const std::string> K);
26     void put(std::shared_ptr<const std::string> K,
27             OperatorDefinition Def);
28     void enterScope();
29     void leaveScope();
30 };
31 // End namespace fix.
32 // End namespace ast.
33 // End namespace ssml.
34 #endif // LLVM_TOOLS_SSML_LIB_AST_FIXUP_OPERATORDEFMAP_H

```

Listing 64: lib/AST/Fixup/OperatorFixupParser.h

```

1 #ifndef LLVM_TOOLS_SSML_LIB_AST_FIXUP_OPERATORFIXUPPARSER_H
2 #define LLVM_TOOLS_SSML_LIB_AST_FIXUP_OPERATORFIXUPPARSER_H
3
4 #include "OperatorDefinition.h"
5
6 #include <vector>
7 #include <memory>
8 #include <string>
9
10 // parse: binary0
11 //
12 // binary0: binary0 'infix0' binary1
13 //         | binary1 'infixr0' binary0
14 //         | binary1
15 //
16 // binary1: binary1 'infix1' binary2
17 //         | binary2 'infixr1' binary1
18 //         | binary2
19 //
20 // ...
21 //
22 // binary9: binary9 'infix9' nonfix
23 //         | application 'infixr9' binary9
24 //         | application
25 //
26 // application: application 'nonfix'
27 //             | 'nonfix'
28
29 namespace ssml {
30 namespace ast {
31 namespace fix {
32 class OperatorDefMap;
33 } // End namespace fix.
34 } // End namespace ast.
35 } // End namespace ssml.
36
37 namespace ssml {
38 namespace ast {
39 class Expression;
40 class ApplyExpression;
41 class SeqExpression;

```

```

42 class TupleExpression;
43 class Pattern;
44 class ApplyPattern;
45 class SeqPattern;
46 class TuplePattern;
47 } // End namespace ast.
48 } // End namespace ssml.
49
50 namespace ssml {
51 namespace ast {
52 namespace fix {
53 template <typename VectorT, typename ItemT, typename TupleT,
54           typename SeqT>
55 class OperatorFixupParser {
56 private:
57     // using VectorT = ApplyExpression;
58     // using ItemT = Expression;
59     using SharedPtrT = std::shared_ptr<ItemT>;
60     using SharedPtrVec = std::shared_ptr<VectorT>;
61     // using TupleT = TupleExpression;
62     using SharedPtrTup = std::shared_ptr<TupleT>;
63     // using SeqT = SeqExpression;
64     using SharedPtrSeq = std::shared_ptr<SeqT>;
65     using Precedence = OperatorDefinition::Precedence;
66 private:
67     VectorT &Operators;
68     typename std::vector<SharedPtrT>::iterator Current;
69     typename std::vector<SharedPtrT>::iterator End;
70     OperatorDefMap &OpDefs;
71 private:
72     SharedPtrT peek();
73     SharedPtrT next();
74
75     SharedPtrVec infixToApply(Precedence OpPrec, SharedPtrVec LHS,
76                               SharedPtrT BinOp);
77     SharedPtrVec binary(Precedence CurrPrec, SharedPtrVec LHS);
78     SharedPtrVec application(SharedPtrVec Op);
79
80     OperatorDefinition getDef(SharedPtrT);
81
82     void errorExitIfInfix(SharedPtrT P);
83
84     void errorExitIfIllegalOperatorMix();
85
86 public:
87     OperatorFixupParser(VectorT &Operators, OperatorDefMap &OpDefs);
88     SharedPtrVec parse();
89 };
90 } // End namespace fix.
91 } // End namespace ast.
92 } // End namespace ssml.
93
94 extern template class ssml::ast::fix::OperatorFixupParser<
95     ssml::ast::ApplyExpression, ssml::ast::Expression,
96     ssml::ast::TupleExpression, ssml::ast::SeqExpression>;
97 extern template class ssml::ast::fix::OperatorFixupParser<
98     ssml::ast::ApplyPattern, ssml::ast::Pattern,
99     ssml::ast::TuplePattern,
100     ssml::ast::SeqPattern>;
101

```

```
102 #endif // LLVM_TOOLS_SSML_LIB_AST_FIXUP_OPERATORFIXUPPARSER_H
```

Listing 65: lib/AST/Fixup/OperatorDefinition.h

```

1 #ifndef LLVM_TOOLS_SSML_LIB_AST_FIXUP_OPERATORDEFINITION_H
2 #define LLVM_TOOLS_SSML_LIB_AST_FIXUP_OPERATORDEFINITION_H
3
4 #include <memory>
5 #include <string>
6
7 namespace ssml {
8 namespace ast {
9 namespace fix {
10 class OperatorDefinition {
11 public:
12     enum Associativity { NONFIX, INFIX, INFIXR };
13     using Precedence = unsigned;
14     constexpr static Precedence maxPrecedence() { return 9; }
15     constexpr static Precedence minPrecedence() { return 0; }
16
17 private:
18     std::shared_ptr<std::string> ID;
19     Associativity Assoc;
20     Precedence Prec;
21
22 public:
23     explicit OperatorDefinition(
24         std::shared_ptr<const std::string> ID =
25         std::shared_ptr<const std::string>(nullptr),
26         Associativity A = NONFIX,
27         Precedence P = OperatorDefinition::minPrecedence());
28     std::shared_ptr<std::string> getID() const { return this->ID; }
29     Associativity getAssoc() const { return this->Assoc; }
30     Precedence getPrec() const { return this->Prec; }
31 };
32 } // End namespace fix.
33 } // End namespace ast.
34 } // End namespace ssml.
35
36 #endif // LLVM_TOOLS_SSML_LIB_AST_FIXUP_OPERATORDEFINITION_H

```

Listing 66: lib/AST/Fixup/FixupVisitor.h

```

1 #ifndef LLVM_TOOLS_SSML_LIB_AST_FIXUP_FIXUPVISITOR_H
2 #define LLVM_TOOLS_SSML_LIB_AST_FIXUP_FIXUPVISITOR_H
3
4 #include "OperatorDefMap.h"
5 #include "OperatorDefinition.h"
6
7 #include "ssml/AST/Visitor.h"
8 #include "ssml/Common/SourceLocation.h"
9
10 #include <memory>
11 #include <vector>
12
13 namespace ssml {
14 namespace ast {
15 class Node;
16 } // End namespace ast.
17 } // End namespace ssml.
18

```

```

19 namespace ssml {
20 namespace ast {
21 namespace fix {
22 class FixupVisitor : public Visitor {
23 private:
24     OperatorDefinition CurrOpDef;
25     OperatorDefMap OpDefs;
26
27 private:
28     template <typename VectorT, typename ItemT, typename TupleT,
29               typename SeqT>
30     std::shared_ptr<VectorT> operatorFixup(VectorT &Initial);
31     void setNonfixOpDef(SeqShortIdentifier &IDs);
32     void setOpDef(SeqShortIdentifier &IDs,
33                  OperatorDefinition::Associativity,
34                  IntLiteral &Prec);
35 public:
36     void visit(IntLiteral *) override;
37
38     void visit(ShortIdentifier *) override;
39     void visit(LongIdentifier *) override;
40     void visit(IntIdentifier *) override;
41     void visit(SeqShortIdentifier *) override;
42     void visit(SeqLongIdentifier *) override;
43
44     void visit(Match *) override;
45     void visit(SeqMatch *) override;
46
47     void visit(SeqExpression *) override;
48     void visit(LiteralExpression *) override;
49     void visit(LongIdentifierExpression *) override;
50     void visit(TupleExpression *) override;
51     void visit(ListExpression *) override;
52     void visit(ApplyExpression *) override;
53     void visit(OrElseExpression *) override;
54     void visit(AndAlsoExpression *) override;
55     void visit(LetExpression *) override;
56     void visit(IfExpression *) override;
57     void visit(WhileExpression *) override;
58     void visit(CaseExpression *) override;
59     void visit(LambdaExpression *) override;
60
61     void visit(SeqPattern *) override;
62     void visit(LiteralPattern *) override;
63     void visit(WildcardPattern *) override;
64     void visit(ShortIdentifierPattern *) override;
65     void visit(LongIdentifierPattern *) override;
66     void visit(TypePattern *) override;
67     void visit(ApplyPattern *) override;
68     void visit(ListPattern *) override;
69     void visit(TuplePattern *) override;
70
71     void visit(SeqDeclaration *) override;
72     void visit(Root *) override;
73     void visit(ValDeclaration *) override;
74     void visit(NonfixDeclaration *) override;
75     void visit(InfixDeclaration *) override;
76     void visit(InfixRDeclaration *) override;
77     void visit(RestrictedStructDeclaration *) override;
78     void visit(BareStructDeclaration *) override;
79     void visit(AbstractValDeclaration *) override;

```

```

79     void visit(AbstractTypeDeclaration *) override;
80     void visit(AbstractStructDeclaration *) override;
81     void visit(SigDeclaration *) override;
82     void visit(FunPatternDeclaration *) override;
83     void visit(FunDeclaration *) override;
84     void visit(TypeDeclaration *) override;
85     void visit(DatatypeBareInstanceDeclaration *) override;
86     void visit(DatatypeTypedInstanceDeclaration *) override;
87     void visit(BareDatatypeDeclaration *) override;
88     void visit(TypedDatatypeDeclaration *) override;
89     void visit(ShortBareFunctorDeclaration *) override;
90     void visit(ShortRestrictedFunctorDeclaration *) override;
91     void visit(LongBareFunctorDeclaration *) override;
92     void visit(LongRestrictedFunctorDeclaration *) override;
93
94     void visit(LongIdentifierDefinition *) override;
95     void visit(StructDefinition *) override;
96     void visit(AnnotationDefinition *) override;
97     void visit(ShortFunctorDefinition *) override;
98     void visit(LongFunctorDefinition *) override;
99     void visit(SigDefinition *) override;
100
101     void visit(LongIdentifierType *) override;
102     void visit(VariableType *) override;
103     void visit(SeqType *) override;
104     void visit(SeqVariableType *) override;
105     void visit(TupleType *) override;
106     void visit(ApplyType *) override;
107     void visit(ProductType *) override;
108     void visit(FunctionType *) override;
109 };
110 } // End namespace fix.
111 } // End namespace ast.
112 } // End namespace ssml.
113
114 #endif // LLVM_TOOLS_SSML_LIB_AST_FIXUP_FIXUPVISITOR_H

```

Listing 67: lib/AST/Node.cpp

```

1  #include "ssml/AST/Node.h"
2
3  using namespace ssml::ast;
4
5  Node::Node(SourceLocation L) : Location(L) {}
6
7  Node::~~Node() = default;

```

Listing 68: lib/AST/DumpVisitor.cpp

```

1  #include "ssml/AST/DumpVisitor.h"
2  #include "ssml/AST/Declaration.h"
3  #include "ssml/AST/Definition.h"
4  #include "ssml/AST/Pattern.h"
5  #include "ssml/AST/Expression.h"
6  #include "ssml/AST/Literal.h"
7  #include "ssml/AST/Identifier.h"
8  #include "ssml/AST/Type.h"
9  #include "ssml/AST/Match.h"
10 #include "ssml/Common/SourceLocation.h"
11
12 #include "llvm/ADT/StringRef.h"

```

```

13 #include "llvm/Support/raw_ostream.h"
14
15 using namespace llvm;
16 using namespace ssml;
17 using namespace ssml::ast;
18
19 static constexpr unsigned IndentationSize = 2;
20
21 void DumpVisitor::printIndent() {
22     for (unsigned i = 0; i < this->Indent; ++i)
23         outs() << ' ';
24 }
25
26 void DumpVisitor::incrementIndent() { this->Indent +=
    IndentationSize; }
27
28 void DumpVisitor::decrementIndent() { this->Indent -=
    IndentationSize; }
29
30 void DumpVisitor::printAttribute(const char *Name, const
    std::string &Value) {
31     outs() << ' ' << Name << "=" << Value;
32 }
33
34 void DumpVisitor::printAttribute(const char *Name,
35     std::shared_ptr<const
        std::string> Value) {
36     this->printAttribute(Name, *Value);
37 }
38
39 void DumpVisitor::openBeginTag(const char *Tag, SourceLocation L) {
40     this->printIndent();
41     outs() << '<' << Tag;
42     this->printAttribute("location", L.toString());
43 }
44
45 void DumpVisitor::closeBeginTag() {
46     outs() << ">\n";
47     this->incrementIndent();
48 }
49
50 void DumpVisitor::openCloseEndTag(const char *Tag) {
51     this->decrementIndent();
52     this->printIndent();
53     outs() << "</" << Tag << ">\n";
54 }
55
56 void DumpVisitor::visit(IntLiteral *N) {
57     constexpr auto Tag = "IntLiteral";
58     this->openBeginTag(Tag, N->getLocation());
59     this->printAttribute("value", N->getValueString());
60     this->closeBeginTag();
61     this->openCloseEndTag(Tag);
62 }
63
64 void DumpVisitor::visit(RealLiteral *N) {
65     constexpr auto Tag = "RealLiteral";
66     this->openBeginTag(Tag, N->getLocation());
67     this->printAttribute("value", N->getValueString());
68     this->closeBeginTag();
69     this->openCloseEndTag(Tag);
70 }

```

```

71
72 void DumpVisitor::visit(CharLiteral *N) {
73     constexpr auto Tag = "CharLiteral";
74     this->openBeginTag(Tag, N->getLocation());
75     this->printAttribute("value", N->getValueString());
76     this->closeBeginTag();
77     this->openCloseEndTag(Tag);
78 }
79
80 void DumpVisitor::visit(StringLiteral *N) {
81     constexpr auto Tag = "StringLiteral";
82     this->openBeginTag(Tag, N->getLocation());
83     this->printAttribute("value", N->getValueString());
84     this->closeBeginTag();
85     this->openCloseEndTag(Tag);
86 }
87
88 void DumpVisitor::visit(Match *N) {
89     constexpr auto Tag = "Match";
90     this->openBeginTag(Tag, N->getLocation());
91     this->closeBeginTag();
92     N->getPattern()->accept(this);
93     N->getExpr()->accept(this);
94     this->openCloseEndTag(Tag);
95 }
96
97 void DumpVisitor::visit(SeqMatch *N) {
98     constexpr auto Tag = "SeqMatch";
99     this->openBeginTag(Tag, N->getLocation());
100    this->closeBeginTag();
101    for (auto M : *N)
102        M->accept(this);
103    this->openCloseEndTag(Tag);
104 }
105
106 void DumpVisitor::visit(ShortIdentifier *N) {
107     constexpr auto Tag = "ShortIdentifier";
108     this->openBeginTag(Tag, N->getLocation());
109     this->printAttribute("value", N->getID());
110     this->closeBeginTag();
111     this->openCloseEndTag(Tag);
112 }
113
114 void DumpVisitor::visit(LongIdentifier *N) {
115     constexpr auto Tag = "LongIdentifier";
116     this->openBeginTag(Tag, N->getLocation());
117     this->printAttribute("value", N->toString());
118     this->closeBeginTag();
119     this->openCloseEndTag(Tag);
120 }
121
122 void DumpVisitor::visit(IntIdentifier *N) {
123     constexpr auto Tag = "IntIdentifier";
124     this->openBeginTag(Tag, N->getLocation());
125     this->closeBeginTag();
126     N->getLiteral()->accept(this);
127     this->openCloseEndTag(Tag);
128 }
129
130 void DumpVisitor::visit(SeqShortIdentifier *N) {
131     constexpr auto Tag = "SeqShortIdentifier";
132     this->openBeginTag(Tag, N->getLocation());

```

```

133     this->closeBeginTag();
134     for (auto ID : *N)
135         ID->accept(this);
136     this->openCloseEndTag(Tag);
137 }
138
139 void DumpVisitor::visit(SeqLongIdentifier *N) {
140     constexpr auto Tag = "SeqLongIdentifier";
141     this->openBeginTag(Tag, N->getLocation());
142     this->closeBeginTag();
143     for (auto ID : *N)
144         ID->accept(this);
145     this->openCloseEndTag(Tag);
146 }
147
148 void DumpVisitor::visit(SeqExpression *N) {
149     constexpr auto Tag = "SeqExpression";
150     this->openBeginTag(Tag, N->getLocation());
151     this->closeBeginTag();
152     for (auto E : *N)
153         E->accept(this);
154     this->openCloseEndTag(Tag);
155 }
156
157 void DumpVisitor::visit(LiteralExpression *N) {
158     constexpr auto Tag = "LiteralExpression";
159     this->openBeginTag(Tag, N->getLocation());
160     this->closeBeginTag();
161     N->getValue()->accept(this);
162     this->openCloseEndTag(Tag);
163 }
164
165 void DumpVisitor::visit(LongIdentifierExpression *N) {
166     constexpr auto Tag = "LongIdentifierExpression";
167     this->openBeginTag(Tag, N->getLocation());
168     this->closeBeginTag();
169     N->getIDs()->accept(this);
170     this->openCloseEndTag(Tag);
171 }
172
173 void DumpVisitor::visit(TupleExpression *N) {
174     constexpr auto Tag = "TupleExpression";
175     this->openBeginTag(Tag, N->getLocation());
176     this->closeBeginTag();
177     N->getExprs()->accept(this);
178     this->openCloseEndTag(Tag);
179 }
180
181 void DumpVisitor::visit(ListExpression *N) {
182     constexpr auto Tag = "ListExpression";
183     this->openBeginTag(Tag, N->getLocation());
184     this->closeBeginTag();
185     N->getExprs()->accept(this);
186     this->openCloseEndTag(Tag);
187 }
188
189 void DumpVisitor::visit(IdentifierEqualsExpression *N) {
190     constexpr auto Tag = "IdentifierEqualsExpression";
191     this->openBeginTag(Tag, N->getLocation());
192     this->closeBeginTag();
193     N->getID()->accept(this);
194     N->getExpr()->accept(this);

```

```

195     this->openCloseEndTag(Tag);
196 }
197
198 void DumpVisitor::visit(RecordExpression *N) {
199     constexpr auto Tag = "RecordExpression";
200     this->openBeginTag(Tag, N->getLocation());
201     this->closeBeginTag();
202     for (auto E : *N)
203         E->accept(this);
204     this->openCloseEndTag(Tag);
205 }
206
207 void DumpVisitor::visit(SelectorExpression *N) {
208     constexpr auto Tag = "SelectorExpression";
209     this->openBeginTag(Tag, N->getLocation());
210     this->closeBeginTag();
211     N->getID()->accept(this);
212     this->openCloseEndTag(Tag);
213 }
214
215 void DumpVisitor::visit(ApplyExpression *N) {
216     constexpr auto Tag = "ApplyExpression";
217     this->openBeginTag(Tag, N->getLocation());
218     this->closeBeginTag();
219     for (auto E : *N)
220         E->accept(this);
221     this->openCloseEndTag(Tag);
222 }
223
224 void DumpVisitor::visit(TypeExpression *N) {
225     constexpr auto Tag = "TypeExpression";
226     this->openBeginTag(Tag, N->getLocation());
227     this->closeBeginTag();
228     N->getExpr()->accept(this);
229     N->getType()->accept(this);
230     this->openCloseEndTag(Tag);
231 }
232
233 void DumpVisitor::visit(OrElseExpression *N) {
234     constexpr auto Tag = "OrElseExpression";
235     this->openBeginTag(Tag, N->getLocation());
236     this->closeBeginTag();
237     N->getLeftExpr()->accept(this);
238     N->getRightExpr()->accept(this);
239     this->openCloseEndTag(Tag);
240 }
241
242 void DumpVisitor::visit(AndAlsoExpression *N) {
243     constexpr auto Tag = "AndAlsoExpression";
244     this->openBeginTag(Tag, N->getLocation());
245     this->closeBeginTag();
246     N->getLeftExpr()->accept(this);
247     N->getRightExpr()->accept(this);
248     this->openCloseEndTag(Tag);
249 }
250
251 void DumpVisitor::visit(LetExpression *N) {
252     constexpr auto Tag = "LetExpression";
253     this->openBeginTag(Tag, N->getLocation());
254     this->closeBeginTag();
255     N->getDecls()->accept(this);
256     N->getExprs()->accept(this);

```

```

257     this->openCloseEndTag(Tag);
258 }
259
260 void DumpVisitor::visit(IfExpression *N) {
261     constexpr auto Tag = "IfExpression";
262     this->openBeginTag(Tag, N->getLocation());
263     this->closeBeginTag();
264     N->getCondExpr()->accept(this);
265     N->getThenExpr()->accept(this);
266     N->getElseExpr()->accept(this);
267     this->openCloseEndTag(Tag);
268 }
269
270 void DumpVisitor::visit(WhileExpression *N) {
271     constexpr auto Tag = "WhileExpression";
272     this->openBeginTag(Tag, N->getLocation());
273     this->closeBeginTag();
274     N->getCondExpr()->accept(this);
275     N->getBodyExpr()->accept(this);
276     this->openCloseEndTag(Tag);
277 }
278
279 void DumpVisitor::visit(CaseExpression *N) {
280     constexpr auto Tag = "CaseExpression";
281     this->openBeginTag(Tag, N->getLocation());
282     this->closeBeginTag();
283     N->getExpr()->accept(this);
284     N->getCases()->accept(this);
285     this->openCloseEndTag(Tag);
286 }
287
288 void DumpVisitor::visit(LambdaExpression *N) {
289     constexpr auto Tag = "LambdaExpression";
290     this->openBeginTag(Tag, N->getLocation());
291     this->closeBeginTag();
292     N->getCases()->accept(this);
293     this->openCloseEndTag(Tag);
294 }
295
296 void DumpVisitor::visit(SeqPattern *N) {
297     constexpr auto Tag = "SeqPattern";
298     this->openBeginTag(Tag, N->getLocation());
299     this->closeBeginTag();
300     for (auto P : *N)
301         P->accept(this);
302     this->openCloseEndTag(Tag);
303 }
304
305 void DumpVisitor::visit(LiteralPattern *N) {
306     constexpr auto Tag = "LiteralPattern";
307     this->openBeginTag(Tag, N->getLocation());
308     this->closeBeginTag();
309     N->getValue()->accept(this);
310     this->openCloseEndTag(Tag);
311 }
312
313 void DumpVisitor::visit(WildcardPattern *N) {
314     constexpr auto Tag = "WildcardPattern";
315     this->openBeginTag(Tag, N->getLocation());
316     this->closeBeginTag();
317     this->openCloseEndTag(Tag);
318 }

```

```

319
320 void DumpVisitor::visit(ShortIdentifierPattern *N) {
321     constexpr auto Tag = "ShortIdentifierPattern";
322     this->openBeginTag(Tag, N->getLocation());
323     this->closeBeginTag();
324     N->getID()->accept(this);
325     this->openCloseEndTag(Tag);
326 }
327
328 void DumpVisitor::visit(LongIdentifierPattern *N) {
329     constexpr auto Tag = "LongIdentifierPattern";
330     this->openBeginTag(Tag, N->getLocation());
331     if (N->isPrefixedWithOpKey())
332         this->printAttribute("opPrefix", "true");
333     else
334         this->printAttribute("opPrefix", "false");
335     this->closeBeginTag();
336     N->getIDs()->accept(this);
337     this->openCloseEndTag(Tag);
338 }
339
340 void DumpVisitor::visit(TypePattern *N) {
341     constexpr auto Tag = "TypePattern";
342     this->openBeginTag(Tag, N->getLocation());
343     this->closeBeginTag();
344     N->getPattern()->accept(this);
345     N->getType()->accept(this);
346     this->openCloseEndTag(Tag);
347 }
348
349 void DumpVisitor::visit(AsPattern *N) {
350     constexpr auto Tag = "AsPattern";
351     this->openBeginTag(Tag, N->getLocation());
352     this->closeBeginTag();
353     N->getLeftPattern()->accept(this);
354     N->getRightPattern()->accept(this);
355     this->openCloseEndTag(Tag);
356 }
357
358 void DumpVisitor::visit(ApplyPattern *N) {
359     constexpr auto Tag = "ApplyPattern";
360     this->openBeginTag(Tag, N->getLocation());
361     this->closeBeginTag();
362     for (auto P : *N)
363         P->accept(this);
364     this->openCloseEndTag(Tag);
365 }
366
367 void DumpVisitor::visit(IdentifierEqualsPattern *N) {
368     constexpr auto Tag = "IdentifierEqualsPattern";
369     this->openBeginTag(Tag, N->getLocation());
370     this->closeBeginTag();
371     N->getID()->accept(this);
372     N->getPattern()->accept(this);
373     this->openCloseEndTag(Tag);
374 }
375
376 void DumpVisitor::visit(RecordPattern *N) {
377     constexpr auto Tag = "RecordPattern";
378     this->openBeginTag(Tag, N->getLocation());
379     if (N->isEllipsisTerminated())
380         this->printAttribute("ellipsis", "true");

```

```

381     else
382         this->printAttribute("ellipsis", "false");
383     this->closeBeginTag();
384     for (auto P : *N)
385         P->accept(this);
386     this->openCloseEndTag(Tag);
387 }
388
389 void DumpVisitor::visit(ListPattern *N) {
390     constexpr auto Tag = "ListPattern";
391     this->openBeginTag(Tag, N->getLocation());
392     this->closeBeginTag();
393     N->getPatterns()->accept(this);
394     this->openCloseEndTag(Tag);
395 }
396
397 void DumpVisitor::visit(TuplePattern *N) {
398     constexpr auto Tag = "TuplePattern";
399     this->openBeginTag(Tag, N->getLocation());
400     this->closeBeginTag();
401     N->getPatterns()->accept(this);
402     this->openCloseEndTag(Tag);
403 }
404
405 void DumpVisitor::visit(SeqDeclaration *N) {
406     constexpr auto Tag = "SeqDeclaration";
407     this->openBeginTag(Tag, N->getLocation());
408     this->closeBeginTag();
409     for (auto D : *N)
410         D->accept(this);
411     this->openCloseEndTag(Tag);
412 }
413
414 void DumpVisitor::visit(Root *N) {
415     constexpr auto Tag = "Root";
416     this->openBeginTag(Tag, N->getLocation());
417     this->closeBeginTag();
418     for (auto D : *N)
419         D->accept(this);
420     this->openCloseEndTag(Tag);
421 }
422
423 void DumpVisitor::visit(ValDeclaration *N) {
424     constexpr auto Tag = "ValDeclaration";
425     this->openBeginTag(Tag, N->getLocation());
426     this->closeBeginTag();
427     N->getDest()->accept(this);
428     N->getSource()->accept(this);
429     this->openCloseEndTag(Tag);
430 }
431
432 void DumpVisitor::visit(OpenDeclaration *N) {
433     constexpr auto Tag = "OpenDeclaration";
434     this->openBeginTag(Tag, N->getLocation());
435     this->closeBeginTag();
436     N->getIDs()->accept(this);
437     this->openCloseEndTag(Tag);
438 }
439
440 void DumpVisitor::visit(NonfixDeclaration *N) {
441     constexpr auto Tag = "NonfixDeclaration";
442     this->openBeginTag(Tag, N->getLocation());

```

```

443     this->closeBeginTag();
444     N->getIDs()->accept(this);
445     this->openCloseEndTag(Tag);
446 }
447
448 void DumpVisitor::visit(InfixDeclaration *N) {
449     constexpr auto Tag = "InfixDeclaration";
450     this->openBeginTag(Tag, N->getLocation());
451     this->printAttribute("precedence",
452         N->getPrecedence()->getValueString());
453     this->closeBeginTag();
454     N->getIDs()->accept(this);
455     this->openCloseEndTag(Tag);
456 }
457
458 void DumpVisitor::visit(InfixRDeclaration *N) {
459     constexpr auto Tag = "InfixRDeclaration";
460     this->openBeginTag(Tag, N->getLocation());
461     this->printAttribute("precedence",
462         N->getPrecedence()->getValueString());
463     this->closeBeginTag();
464     N->getIDs()->accept(this);
465     this->openCloseEndTag(Tag);
466 }
467
468 void DumpVisitor::visit(BareStructDeclaration *N) {
469     constexpr auto Tag = "BareStructDeclaration";
470     this->openBeginTag(Tag, N->getLocation());
471     this->closeBeginTag();
472     N->getID()->accept(this);
473     N->getStructDef()->accept(this);
474     this->openCloseEndTag(Tag);
475 }
476
477 void DumpVisitor::visit(RestrictedStructDeclaration *N) {
478     constexpr auto Tag = "RestrictedStructDeclaration";
479     this->openBeginTag(Tag, N->getLocation());
480     if (N->isSigTransparent())
481         this->printAttribute("sigTransparent", "true");
482     else
483         this->printAttribute("sigTransparent", "false");
484     this->closeBeginTag();
485     N->getID()->accept(this);
486     N->getSigDef()->accept(this);
487     N->getStructDef()->accept(this);
488     this->openCloseEndTag(Tag);
489 }
490
491 void DumpVisitor::visit(AbstractValDeclaration *N) {
492     constexpr auto Tag = "AbstractValDeclaration";
493     this->openBeginTag(Tag, N->getLocation());
494     this->closeBeginTag();
495     N->getID()->accept(this);
496     N->getType()->accept(this);
497     this->openCloseEndTag(Tag);
498 }
499
500 void DumpVisitor::visit(AbstractTypeDeclaration *N) {
501     constexpr auto Tag = "AbstractTypeDeclaration";
502     this->openBeginTag(Tag, N->getLocation());
503     this->closeBeginTag();
504     N->getID()->accept(this);

```

```

503     this->openCloseEndTag(Tag);
504 }
505
506 void DumpVisitor::visit(AbstractStructDeclaration *N) {
507     constexpr auto Tag = "AbstractStructDeclaration";
508     this->openBeginTag(Tag, N->getLocation());
509     this->closeBeginTag();
510     N->getID()->accept(this);
511     N->getSigDef()->accept(this);
512     this->openCloseEndTag(Tag);
513 }
514
515 void DumpVisitor::visit(SharingTypeDeclaration *N) {
516     constexpr auto Tag = "SharingTypeDeclaration";
517     this->openBeginTag(Tag, N->getLocation());
518     this->closeBeginTag();
519     for (auto T : *N)
520         T->accept(this);
521     this->openCloseEndTag(Tag);
522 }
523
524 void DumpVisitor::visit(SigDeclaration *N) {
525     constexpr auto Tag = "SigDeclaration";
526     this->openBeginTag(Tag, N->getLocation());
527     this->closeBeginTag();
528     N->getID()->accept(this);
529     N->getSigDef()->accept(this);
530     this->openCloseEndTag(Tag);
531 }
532
533 void DumpVisitor::visit(FunPatternDeclaration *N) {
534     constexpr auto Tag = "FunPatternDeclaration";
535     this->openBeginTag(Tag, N->getLocation());
536     this->closeBeginTag();
537     N->getIDParams()->accept(this);
538     N->getDef()->accept(this);
539     this->openCloseEndTag(Tag);
540 }
541
542 void DumpVisitor::visit(FunDeclaration *N) {
543     constexpr auto Tag = "FunDeclaration";
544     this->openBeginTag(Tag, N->getLocation());
545     this->closeBeginTag();
546     for (auto P : *N)
547         P->accept(this);
548     this->openCloseEndTag(Tag);
549 }
550
551 void DumpVisitor::visit(TypeDeclaration *N) {
552     constexpr auto Tag = "TypeDeclaration";
553     this->openBeginTag(Tag, N->getLocation());
554     this->closeBeginTag();
555     N->getID()->accept(this);
556     N->getType()->accept(this);
557     this->openCloseEndTag(Tag);
558 }
559
560 void DumpVisitor::visit(DatatypeBareInstanceDeclaration *N) {
561     constexpr auto Tag = "DatatypeBareInstanceDeclaration";
562     this->openBeginTag(Tag, N->getLocation());
563     this->closeBeginTag();
564     N->getID()->accept(this);

```

```

565     this->openCloseEndTag(Tag);
566 }
567
568 void DumpVisitor::visit(DatatypeTypedInstanceDeclaration *N) {
569     constexpr auto Tag = "DatatypeTypedInstanceDeclaration";
570     this->openBeginTag(Tag, N->getLocation());
571     this->closeBeginTag();
572     N->getID()->accept(this);
573     N->getType()->accept(this);
574     this->openCloseEndTag(Tag);
575 }
576
577 void DumpVisitor::visit(BareDatatypeDeclaration *N) {
578     constexpr auto Tag = "BareDatatypeDeclaration";
579     this->openBeginTag(Tag, N->getLocation());
580     this->closeBeginTag();
581     N->getID()->accept(this);
582     N->getDef()->accept(this);
583     this->openCloseEndTag(Tag);
584 }
585
586 void DumpVisitor::visit(TypedDatatypeDeclaration *N) {
587     constexpr auto Tag = "TypedDatatypeDeclaration";
588     this->openBeginTag(Tag, N->getLocation());
589     this->closeBeginTag();
590     N->getTypeParams()->accept(this);
591     N->getID()->accept(this);
592     N->getDef()->accept(this);
593     this->openCloseEndTag(Tag);
594 }
595
596 void DumpVisitor::visit(ShortBareFunctorDeclaration *N) {
597     constexpr auto Tag = "ShortBareFunctorDeclaration";
598     this->openBeginTag(Tag, N->getLocation());
599     this->closeBeginTag();
600     N->getFunctorID()->accept(this);
601     N->getParamID()->accept(this);
602     N->getParamSigDef()->accept(this);
603     N->getFunctorStructDef()->accept(this);
604     this->openCloseEndTag(Tag);
605 }
606
607 void DumpVisitor::visit(ShortRestrictedFunctorDeclaration *N) {
608     constexpr auto Tag = "ShortRestrictedFunctorDeclaration";
609     this->openBeginTag(Tag, N->getLocation());
610     if (N->isFunctorSigDefTransparent())
611         this->printAttribute("transparent", "true");
612     else
613         this->printAttribute("transparent", "false");
614     this->closeBeginTag();
615     N->getFunctorID()->accept(this);
616     N->getParamID()->accept(this);
617     N->getParamSigDef()->accept(this);
618     N->getFunctorSigDef()->accept(this);
619     N->getFunctorStructDef()->accept(this);
620     this->openCloseEndTag(Tag);
621 }
622
623 void DumpVisitor::visit(LongBareFunctorDeclaration *N) {
624     constexpr auto Tag = "LongBareFunctorDeclaration";
625     this->openBeginTag(Tag, N->getLocation());
626     this->closeBeginTag();

```

```

627     N->getFunctorID()->accept(this);
628     N->getParams()->accept(this);
629     N->getFunctorStructDef()->accept(this);
630     this->openCloseEndTag(Tag);
631 }
632
633 void DumpVisitor::visit(LongRestrictedFunctorDeclaration *N) {
634     constexpr auto Tag = "LongRestrictedFunctorDeclaration";
635     this->openBeginTag(Tag, N->getLocation());
636     if (N->isFunctorSigDefTransparent())
637         this->printAttribute("transparent", "true");
638     else
639         this->printAttribute("transparent", "false");
640     this->closeBeginTag();
641     N->getFunctorID()->accept(this);
642     N->getParams()->accept(this);
643     N->getFunctorSigDef()->accept(this);
644     N->getFunctorStructDef()->accept(this);
645     this->openCloseEndTag(Tag);
646 }
647
648 void DumpVisitor::visit(LongIdentifierDefinition *N) {
649     constexpr auto Tag = "LongIdentifierDefinition";
650     this->openBeginTag(Tag, N->getLocation());
651     this->closeBeginTag();
652     N->getID()->accept(this);
653     this->openCloseEndTag(Tag);
654 }
655
656 void DumpVisitor::visit(StructDefinition *N) {
657     constexpr auto Tag = "StructDefinition";
658     this->openBeginTag(Tag, N->getLocation());
659     this->closeBeginTag();
660     N->getDecls()->accept(this);
661     this->openCloseEndTag(Tag);
662 }
663
664 void DumpVisitor::visit(AnnotationDefinition *N) {
665     constexpr auto Tag = "AnnotationDefinition";
666     this->openBeginTag(Tag, N->getLocation());
667     if (N->isTransparent())
668         this->printAttribute("transparent", "true");
669     else
670         this->printAttribute("transparent", "false");
671     this->closeBeginTag();
672     N->getStructDef()->accept(this);
673     N->getSigDef()->accept(this);
674     this->openCloseEndTag(Tag);
675 }
676
677 void DumpVisitor::visit(ShortFunctorDefinition *N) {
678     constexpr auto Tag = "ShortFunctorDefinition";
679     this->openBeginTag(Tag, N->getLocation());
680     this->closeBeginTag();
681     N->getFunctorID()->accept(this);
682     N->getStructDef()->accept(this);
683     this->openCloseEndTag(Tag);
684 }
685
686 void DumpVisitor::visit(LongFunctorDefinition *N) {
687     constexpr auto Tag = "LongFunctorDefinition";
688     this->openBeginTag(Tag, N->getLocation());

```

```

689     this->closeBeginTag();
690     N->getFunctorID()->accept(this);
691     N->getDecls()->accept(this);
692     this->openCloseEndTag(Tag);
693 }
694
695 void DumpVisitor::visit(SigDefinition *N) {
696     constexpr auto Tag = "SigDefinition";
697     this->openBeginTag(Tag, N->getLocation());
698     this->closeBeginTag();
699     N->getDecls()->accept(this);
700     this->openCloseEndTag(Tag);
701 }
702
703 void DumpVisitor::visit(LongIdentifierType *N) {
704     constexpr auto Tag = "LongIdentifierType";
705     this->openBeginTag(Tag, N->getLocation());
706     this->closeBeginTag();
707     N->getIDs()->accept(this);
708     this->openCloseEndTag(Tag);
709 }
710
711 void DumpVisitor::visit(VariableType *N) {
712     constexpr auto Tag = "VariableType";
713     this->openBeginTag(Tag, N->getLocation());
714     this->closeBeginTag();
715     N->getID()->accept(this);
716     this->openCloseEndTag(Tag);
717 }
718
719 void DumpVisitor::visit(SeqType *N) {
720     constexpr auto Tag = "SeqType";
721     this->openBeginTag(Tag, N->getLocation());
722     this->closeBeginTag();
723     for (auto T : *N)
724         T->accept(this);
725     this->openCloseEndTag(Tag);
726 }
727
728 void DumpVisitor::visit(SeqVariableType *N) {
729     constexpr auto Tag = "SeqVariableType";
730     this->openBeginTag(Tag, N->getLocation());
731     this->closeBeginTag();
732     for (auto T : *N)
733         T->accept(this);
734     this->openCloseEndTag(Tag);
735 }
736
737 void DumpVisitor::visit(TupleType *N) {
738     constexpr auto Tag = "TupleType";
739     this->openBeginTag(Tag, N->getLocation());
740     this->closeBeginTag();
741     N->getTypes()->accept(this);
742     this->openCloseEndTag(Tag);
743 }
744
745 void DumpVisitor::visit(IdentifierColonType *N) {
746     constexpr auto Tag = "IdentifierColonType";
747     this->openBeginTag(Tag, N->getLocation());
748     this->closeBeginTag();
749     N->getIDs()->accept(this);
750     N->getType()->accept(this);

```



```

751     this->openCloseEndTag(Tag);
752 }
753
754 void DumpVisitor::visit(RecordType *N) {
755     constexpr auto Tag = "RecordType";
756     this->openBeginTag(Tag, N->getLocation());
757     this->closeBeginTag();
758     for (auto T : *N)
759         T->accept(this);
760     this->openCloseEndTag(Tag);
761 }
762
763 void DumpVisitor::visit(ApplyType *N) {
764     constexpr auto Tag = "ApplyType";
765     this->openBeginTag(Tag, N->getLocation());
766     this->closeBeginTag();
767     for (auto T : *N)
768         T->accept(this);
769     this->openCloseEndTag(Tag);
770 }
771
772 void DumpVisitor::visit(ProductType *N) {
773     constexpr auto Tag = "ProductType";
774     this->openBeginTag(Tag, N->getLocation());
775     this->closeBeginTag();
776     for (auto T : *N)
777         T->accept(this);
778     this->openCloseEndTag(Tag);
779 }
780
781 void DumpVisitor::visit(FunctionType *N) {
782     constexpr auto Tag = "FunctionType";
783     this->openBeginTag(Tag, N->getLocation());
784     this->closeBeginTag();
785     for (auto T : *N)
786         T->accept(this);
787     this->openCloseEndTag(Tag);
788 }

```

Listing 69: lib/AST/Definition.cpp

```

1  #include "ssml/AST/Definition.h"
2
3  #include "ssml/AST/Identifier.h"
4  #include "ssml/AST/Visitor.h"
5  #include "ssml/AST/Declaration.h"
6
7  using namespace ssml::ast;
8
9  Definition::Definition(SourceLocation L) : Node(L) {}
10
11 LongIdentifierDefinition::LongIdentifierDefinition(
12     SourceLocation L, std::shared_ptr<LongIdentifier> ID)
13     : Definition(L), ID(ID) {}
14
15 void LongIdentifierDefinition::accept(Visitor *V) {
16     V->visit(this); }
17
18 StructDefinition::StructDefinition(SourceLocation L,
19     std::shared_ptr<SeqDeclaration>
20     Ds)

```

```

19     : Definition(L), Decls(Ds) {}
20
21 void StructDefinition::accept(Visitor *V) { V->visit(this); }
22
23 AnnotationDefinition::AnnotationDefinition(SourceLocation L,
24                                             std::shared_ptr<Definition>
25                                             Struct,
26                                             std::shared_ptr<Definition>
27                                             Sig,
28                                             bool T)
29     : Definition(L), StructDef(Struct), SigDef(Sig),
30       Transparent(T) {}
31
32 void AnnotationDefinition::accept(Visitor *V) { V->visit(this); }
33
34 ShortFunctorDefinition::ShortFunctorDefinition(
35     SourceLocation L, std::shared_ptr<LongIdentifier> ID,
36     std::shared_ptr<Definition> Struct)
37     : Definition(L), FunctorID(ID), StructDef(Struct) {}
38
39 void ShortFunctorDefinition::accept(Visitor *V) { V->visit(this); }
40
41 LongFunctorDefinition::LongFunctorDefinition(SourceLocation L,
42                                             std::shared_ptr<LongIdentifier>
43                                             ID,
44                                             std::shared_ptr<SeqDeclaration>
45                                             Ds)
46     : Definition(L), FunctorID(ID), Decls(Ds) {}
47
48 void LongFunctorDefinition::accept(Visitor *V) { V->visit(this); }
49
50 SigDefinition::SigDefinition(SourceLocation L,
51                             std::shared_ptr<SeqDeclaration> Ds)
52     : Definition(L), Decls(Ds) {}
53
54 void SigDefinition::accept(Visitor *V) { V->visit(this); }

```

Listing 70: lib/AST/Declaration.cpp

```

1 #include "ssml/AST/Declaration.h"
2
3 #include "ssml/AST/Visitor.h"
4 #include "ssml/AST/Pattern.h"
5 #include "ssml/AST/Expression.h"
6 #include "ssml/AST/Identifier.h"
7 #include "ssml/AST/Definition.h"
8 #include "ssml/AST/Type.h"
9
10 using namespace ssml::ast;
11
12 Declaration::Declaration(SourceLocation L) : Node(L) {}
13
14 SeqDeclaration::SeqDeclaration(SourceLocation L) : Declaration(L)
15 {}
16
17 SeqDeclaration::SeqDeclaration(SourceLocation L,
18                             std::shared_ptr<Declaration> First)
19     : Declaration(L),
20       std::vector<std::shared_ptr<Declaration>>{First} {}
21
22 void SeqDeclaration::accept(Visitor *V) { V->visit(this); }

```

```

21
22 Root::Root(SourceLocation L) : SeqDeclaration(L) {}
23
24 Root::Root(SourceLocation L, std::shared_ptr<Declaration> F)
25   : SeqDeclaration(L, F) {}
26
27 void Root::accept(Visitor *V) { V->visit(this); }
28
29 ValDeclaration::ValDeclaration(SourceLocation L,
30   std::shared_ptr<Pattern> Dest,
31   std::shared_ptr<Expression> Src)
32   : Declaration(L, Dest(Dest), Source(Src)) {}
33
34 void ValDeclaration::accept(Visitor *V) { V->visit(this); }
35
36 OpenDeclaration::OpenDeclaration(SourceLocation L,
37   std::shared_ptr<SeqLongIdentifier>
38   IDs)
39   : Declaration(L, IDs(IDs)) {}
40
41 void OpenDeclaration::accept(Visitor *V) { V->visit(this); }
42
43 NonfixDeclaration::NonfixDeclaration(SourceLocation L,
44   std::shared_ptr<SeqShortIdentifier>
45   IDs)
46   : Declaration(L, IDs(IDs)) {}
47
48 void NonfixDeclaration::accept(Visitor *V) { V->visit(this); }
49
50 InfixDeclaration::InfixDeclaration(SourceLocation L,
51   std::shared_ptr<IntLiteral>
52   Prec,
53   std::shared_ptr<SeqShortIdentifier>
54   IDs)
55   : Declaration(L, Precedence(Prec), Idens(IDs)) {}
56
57 void InfixDeclaration::accept(Visitor *V) { V->visit(this); }
58
59 InfixRDeclaration::InfixRDeclaration(SourceLocation L,
60   std::shared_ptr<IntLiteral>
61   Prec,
62   std::shared_ptr<SeqShortIdentifier>
63   IDs)
64   : Declaration(L, Precedence(Prec), Idens(IDs)) {}
65
66 void InfixRDeclaration::accept(Visitor *V) { V->visit(this); }
67
68 RestrictingSigDefinition::RestrictingSigDefinition(
69   std::shared_ptr<Definition> D, bool B)
70   : Def(D), IsTransparent(B) {}
71
72 BareStructDeclaration::BareStructDeclaration(
73   SourceLocation L, std::shared_ptr<ShortIdentifier> ID,
74   std::shared_ptr<Definition> Def)
75   : Declaration(L, ID(ID), StructDef(Def)) {}
76
77 void BareStructDeclaration::accept(Visitor *V) { V->visit(this); }
78
79 RestrictedStructDeclaration::RestrictedStructDeclaration(
80   SourceLocation L, std::shared_ptr<ShortIdentifier> ID,
81   RestrictingSigDefinition SigDef, std::shared_ptr<Definition>
82   StructDef)

```

```

75         : BareStructDeclaration(L, ID, StructDef), SigDef(SigDef) {}
76
77 void RestrictedStructDeclaration::accept(Visitor *V) {
78     V->visit(this); }
79
80 AbstractValDeclaration::AbstractValDeclaration(
81     SourceLocation L, std::shared_ptr<ShortIdentifier> ID,
82     std::shared_ptr<Type> T)
83     : Declaration(L), ID(ID), Typ(T) {}
84
85 void AbstractValDeclaration::accept(Visitor *V) { V->visit(this); }
86
87 AbstractTypeDeclaration::AbstractTypeDeclaration(
88     SourceLocation L, std::shared_ptr<ShortIdentifier> ID)
89     : Declaration(L), ID(ID) {}
90
91 void AbstractTypeDeclaration::accept(Visitor *V) { V->visit(this); }
92
93 AbstractStructDeclaration::AbstractStructDeclaration(
94     SourceLocation L, std::shared_ptr<ShortIdentifier> ID,
95     std::shared_ptr<Definition> D)
96     : Declaration(L), ID(ID), SigDef(D) {}
97
98 void AbstractStructDeclaration::accept(Visitor *V) {
99     V->visit(this); }
100
101 SharingTypeDeclaration::SharingTypeDeclaration(
102     SourceLocation L, std::shared_ptr<LongIdentifier> First)
103     : Declaration(L),
104       std::vector<std::shared_ptr<LongIdentifier>>{First} {}
105
106 void SharingTypeDeclaration::accept(Visitor *V) { V->visit(this); }
107
108 SigDeclaration::SigDeclaration(SourceLocation L,
109                                std::shared_ptr<ShortIdentifier> ID,
110                                std::shared_ptr<Definition> Def)
111     : Declaration(L), ID(ID), Def(Def) {}
112
113 void SigDeclaration::accept(Visitor *V) { V->visit(this); }
114
115 FunPatternDeclaration::FunPatternDeclaration(SourceLocation L,
116                                               std::shared_ptr<Pattern>
117                                               P,
118                                               std::shared_ptr<Expression>
119                                               Def)
120     : Declaration(L), IDParams(P), Def(Def) {}
121
122 void FunPatternDeclaration::accept(Visitor *V) { V->visit(this); }
123
124 FunDeclaration::FunDeclaration(SourceLocation L,
125                                std::shared_ptr<FunPatternDeclaration>
126                                F)
127     : Declaration(L),
128       std::vector<std::shared_ptr<FunPatternDeclaration>>{F} {}
129
130 void FunDeclaration::accept(Visitor *V) { V->visit(this); }
131
132 TypeDeclaration::TypeDeclaration(SourceLocation L,
133                                   std::shared_ptr<ShortIdentifier>
134                                   ID,
135                                   std::shared_ptr<Type> Typ)

```

```

128         : Declaration(L), ID(ID), Typ(Typ) {}
129
130 void TypeDeclaration::accept(Visitor *V) { V->visit(this); }
131
132 DatatypeBareInstanceDeclaration::DatatypeBareInstanceDeclaration(
133     SourceLocation L, std::shared_ptr<ShortIdentifier> ID)
134     : Declaration(L), ID(ID) {}
135
136 void DatatypeBareInstanceDeclaration::accept(Visitor *V) {
137     V->visit(this); }
138
139 DatatypeTypedInstanceDeclaration::DatatypeTypedInstanceDeclaration(
140     SourceLocation L, std::shared_ptr<ShortIdentifier> ID,
141     std::shared_ptr<Type> Typ)
142     : DatatypeBareInstanceDeclaration(L, ID), Typ(Typ) {}
143
144 void DatatypeTypedInstanceDeclaration::accept(Visitor *V) {
145     V->visit(this); }
146
147 BareDatatypeDeclaration::BareDatatypeDeclaration(
148     SourceLocation L, std::shared_ptr<ShortIdentifier> ID,
149     std::shared_ptr<SeqDeclaration> D)
150     : Declaration(L), ID(ID), Def(D) {}
151
152 void BareDatatypeDeclaration::accept(Visitor *V) { V->visit(this); }
153
154 TypedDatatypeDeclaration::TypedDatatypeDeclaration(
155     SourceLocation L, std::shared_ptr<SeqVariableType> TypeParams,
156     std::shared_ptr<ShortIdentifier> ID,
157     std::shared_ptr<SeqDeclaration> Def)
158     : BareDatatypeDeclaration(L, ID, Def), TypeParams(TypeParams)
159     {}
160
161 void TypedDatatypeDeclaration::accept(Visitor *V) {
162     V->visit(this); }
163
164 ShortBareFunctorDeclaration::ShortBareFunctorDeclaration(
165     SourceLocation L, std::shared_ptr<ShortIdentifier> FunctorID,
166     std::shared_ptr<ShortIdentifier> ParamID,
167     std::shared_ptr<Definition> ParamSig,
168     std::shared_ptr<Definition> StructDef)
169     : Declaration(L, FunctorID(FunctorID), ParamID(ParamID),
170     ParamSigDef(ParamSig), FunctorStructDef(StructDef) {}
171
172 void ShortBareFunctorDeclaration::accept(Visitor *V) {
173     V->visit(this); }
174
175 ShortRestrictedFunctorDeclaration::ShortRestrictedFunctorDeclaration(
176     SourceLocation L, std::shared_ptr<ShortIdentifier> FunctorID,
177     std::shared_ptr<ShortIdentifier> ParamID,
178     std::shared_ptr<Definition> ParamSig,
179     RestrictingSigDefinition FunctorSigDef,
180     std::shared_ptr<Definition> StructDef)
181     : ShortBareFunctorDeclaration(L, FunctorID, ParamID, ParamSig,
182     StructDef),
183     FunctorSigDef(FunctorSigDef) {}
184
185 void ShortRestrictedFunctorDeclaration::accept(Visitor *V) {
186     V->visit(this); }
187
188 LongBareFunctorDeclaration::LongBareFunctorDeclaration(

```

```

180     SourceLocation L, std::shared_ptr<ShortIdentifier> FunctorID,
181     std::shared_ptr<SeqDeclaration> Params,
182     std::shared_ptr<Definition> StructDef)
183     : Declaration(L, FunctorID(FunctorID), Params(Params),
184     FunctorStructDef(StructDef) {})
185
186 void LongBareFunctorDeclaration::accept(Visitor *V) {
187     V->visit(this); }
188
189 LongRestrictedFunctorDeclaration::LongRestrictedFunctorDeclaration(
190     SourceLocation L, std::shared_ptr<ShortIdentifier> FunctorID,
191     std::shared_ptr<SeqDeclaration> Params,
192     RestrictingSigDefinition SigDef,
193     std::shared_ptr<Definition> StructDef)
194     : LongBareFunctorDeclaration(L, FunctorID, Params, StructDef),
195     FunctorSigDef(SigDef) {}
196
197 void LongRestrictedFunctorDeclaration::accept(Visitor *V) {
198     V->visit(this); }

```

Listing 71: lib/AST/Expression.cpp

```

1  #include "ssml/AST/Expression.h"
2  #include "ssml/AST/Visitor.h"
3  #include "ssml/AST/Identifier.h"
4  #include "ssml/AST/Type.h"
5  #include "ssml/AST/Declaration.h"
6  #include "ssml/AST/Match.h"
7
8  #include <cassert>
9
10 using namespace ssml::ast;
11
12 Expression::Expression(SourceLocation L) : Node(L) {}
13
14 LongIdentifierExpression *Expression::asLongIdentifierExpression()
15 {
16     return nullptr;
17 }
18 TupleExpression *Expression::asTupleExpression() { return nullptr; }
19
20 SeqExpression::SeqExpression(SourceLocation L) : Expression(L) {}
21
22 SeqExpression::SeqExpression(SourceLocation L,
23     std::shared_ptr<Expression> F)
24     : Expression(L), std::vector<std::shared_ptr<Expression>>{F} {}
25
26 void SeqExpression::accept(Visitor *V) { V->visit(this); }
27
28 LiteralExpression::LiteralExpression(SourceLocation L,
29     std::shared_ptr<Literal> Lit)
30     : Expression(L), Value(Lit) {}
31
32 void LiteralExpression::accept(Visitor *V) { V->visit(this); }
33
34 LongIdentifierExpression::LongIdentifierExpression(
35     SourceLocation L, std::shared_ptr<LongIdentifier> ID, bool
36     OpKey)
37     : Expression(L), ID(ID), opKeyPrefixed(OpKey) {}

```

```

36
37 LongIdentifierExpression *
38 LongIdentifierExpression::asLongIdentifierExpression() {
39     return this;
40 }
41
42 void LongIdentifierExpression::accept(Visitor *V) {
43     V->visit(this); }
44
45 TupleExpression::TupleExpression(SourceLocation L,
46                                 std::shared_ptr<SeqExpression> Es)
47     : Expression(L), Expressions(Es) {
48     assert(Es->size() != 1);
49 }
50
51 TupleExpression *TupleExpression::asTupleExpression() { return
52     this; }
53
54 void TupleExpression::accept(Visitor *V) { V->visit(this); }
55
56 ListExpression::ListExpression(SourceLocation L,
57                                std::shared_ptr<SeqExpression> Es)
58     : Expression(L), Expressions(Es) {}
59
60 void ListExpression::accept(Visitor *V) { V->visit(this); }
61
62 IdentifierEqualsExpression::IdentifierEqualsExpression(
63     SourceLocation L, std::shared_ptr<Identifier> LHS,
64     std::shared_ptr<Expression> RHS)
65     : Expression(L), Iden(LHS), Expr(RHS) {}
66
67 void IdentifierEqualsExpression::accept(Visitor *V) {
68     V->visit(this); }
69
70 RecordExpression::RecordExpression(
71     SourceLocation L, std::shared_ptr<IdentifierEqualsExpression>
72     F)
73     : Expression(L),
74     std::vector<std::shared_ptr<IdentifierEqualsExpression>>{F}
75     {}
76
77 void RecordExpression::accept(Visitor *V) { V->visit(this); }
78
79 SelectorExpression::SelectorExpression(SourceLocation L,
80                                        std::shared_ptr<Identifier>
81                                        ID)
82     : Expression(L), ID(ID) {}
83
84 void SelectorExpression::accept(Visitor *V) { V->visit(this); }
85
86 ApplyExpression::ApplyExpression(SourceLocation L,
87                                  std::shared_ptr<Expression> F)
88     : Expression(L), std::vector<std::shared_ptr<Expression>>{F} {}
89
90 void ApplyExpression::accept(Visitor *V) { V->visit(this); }
91
92 TypeExpression::TypeExpression(SourceLocation L,
93                                std::shared_ptr<Expression> E,
94                                std::shared_ptr<Type> T)
95     : Expression(L), Expr(E), Typ(T) {}
96
97 void TypeExpression::accept(Visitor *V) { V->visit(this); }

```

```

91
92 OrElseExpression::OrElseExpression(SourceLocation L,
93                                     std::shared_ptr<Expression> LHS,
94                                     std::shared_ptr<Expression> RHS)
95     : Expression(L), Left(LHS), Right(RHS) {}
96
97 void OrElseExpression::accept(Visitor *V) { V->visit(this); }
98
99 AndAlsoExpression::AndAlsoExpression(SourceLocation L,
100                                       std::shared_ptr<Expression>
101                                       LHS,
102                                       std::shared_ptr<Expression>
103                                       RHS)
104     : Expression(L), Left(LHS), Right(RHS) {}
105
106 void AndAlsoExpression::accept(Visitor *V) { V->visit(this); }
107
108 LetExpression::LetExpression(SourceLocation L,
109                               std::shared_ptr<SeqDeclaration> D,
110                               std::shared_ptr<SeqExpression> E)
111     : Expression(L), Declarations(D), Expressions(E) {}
112
113 void LetExpression::accept(Visitor *V) { V->visit(this); }
114
115 IfExpression::IfExpression(SourceLocation L,
116                             std::shared_ptr<Expression> C,
117                             std::shared_ptr<Expression> T,
118                             std::shared_ptr<Expression> E)
119     : Expression(L), CondExpr(C), ThenExpr(T), ElseExpr(E) {}
120
121 void IfExpression::accept(Visitor *V) { V->visit(this); }
122
123 WhileExpression::WhileExpression(SourceLocation L,
124                                   std::shared_ptr<Expression> C,
125                                   std::shared_ptr<Expression> B)
126     : Expression(L), CondExpr(C), BodyExpr(B) {}
127
128 void WhileExpression::accept(Visitor *V) { V->visit(this); }
129
130 CaseExpression::CaseExpression(SourceLocation L,
131                                std::shared_ptr<Expression> E,
132                                std::shared_ptr<SeqMatch> M)
133     : Expression(L), Expr(E), Cases(M) {}
134
135 void CaseExpression::accept(Visitor *V) { V->visit(this); }
136
137 LambdaExpression::LambdaExpression(SourceLocation L,
138                                     std::shared_ptr<SeqMatch> M)
139     : Expression(L), Cases(M) {}
140
141 void LambdaExpression::accept(Visitor *V) { V->visit(this); }

```

Listing 72: lib/AST/Pattern.cpp

```

1 #include "ssml/AST/Pattern.h"
2 #include "ssml/AST/Visitor.h"
3 #include "ssml/AST/Literal.h"
4 #include "ssml/AST/Identifier.h"
5 #include "ssml/AST/Type.h"
6
7 #include <cassert>

```

```

8
9 using namespace ssml::ast;
10
11 Pattern::Pattern(SourceLocation L) : Node(L) {}
12
13 TypePattern *Pattern::asTypePattern() { return nullptr; }
14
15 LongIdentifierPattern *Pattern::asLongIdentifierPattern() { return
    nullptr; }
16
17 ApplyPattern *Pattern::asApplyPattern() { return nullptr; }
18
19 SeqPattern::SeqPattern(SourceLocation L) : Pattern(L) {}
20
21 SeqPattern::SeqPattern(SourceLocation L, std::shared_ptr<Pattern>
    First)
22     : Pattern(L), std::vector<std::shared_ptr<Pattern>>{First} {}
23
24 void SeqPattern::accept(Visitor *V) { V->visit(this); }
25
26 LiteralPattern::LiteralPattern(SourceLocation L,
    std::shared_ptr<Literal> Lit)
27     : Pattern(L), Value(Lit) {}
28
29 void LiteralPattern::accept(Visitor *V) { V->visit(this); }
30
31 WildcardPattern::WildcardPattern(SourceLocation L) : Pattern(L) {}
32
33 void WildcardPattern::accept(Visitor *V) { V->visit(this); }
34
35 LongIdentifierPattern::LongIdentifierPattern(SourceLocation L,
    std::shared_ptr<LongIdentifier>
36         ID,
37         bool opPrefixed)
38     : Pattern(L), ID(ID), opKeyPrefixed(opPrefixed) {}
39
40 void LongIdentifierPattern::accept(Visitor *V) { V->visit(this); }
41
42 LongIdentifierPattern
43     *LongIdentifierPattern::asLongIdentifierPattern() {
44     return this;
45 }
46
47 TypePattern::TypePattern(SourceLocation L,
    std::shared_ptr<Pattern> P,
48     std::shared_ptr<Type> T)
49     : Pattern(L), Pat(P), Typ(T) {}
50
51 TypePattern *TypePattern::asTypePattern() { return this; }
52
53 void TypePattern::accept(Visitor *V) { V->visit(this); }
54
55 AsPattern::AsPattern(SourceLocation L, std::shared_ptr<Pattern>
    LHS,
56     std::shared_ptr<Pattern> RHS)
57     : Pattern(L), LeftPattern(LHS), RightPattern(RHS) {}
58
59 void AsPattern::accept(Visitor *V) { V->visit(this); }
60
61 ApplyPattern::ApplyPattern(SourceLocation L,
    std::shared_ptr<Pattern> First)
62     : Pattern(L), std::vector<std::shared_ptr<Pattern>>{First} {}

```

```

62
63 void ApplyPattern::accept(Visitor *V) { V->visit(this); }
64
65 ApplyPattern *ApplyPattern::asApplyPattern() { return this; }
66
67 IdentifierEqualsPattern::IdentifierEqualsPattern(
68     SourceLocation L, std::shared_ptr<Identifier> LHS,
69     std::shared_ptr<Pattern> RHS)
70     : Pattern(L), Iden(LHS), Pat(RHS) {}
71
72 void IdentifierEqualsPattern::accept(Visitor *V) { V->visit(this);
73     }
74
75 ShortIdentifierPattern::ShortIdentifierPattern(
76     SourceLocation L, std::shared_ptr<ShortIdentifier> ID)
77     : Pattern(L), ID(ID) {}
78
79 void ShortIdentifierPattern::accept(Visitor *V) { V->visit(this); }
80
81 RecordPattern::RecordPattern(SourceLocation L,
82     std::shared_ptr<Pattern> First)
83     : Pattern(L), std::vector<std::shared_ptr<Pattern>>{First} {}
84
85 void RecordPattern::accept(Visitor *V) { V->visit(this); }
86
87 ListPattern::ListPattern(SourceLocation L,
88     std::shared_ptr<SeqPattern> P)
89     : Pattern(L), Patterns(P) {}
90
91 void ListPattern::accept(Visitor *V) { V->visit(this); }
92
93 TuplePattern::TuplePattern(SourceLocation L,
94     std::shared_ptr<SeqPattern> P)
95     : Pattern(L), Patterns(P) {
96     assert(P->size() != 1);
97 }
98
99 void TuplePattern::accept(Visitor *V) { V->visit(this); }

```

Listing 73: lib/GCPrinter/SSMLGC.cpp

```

1 #include "llvm/CodeGen/GCs.h"
2 #include "llvm/CodeGen/GCStrategy.h"
3 #include "llvm/CodeGen/MachineInstrBuilder.h"
4 #include "llvm/MC/MCContext.h"
5 #include "llvm/MC/MCSymbol.h"
6 #include "llvm/Target/TargetInstrInfo.h"
7 #include "llvm/Target/TargetMachine.h"
8 #include "llvm/Target/TargetSubtargetInfo.h"
9
10 using namespace llvm;
11
12 namespace {
13 class SSMLGC : public GCStrategy {
14 public:
15     SSMLGC();
16 };
17 }
18
19 static GCRegistry::Add<SSMLGC> X("ssml", "ssml garbage collector
    strategy");

```

```

20
21 SSMLGC::SSMLGC() {
22     this->UsesMetadata = true;
23     this->InitRoots = true;
24     this->NeededSafePoints = 1 << GC::PostCall;
25 }

```

Listing 74: lib/GCPrinter/SSMLGCPrinter.cpp

```

1  #include "llvm/CodeGen/AsmPrinter.h"
2  #include "llvm/CodeGen/GCMetadataPrinter.h"
3  #include "llvm/CodeGen/GCs.h"
4  #include "llvm/IR/DataLayout.h"
5  #include "llvm/IR/Function.h"
6  #include "llvm/IR/Instruction.h"
7  #include "llvm/IR/IntrinsicInst.h"
8  #include "llvm/IR/Metadata.h"
9  #include "llvm/MC/MCAsmInfo.h"
10 #include "llvm/MC/MCContext.h"
11 #include "llvm/MC/MCSectionELF.h"
12 #include "llvm/MC/MCStreamer.h"
13 #include "llvm/MC/MCSymbol.h"
14 #include "llvm/Target/TargetLoweringObjectFile.h"
15 #include "llvm/Target/TargetMachine.h"
16 #include "llvm/Target/TargetSubtargetInfo.h"
17
18 #include <stdint.h>
19
20 using namespace llvm;
21
22 namespace {
23
24 class SSMLGCPrinter : public GCMetadataPrinter {
25 public:
26     void finishAssembly(Module &M, GCModuleInfo &Info, AsmPrinter
27         &AP) override;
28 };
29
30 static GCMetadataPrinterRegistry::Add<SSMLGCPrinter>
31     X("ssml", "ssml garbage collector strategy");
32
33 void SSMLGCPrinter::finishAssembly(Module &M, GCModuleInfo &Info,
34     AsmPrinter &AP) {
35     MCStreamer &OS = *AP.OutStreamer;
36     unsigned IntPtrSize = M.getDataLayout().getPointerSize();
37
38     // Add end of code label.
39     AP.OutStreamer->SwitchSection(AP.getObjFileLowering().getTextSection());
40     MCSymbol *CodeEndLabel =
41         AP.OutContext.getOrCreateSymbol("code_end");
42     AP.OutStreamer->EmitLabel(CodeEndLabel);
43
44     // Put GC Map in a custom section.
45     OS.SwitchSection(AP.getObjFileLowering().getContext().getELFSection(
46         "ssmlgcmap", ELF::SHT_PROGBITS, ELF::SHF_ALLOC));
47
48     #if 0
49     MCSymbol *GCLabel =
50         AP.OutContext.getOrCreateSymbol("ssml_gcmap");
51     AP.OutStreamer->EmitSymbolAttribute(GCLabel, MCSA_Global);
52     #endif
53 }

```

```

50 AP.OutStreamer->EmitLabel(GCLabel);
51 #endif
52
53 std::vector<MCSymbol *> TableSyms;
54 std::string SymPrefix("gcmmap_sym_");
55 size_t SymIdx = 0;
56
57 // For each function...
58 for (GCModuleInfo::FuncInfoVec::iterator FI =
59     Info.funcinfo_begin(),
60     IE = Info.funcinfo_end();
61     FI != IE; ++FI) {
62     GCFunctionInfo &MD = **FI;
63     if (MD.getStrategy().getName() != getStrategy().getName())
64         continue; // this function is managed by some other GC
65     /** A compact GC layout. Emit this data structure:
66     *
67     * struct {
68     *     int16_t StackFrameSize; (in words)
69     *     int16_t LiveCount;
70     *     int16_t LiveOffsets[LiveCount];
71     * } gcmmap_sym_${SymIdx};
72     */
73
74     AP.EmitAlignment(3);
75
76     MCSymbol *GCMMapSym =
77         AP.OutContext.getOrCreateSymbol(SymPrefix +
78             std::to_string(SymIdx));
79     ++SymIdx;
80     AP.OutStreamer->EmitLabel(GCMMapSym);
81     TableSyms.push_back(GCMMapSym);
82
83     // Stack information never change in safe points! Only print
84     // info from the
85     // first call-site.
86     GCFunctionInfo::iterator PI = MD.begin();
87
88     // Emit the stack frame size.
89     OS.AddComment("stack frame size (in words)");
90     AP.EmitInt16(MD.getFrameSize() / IntPtrSize);
91
92     // Emit the number of live roots in the function.
93     OS.AddComment("live root count");
94     AP.EmitInt16(MD.live_size(PI));
95
96     // And for each live root...
97     for (GCFunctionInfo::live_iterator LI = MD.live_begin(PI),
98         LE = MD.live_end(PI);
99         LI != LE; ++LI) {
100         // Emit live root's offset within the stack frame.
101         OS.AddComment("stack index (offset / wordsize)");
102         AP.EmitInt16(LI->StackOffset / IntPtrSize);
103     }
104
105     // Put lookup table in custom section.
106     OS.SwitchSection(AP.getObjFileLowering().getContext().getELFSection(
107         "ssm1gcllookup", ELF::SHT_PROGBITS, ELF::SHF_ALLOC));

```

```

108 AP.EmitAlignment(3);
109 MCSymbol *SizeLabel =
    AP.OutContext.getOrCreateSymbol("ssml_gclookup_size");
110 AP.OutStreamer->EmitSymbolAttribute(SizeLabel, MCSA_Global);
111 AP.OutStreamer->EmitLabel(SizeLabel);
112
113 // Emit number of functions.
114 OS.AddComment("function count");
115 assert(TableSyms.size() <= INT16_MAX);
116 AP.EmitInt16(TableSyms.size());
117
118 AP.EmitAlignment(3);
119 MCSymbol *MapLabel =
    AP.OutContext.getOrCreateSymbol("ssml_gclookup_map");
120 AP.OutStreamer->EmitSymbolAttribute(MapLabel, MCSA_Global);
121 AP.OutStreamer->EmitLabel(MapLabel);
122
123 size_t Idx = 0;
124 for (GCModuleInfo::FuncInfoVec::iterator FI =
    Info.funcinfo_begin(),
125                                     IE =
    Info.funcinfo_end();
126      FI != IE; ++FI) {
127     GCFunctionInfo &MD = **FI;
128     MCSymbol *NextFun = AP.getSymbol(MD.getFunction());
129     AP.EmitLabelReference(NextFun, 8);
130     AP.EmitLabelReference(TableSyms[Idx], 8);
131     ++Idx;
132 }
133 AP.EmitLabelReference(CodeEndLabel, 8);
134 }

```

Listing 75: lib/Typecheck/Typedef.h

```

1 #ifndef LLVM_TOOLS_SSML_LIB_TYPECHECK_TYPEDEF_H
2 #define LLVM_TOOLS_SSML_LIB_TYPECHECK_TYPEDEF_H
3
4 #include "ssml/Common/SourceLocation.h"
5
6 #include "Typemap.h"
7
8 #include <string>
9 #include <memory>
10 #include <vector>
11
12 namespace ssml {
13 namespace typecheck {
14 class Typedef {
15 public:
16     enum Kind {
17         SimpleBuildin,
18         Product,
19         Function,
20         Tuple,
21         Datatype,
22         Variable,
23         DatatypeInstance
24     };
25
26     using VartypeVector =
27         std::vector<std::pair<std::string,

```

```

28         std::shared_ptr<Typedef>>>;
29 public:
30     virtual ~Typedef();
31     virtual Typedef *clone() const = 0;
32     virtual Kind getKind() const = 0;
33     virtual void setArgument(SourceLocation,
34         std::shared_ptr<Typedef>) = 0;
35     virtual void unmarkInScope() = 0;
36     virtual std::shared_ptr<Typedef>
37         funApplication(SourceLocation, std::shared_ptr<Typedef>)
38             const = 0;
39     virtual VartypeVector getVariableTypes(SourceLocation,
40         std::shared_ptr<Typedef>)
41         const = 0;
42     virtual bool canBeAssignWith(const Typedef &, const
43         VartypeVector *,
44         bool ScopedVarIsVar) const;
45     virtual bool cannotBeAssignWith(const Typedef &, const
46         VartypeVector *,
47         bool ScopedVarIsVar) const;
48     virtual bool isTypeComplete() const = 0;
49     virtual bool isTypeMissingArgument() const;
50     virtual bool isType() const;
51     virtual bool operator==(const Typedef &) const;
52     virtual bool operator!=(const Typedef &) const;
53     virtual bool hasVariable() const = 0;
54     virtual std::shared_ptr<Typedef>
55         withoutVariableTypedefs(const VartypeVector &) = 0;
56 };
57 } // End namespace typecheck.
58 } // End namespace ssml.
59
60 namespace ssml {
61     namespace typecheck {
62     class SimpleBuildinTypedef : public Typedef {
63     private:
64         std::string Typename;
65     public:
66         SimpleBuildinTypedef(std::string Name);
67         SimpleBuildinTypedef *clone() const override;
68         Typedef::Kind getKind() const final;
69         void setArgument(SourceLocation, std::shared_ptr<Typedef>)
70             override;
71         void unmarkInScope() override;
72         Typedef::VartypeVector getVariableTypes(SourceLocation,
73             std::shared_ptr<Typedef>)
74             const;
75         bool canBeAssignWith(const Typedef &, const
76             Typedef::VartypeVector *,
77             bool ScopedVarIsVar) const override;
78         bool operator==(const Typedef &) const override;
79         bool isTypeComplete() const override;
80         bool hasVariable() const override;
81         std::shared_ptr<Typedef>
82             withoutVariableTypedefs(const VartypeVector &) override;
83     };
84 } // End namespace typecheck.
85 } // End namespace ssml.
86
87 namespace ssml {

```

```

81 namespace typecheck {
82 class ProductTypedef : public Typedef {
83 protected:
84     std::vector<std::shared_ptr<Typedef>> Typedefs;
85
86 public:
87     ProductTypedef *clone() const override;
88     Typedef::Kind getKind() const override;
89     void setArgument(SourceLocation, std::shared_ptr<Typedef>)
90         override;
91     void unmarkInScope() override;
92     Typedef::VartypeVector getVariableTypes(SourceLocation,
93                                             std::shared_ptr<Typedef>)
94         const;
95
96     bool canBeAssignWith(const Typedef &, const
97                         Typedef::VartypeVector *,
98                         bool ScopedVarIsVar) const override;
99
100     bool operator==(const Typedef &) const override;
101     virtual void append(std::shared_ptr<Typedef>);
102     bool isTypeComplete() const override;
103     bool hasVariable() const override;
104
105     const std::vector<std::shared_ptr<Typedef>> &getTypedefs() const
106     {
107         return this->Typedefs;
108     }
109     std::shared_ptr<Typedef>
110     withoutVariableTypedefs(const VartypeVector &) override;
111 };
112 // End namespace typecheck.
113 // End namespace ssml.
114
115 namespace ssml {
116 namespace typecheck {
117 class FunctionTypedef : public ProductTypedef {
118 private:
119     mutable Typedef::VartypeVector ApplicationVarTypes;
120     bool HasCreatedAppendFunc = false;
121
122 public:
123     FunctionTypedef *clone() const override;
124     Typedef::Kind getKind() const final;
125     bool canBeAssignWith(const Typedef &, const
126                         Typedef::VartypeVector *,
127                         bool ScopedVarIsVar) const override;
128     std::shared_ptr<Typedef>
129     withoutVariableTypedefs(const VartypeVector &) override;
130     std::shared_ptr<Typedef>
131     funApplication(SourceLocation, std::shared_ptr<Typedef>)
132         const override;
133     void append(std::shared_ptr<Typedef>) override;
134 };
135 // End namespace typecheck.
136 // End namespace ssml.
137
138 namespace ssml {
139 namespace typecheck {
140 class TupleTypedef : public Typedef {
141 private:
142     std::vector<std::shared_ptr<Typedef>> Typedefs;
143
144 public:

```

```

137 TupleTypedef *clone() const override;
138 Typedef::Kind getKind() const final;
139 void append(std::shared_ptr<Typedef>);
140 void setArgument(SourceLocation, std::shared_ptr<Typedef>)
    override;
141 void unmarkInScope() override;
142 Typedef::VartypeVector getVariableTypes(SourceLocation,
143                                         std::shared_ptr<Typedef>)
                                         const;
144 bool hasVariable() const override;
145 bool canBeAssignWith(const Typedef &, const
    Typedef::VartypeVector *,
146                      bool ScopedVarIsVar) const override;
147 bool operator==(const Typedef &) const override;
148 bool isTypeComplete() const override;
149 bool isType() const override;
150
151 const std::vector<std::shared_ptr<Typedef>> getTypedefs() const {
152     return this->Typedefs;
153 }
154 std::shared_ptr<Typedef>
155 withoutVariableTypedefs(const VartypeVector &) override;
156 };
157 } // End namespace typecheck.
158 } // End namespace ssml.
159
160 namespace ssml {
161 namespace typecheck {
162 class DatatypeTypedef : public Typedef {
163 public:
164     std::string Typename;
165     std::vector<std::pair<std::string, std::shared_ptr<Typedef>>>
        Params;
166     bool IsArgumentSet = false;
167
168 public:
169     DatatypeTypedef(std::string Name, std::vector<std::string>
        ParamNames);
170     DatatypeTypedef(
171         std::string Name,
172         const std::vector<std::pair<std::string,
173             std::shared_ptr<Typedef>>>
            &ParamNames);
174     DatatypeTypedef *clone() const override;
175     Typedef::Kind getKind() const final;
176     void setArgument(SourceLocation, std::shared_ptr<Typedef>)
        override;
177     void unmarkInScope() override;
178     void setVariableType(const std::string &Name,
        std::shared_ptr<Typedef> T);
179     Typedef::VartypeVector getVariableTypes(SourceLocation,
180                                             std::shared_ptr<Typedef>)
                                                const;
181     bool canBeAssignWith(const Typedef &, const
        Typedef::VartypeVector *,
182                          bool ScopedVarIsVar) const override;
183     bool hasVariable() const override;
184     bool operator==(const Typedef &) const override;
185     bool isTypeComplete() const override;
186     bool isTypeMissingArgument() const override;
187     std::shared_ptr<Typedef>
188     withoutVariableTypedefs(const VartypeVector &) override;

```

```

189
190     const std::vector<std::pair<std::string,
191         std::shared_ptr<Typedef>>> &
192     getParams() const {
193         return this->Params;
194     }
195
196     const std::string &getUserTypename() const { return
197         this->Typename; }
198 };
199 // End namespace typecheck.
200 // End namespace ssml.
201
202 namespace ssml {
203     namespace typecheck {
204         extern std::shared_ptr<DatatypeTypedef> getListTypedef();
205         extern std::shared_ptr<DatatypeTypedef> getRefTypedef();
206         extern std::shared_ptr<DatatypeTypedef> getArrayTypedef();
207     } // End namespace typecheck.
208 } // End namespace ssml.
209
210 namespace ssml {
211     namespace typecheck {
212         class VariableTypedef : public Typedef {
213         private:
214             std::string UserTypename;
215             std::string InternTypename;
216             bool IsInScope = false;
217
218         public:
219             VariableTypedef(std::string UserName, std::string InternName);
220             VariableTypedef *clone() const override;
221             Typedef::Kind getKind() const final;
222             void setArgument(SourceLocation, std::shared_ptr<Typedef>)
223                 override;
224             void unmarkInScope() override;
225             Typedef::VartypeVector getVariableTypes(SourceLocation,
226                 std::shared_ptr<Typedef>)
227                 const;
228
229             bool canBeAssignWith(const Typedef &, const
230                 Typedef::VartypeVector *,
231                 bool ScopedVarIsVar) const override;
232             bool hasVariable() const override;
233             bool operator==(const Typedef &) const override;
234             bool isTypeComplete() const override;
235             void setIsInScope(bool B) { this->IsInScope = B; }
236             bool isInScope() const { return this->IsInScope; }
237             std::shared_ptr<Typedef>
238             withoutVariableTypedefs(const VartypeVector &) override;
239
240             const std::string &getUserTypename() const { return
241                 this->UserTypename; }
242             const std::string &getInternTypename() const { return
243                 this->InternTypename; }
244         };
245     } // End namespace typecheck.
246 } // End namespace ssml.
247
248 namespace ssml {
249     namespace typecheck {
250         class DatatypeInstanceTypedef : public Typedef {
251         private:

```

```

244 DatatypeTypedef DatatypeTypedef;
245 std::shared_ptr<Typedef> ArgumentTypedef;
246 std::shared_ptr<Typedef> Argument;
247
248 public:
249     DatatypeInstanceTypedef(const DatatypeTypedef &Datatype,
250                             std::shared_ptr<Typedef> Argument =
251                                 nullptr);
252     DatatypeInstanceTypedef *clone() const override;
253     Typedef::Kind getKind() const final;
254     void setArgument(SourceLocation, std::shared_ptr<Typedef>)
255         override;
256     void unmarkInScope() override;
257     std::shared_ptr<Typedef>
258         funApplication(SourceLocation, std::shared_ptr<Typedef>)
259             const override;
260     Typedef::VartypeVector getVariableTypes(SourceLocation,
261                                             std::shared_ptr<Typedef>)
262         const;
263     bool canBeAssignWith(const Typedef &, const
264                         Typedef::VartypeVector *,
265                         bool ScopedVarIsVar) const override;
266     bool operator==(const Typedef &) const override;
267     bool isTypeComplete() const override;
268     bool isTypeMissingArgument() const override;
269     bool hasVariable() const override;
270     std::shared_ptr<Typedef>
271     withoutVariableTypedefs(const VartypeVector &) override;
272
273     const DatatypeTypedef &getDatatypeTypedef() const { return
274         this->DatatypeTypedef; }
275     std::shared_ptr<Typedef> getArgumentTypedef() {
276         return this->ArgumentTypedef;
277     }
278 };
279 } // End namespace typecheck.
280 } // End namespace ssml.
281
282 #endif // LLVM_TOOLS_SSML_LIB_TYPECHECK_TYPEDEF_H

```

Listing 76: lib/Typecheck/Typecheck.cpp

```

1 #include "ssml/Typecheck/Typecheck.h"
2
3 #include "TypecheckVisitor.h"
4
5 #include "ssml/AST/Declaration.h"
6
7 using namespace ssml::typecheck;
8
9 void ssml::typecheck::typecheck(ssml::ast::Root *R) {
10     TypecheckVisitor V;
11     R->accept(&V);
12 }

```

Listing 77: lib/Typecheck/Typemap.h

```

1 #ifndef LLVM_TOOLS_SSML_LIB_TYPECHECK_TYPEMAP_H
2 #define LLVM_TOOLS_SSML_LIB_TYPECHECK_TYPEMAP_H
3
4 #include "ssml/Common/Compare.h"

```

```

5
6 #include <string>
7 #include <memory>
8 #include <map>
9 #include <vector>
10
11 namespace ssml {
12 namespace typecheck {
13 class Typedef;
14 } // End namespace typecheck.
15 } // End namespace ssml.
16
17 namespace ssml {
18 namespace typecheck {
19 class Typemap {
20 public:
21     using ValueType = std::shared_ptr<Typedef>;
22     using MapType = std::map<std::string, ValueType>;
23
24 private:
25     std::vector<MapType> Maps;
26
27 public:
28     void insert(std::string K, ValueType V, size_t Scope);
29     ValueType get(const std::string &K) const;
30     void mergeWith(MapType, size_t Scope);
31     void enterScope();
32     void leaveScope();
33     void erase(const std::string &S, size_t Scope) {
34         Maps[Maps.size() - 1 - Scope].erase(S);
35     }
36     void erase(const MapType &M, size_t Scope) {
37         for (auto &&P : M)
38             this->erase(P.first, Scope);
39     }
40 };
41 } // End namespace typecheck.
42 } // End namespace ssml.
43
44 #endif // LLVM_TOOLS_SSML_LIB_TYPECHECK_TYPEMAP_H

```

Listing 78: lib/Typecheck/TypecheckVisitor.cpp

```

1 #define DEBUG_TYPE "Typecheck"
2
3 #include "TypecheckVisitor.h"
4 #include "Patternmatch.h"
5
6 #include "ssml/AST/All.h"
7 #include "ssml/Common/ErrorMessage.h"
8 #include "ssml/Common/FatalExit.h"
9
10 #include "llvm/Support/raw_ostream.h"
11 #include "llvm/Support/Debug.h"
12
13 #define TYPECHECK_DLOG(msg)
14
15     DEBUG(llvm::errs() << DEBUG_TYPE " :: " << msg << '\n')
16
17 using namespace ssml;
18 using namespace ssml::ast;

```

```

18 using namespace ssml::typecheck;
19
20 std::shared_ptr<Typedef> TypecheckVisitor::getPrevPatternType() {
21     assert(!this->PrevPatternmatch);
22     auto T = this->PrevPatternmatch->getFreeMatchPatternType();
23     if (T && T->isTypeComplete())
24         return T;
25     return nullptr;
26 }
27
28 bool TypecheckVisitor::cannotBeAssignWith(std::shared_ptr<Typedef>
29     DestType,
30     SourceLocation SourceL,
31     std::shared_ptr<Typedef>
32     SourceType,
33     bool
34     ExprHasValApplication,
35     bool ScopedVarIsVar,
36     bool IsRet) {
37     bool B;
38     if (SourceType->getKind() == Typedef::Function) {
39         auto VarTs = SourceType->getVariableTypes(SourceL, DestType);
40         B = DestType->cannotBeAssignWith(*SourceType, &VarTs,
41             ScopedVarIsVar);
42     } else {
43         B = DestType->cannotBeAssignWith(*SourceType, nullptr,
44             ScopedVarIsVar);
45     }
46
47     if (ExprHasValApplication && !IsRet)
48         if (DestType->hasVariable())
49             errorExit(SourceL, "expression involving function
50                 application cannot "
51                 "be of variable type");
52
53     return B;
54 }
55
56 void TypecheckVisitor::enterScope() {
57     static unsigned ScopeCount = 0;
58     if (ScopeCount)
59         this->TypenamePrefix =
60             "?" + std::to_string(ScopeCount) + "." +
61             this->TypenamePrefix;
62     ++ScopeCount;
63     this->TypenameTypes.enterScope();
64     this->ValnameTypes.enterScope();
65     this->ConstnameTypes.enterScope();
66 }
67
68 void TypecheckVisitor::leaveScope() {
69     auto Idx = this->TypenamePrefix.find_first_of(".");
70     if (Idx != std::string::npos)
71         this->TypenamePrefix = this->TypenamePrefix.substr(Idx + 1);
72     this->TypenameTypes.leaveScope();
73     this->ValnameTypes.leaveScope();
74     this->ConstnameTypes.leaveScope();
75 }
76
77 void TypecheckVisitor::doVisit(std::shared_ptr<ssml::ast::Node> N)
78 {
79     this->doVisit(N.get());
80 }

```

```

72 }
73
74 void TypecheckVisitor::doVisit(ssml::ast::Node *N) {
75     this->PrevType.reset();
76     this->PrevPatternmatch.reset();
77     N->accept(this);
78 }
79
80 void TypecheckVisitor::visit(IntLiteral *N) {
81     this->PrevType = this->IntTypedef;
82 }
83
84 void TypecheckVisitor::visit(ShortIdentifier *N) {
85     fatalExit("TypecheckVisitor visit ShortIdentifier");
86 }
87
88 void TypecheckVisitor::visit(LongIdentifier *N) {
89     fatalExit("TypecheckVisitor visit LongIdentifier");
90 }
91
92 void TypecheckVisitor::visit(SeqShortIdentifier *N) {
93     fatalExit("TypecheckVisitor visit SeqShortIdentifier");
94 }
95
96 void TypecheckVisitor::visit(SeqLongIdentifier *N) {
97     fatalExit("TypecheckVisitor visit SeqLongIdentifier");
98 }
99
100 void TypecheckVisitor::visit(SeqExpression *N) {
101     for (auto C : *N)
102         this->doVisit(C);
103     // this->PrevType is set to the last expression in the list:
104     // let ... in ... end
105 }
106
107 void TypecheckVisitor::visit(LiteralExpression *N) {
108     this->doVisit(N->getValue());
109     // this->PrevType is set.
110     this->ApplyExprWasVal = true;
111 }
112
113 void TypecheckVisitor::visit(LongIdentifierExpression *N) {
114     this->PrevType = this->ValnameTypes.get(N->getIDs()->toString());
115     if (!this->PrevType)
116         this->PrevType =
117             this->ConstnameTypes.get(N->getIDs()->toString());
118     else
119         this->ApplyExprWasVal = true;
120     if (!this->PrevType)
121         errorExit(N->getIDs()->getLocation(), "undeclared identifier");
122 }
123
124 void TypecheckVisitor::visit(TupleExpression *N) {
125     auto T = std::make_shared<ProductTypedef>();
126     for (auto C : *N->getExprs()) {
127         this->doVisit(C);
128         T->append(this->PrevType);
129     }
130     this->PrevType = T;
131     this->ApplyExprWasVal = true;
132 }

```

```

133 void TypecheckVisitor::visit(ListExpression *N) {
134     auto List = getListTypedef();
135     std::shared_ptr<Typedef> ListType;
136     for (auto E : *N->getExprs()) {
137         this->doVisit(E);
138         if (ListType) {
139             if (this->cannotBeAssignWith(ListType, N->getLocation(),
140                 this->PrevType,
141                     false, false) &&
142                 this->cannotBeAssignWith(this->PrevType,
143                     N->getLocation(), ListType,
144                         false, false))
145                     errorExit(E->getLocation(), "contradicting expression
146                         types in list");
147         }
148         if (!ListType || this->PrevType->isTypeComplete())
149             ListType = this->PrevType;
150     }
151     if (ListType)
152         List->setArgument(N->getLocation(), ListType);
153     this->PrevType = List;
154     this->ApplyExprWasVal = true;
155 }
156
157 void TypecheckVisitor::visit(ApplyExpression *N) {
158     size_t Count = 0;
159     std::shared_ptr<Typedef> Fun;
160     bool FirstWasValname = false;
161     for (auto C : *N) {
162         this->ApplyExprWasVal = false;
163         this->doVisit(C);
164         if (!this->PrevType->isType())
165             fatalExit("Expression prev type is not a real type");
166
167         if (Count) {
168             if (!this->ApplyExprWasVal &&
169                 this->PrevType->isTypeMissingArgument())
170                 errorExit(C->getLocation(), "missing argument");
171             Fun = Fun->funApplication(C->getLocation(), this->PrevType);
172         } else {
173             Fun = this->PrevType;
174             FirstWasValname = this->ApplyExprWasVal;
175         }
176         ++Count;
177     }
178     if (Fun->getKind() == Typedef::DatatypeInstance &&
179         Fun->isTypeMissingArgument())
180         errorExit(N->getLocation(), "missing argument");
181
182     this->PrevType = Fun;
183     if (FirstWasValname && Count > 1)
184         this->ExprHasValnameApply = true;
185 }
186
187 void TypecheckVisitor::visit(OrElseExpression *N) {
188     this->doVisit(N->getLeftExpr());
189     if (*this->PrevType != *this->BoolTypedef)
190         errorExit(N->getLeftExpr()->getLocation(), "unexpected
191             expression type");
192     this->doVisit(N->getRightExpr());
193     if (*this->PrevType != *this->BoolTypedef)
194         errorExit(N->getLeftExpr()->getLocation(), "unexpected

```

```

190         expression type");
191     this->PrevType.reset(this->BoolTypedef->clone());
192     this->ApplyExprWasVal = true;
193 }
194 void TypecheckVisitor::visit(AndAlsoExpression *N) {
195     this->doVisit(N->getLeftExpr());
196     if (*this->PrevType != *this->BoolTypedef)
197         errorExit(N->getLeftExpr()->getLocation(), "unexpected
198             expression type");
199     this->doVisit(N->getRightExpr());
200     if (*this->PrevType != *this->BoolTypedef)
201         errorExit(N->getLeftExpr()->getLocation(), "unexpected
202             expression type");
203     this->PrevType.reset(this->BoolTypedef->clone());
204     this->ApplyExprWasVal = true;
205 }
206 void TypecheckVisitor::visit(LetExpression *N) {
207     this->enterScope();
208     this->doVisit(N->getDecls());
209     this->doVisit(N->getExprs());
210     // this->PrevType is set.
211     this->leaveScope();
212     // this->ApplyExprWasVal is set.
213 }
214 void TypecheckVisitor::visit(IfExpression *N) {
215     this->doVisit(N->getCondExpr());
216     if (*this->PrevType != *this->BoolTypedef)
217         errorExit(N->getCondExpr()->getLocation(), "unexpected
218             expression type");
219     this->doVisit(N->getThenExpr());
220     auto Ret = this->PrevType;
221     this->doVisit(N->getElseExpr());
222     if (this->cannotBeAssignWith(this->PrevType,
223         N->getThenExpr()->getLocation(),
224         Ret, false, false) &&
225         this->cannotBeAssignWith(Ret,
226         N->getElseExpr()->getLocation(),
227         this->PrevType, false, false))
228         errorExit(N->getLocation(), "conflicting if expression types");
229     this->PrevType = Ret;
230     this->ApplyExprWasVal = true;
231 }
232 void TypecheckVisitor::visit(WhileExpression *N) {
233     this->doVisit(N->getCondExpr());
234     if (*this->PrevType != *this->BoolTypedef)
235         errorExit(N->getCondExpr()->getLocation(), "unexpected
236             expression type");
237     this->doVisit(N->getBodyExpr());
238     this->PrevType.reset(this->UnitTypedef->clone());
239     this->ApplyExprWasVal = true;
240 }
241 void TypecheckVisitor::visit(Match *N) {
242     bool PrevAdd = this->AddVariableTypeToTypenames;
243     this->AddVariableTypeToTypenames = true;
244     this->doVisit(N->getPattern());

```

```

245     auto Pat = this->PrevPatternmatch;
246     auto NameTypes = Pat->getNameTypes();
247
248     for (auto &S : this->PatternIdenInsertionOrder) {
249         assert(NameTypes.count(S));
250         auto Def = NameTypes[S];
251         if (Def->getKind() == Typedef::SimpleBuildin)
252             this->peekFuncVals()->addVal(false);
253         else
254             this->peekFuncVals()->addVal(true);
255     }
256     this->PatternIdenInsertionOrder.clear();
257
258     this->enterScope();
259     this->ValnameTypes.mergeWith(NameTypes, 0);
260     this->ConstnameTypes.erase(NameTypes, 0);
261
262     this->doVisit(N->getExpr());
263     // this->PrevType is set.
264     this->PrevPatternmatch = Pat;
265
266     this->AddVariableTypeToTypenames = PrevAdd;
267     this->leaveScope();
268 }
269
270 void TypecheckVisitor::visit(SeqMatch *N) {
271     fatalExit("TypecheckVisitor visit SeqMatch");
272 }
273
274 void TypecheckVisitor::visit(LambdaExpression *N) {
275     bool PrevHasApply = this->ExprHasValnameApply;
276
277     this->pushFuncVals();
278     N->setFunctionVals(this->peekFuncVals());
279     this->enterScope();
280
281     std::shared_ptr<Typedef> PatternType;
282     std::shared_ptr<Typedef> CaseType;
283     SourceLocation CaseTypeLocation;
284     SourceLocation PatternTypeLocation;
285     auto Matches = N->getCases();
286     for (auto &&M : *Matches) {
287         this->ExprHasValnameApply = false;
288         this->doVisit(M);
289         auto T = this->PrevType;
290         auto PatType = this->getPrevPatternType();
291         if (!PatType)
292             errorExit(M->getLocation(), "pattern missing type
specification");
293         if (CaseType &&
294             this->cannotBeAssignWith(T, CaseTypeLocation, CaseType,
false, true) &&
295             this->cannotBeAssignWith(CaseType, M->getLocation(),
this->PrevType,
false, true))
296             errorExit(N->getLocation(), "conflicting fn case types");
297         bool HasApply = this->ExprHasValnameApply;
298         if (PatternType &&
299             this->cannotBeAssignWith(PatType, PatternTypeLocation,
PatternType,
300                                     HasApply, false) &&
301             this->cannotBeAssignWith(PatternType, M->getLocation(),

```

```

303         PatType,
304         HasApply, false))
305     errorExit(M->getLocation(), "conflicting fn pattern types");
306     if (!PatternType) {
307         PatternType = PatType;
308         PatternTypeLocation = M->getLocation();
309     }
310     if (!CaseType || (!CaseType->isTypeComplete() &&
311         T->isTypeComplete())) {
312         CaseType = T;
313         CaseTypeLocation = M->getLocation();
314     }
315     }
316     auto Fun = std::make_shared<FunctionTypedef>();
317     Fun->append(PatternType);
318     Fun->append(CaseType);
319     this->PrevType = Fun;
320     this->ApplyExprWasVal = true;
321     this->ExprHasValnameApply = PrevHasApply;
322     this->leaveScope();
323     this->popFuncVals();
324 }
325
326 void TypecheckVisitor::visit(SeqPattern *N) {
327     for (auto C : *N)
328         this->doVisit(C);
329     if (N->size() > 1)
330         this->PrevType.reset();
331 }
332
333 void TypecheckVisitor::visit(LiteralPattern *N) {
334     this->doVisit(N->getValue());
335     auto T = this->PrevType;
336     this->PrevPatternmatch.reset(
337         new LiteralPatternmatch(N->getValue()->getLocation(), T));
338     this->PrevType.reset();
339 }
340
341 void TypecheckVisitor::visit(WildcardPattern *N) {
342     this->PrevPatternmatch.reset(new
343         WildcardPatternmatch(N->getLocation()));
344     this->PrevType.reset();
345 }
346
347 void TypecheckVisitor::visit(LongIdentifierPattern *N) {
348     if (!this->ConstnameTypes.get(N->getIDs()->toString()))
349         this->PatternIdenInsertionOrder.push_back(N->getIDs()->toString());
350     this->PrevPatternmatch.reset(new LongIdentifierPatternmatch(
351         N->getLocation(), this->ConstnameTypes,
352         N->getIDs()->toString()));
353     this->PrevType.reset();
354 }
355
356 void TypecheckVisitor::visit(TypePattern *N) {
357     this->doVisit(N->getPattern());
358     auto Match = this->PrevPatternmatch;
359     this->doVisit(N->getType());
360     if (!this->PrevType->isType())
361         errorExit(N->getLocation(), "expected type");
362     Match->matchType(this->PrevType);

```

```

361     this->PrevPatternmatch = Match;
362     // this->PrevType is set correctly at this point.
363     // this->PrevType can be used to verify that this node has been
        visited.
364 }
365
366 void TypecheckVisitor::visit(ApplyPattern *N) {
367     int Count = 0;
368     std::shared_ptr<Patternmatch> First;
369     for (auto C : *N) {
370         this->doVisit(C);
371         switch (Count) {
372             case 0:
373                 First = this->PrevPatternmatch;
374                 break;
375             case 1:
376                 this->PrevPatternmatch.reset(new ApplyPatternmatch(
377                     N->getLocation(), C->getLocation(), First,
378                     this->PrevPatternmatch));
379                 break;
380             default:
381                 errorExit(C->getLocation(), "invalid argument [1]");
382         }
383         ++Count;
384     }
385     assert(!this->PrevPatternmatch);
386     this->PrevType.reset();
387 }
388
389 void TypecheckVisitor::visit(ListPattern *N) {
390     auto Pats = std::make_shared<ListPatternmatch>(N->getLocation());
391     for (auto P : *N->getPatterns()) {
392         this->doVisit(P);
393         Pats->append(this->PrevPatternmatch);
394     }
395     this->PrevPatternmatch = Pats;
396     this->PrevType.reset();
397 }
398
399 void TypecheckVisitor::visit(TuplePattern *N) {
400     auto Pats =
401         std::make_shared<TuplePatternmatch>(N->getLocation());
402     for (auto P : *N->getPatterns()) {
403         this->doVisit(P);
404         Pats->append(this->PrevPatternmatch);
405     }
406     this->PrevPatternmatch = Pats;
407     this->PrevType.reset();
408 }
409
410 void TypecheckVisitor::visit(SeqDeclaration *N) {
411     for (auto C : *N)
412         this->doVisit(C);
413 }
414
415 void TypecheckVisitor::addDefaults() {
416     // Add default type declarations:
417     this->TypenameTypes.insert("int", this->IntTypedef,
418         this->InsertionScope);
419     this->TypenameTypes.insert("bool", this->BoolTypedef,
420         this->InsertionScope);
421     this->TypenameTypes.insert("unit", this->UnitTypedef,

```

```

418     this->InsertionScope);
419 auto ListDatatype = getListTypedef();
420 this->TypenameTypes.insert("list", ListDatatype,
421     this->InsertionScope);
422 auto RefDatatype = getRefTypedef();
423 this->TypenameTypes.insert("ref", RefDatatype,
424     this->InsertionScope);
425 auto ArrayDatatype = getArrayTypedef();
426 this->TypenameTypes.insert("array", ArrayDatatype,
427     this->InsertionScope);
428
429 this->ConstnameTypes.insert("true", this->BoolTypedef,
430     this->InsertionScope);
431 this->ConstnameTypes.insert("false", this->BoolTypedef,
432     this->InsertionScope);
433
434 auto ConsOf = std::make_shared<ProductTypedef>();
435 ConsOf->append(
436     std::make_shared<VariableTypedef>("'a", this->TypenamePrefix
437         + "'a"));
438 auto ConsList = getListTypedef();
439 ConsList->setArgument({0, 0}, std::make_shared<VariableTypedef>(
440     "'a", this->TypenamePrefix +
441         "'a"));
442 ConsOf->append(ConsList);
443
444 this->ConstnameTypes.insert(
445     ":",
446     std::make_shared<DatatypeInstanceTypedef>(*ListDatatype,
447         ConsOf),
448     this->InsertionScope);
449 this->ConstnameTypes.insert(
450     "nil",
451     std::make_shared<DatatypeInstanceTypedef>(*ListDatatype),
452     this->InsertionScope);
453
454 // Add default function declarations:
455 {
456     auto PrintType = std::make_shared<FunctionTypedef>();
457     PrintType->append(
458         std::make_shared<VariableTypedef>("'a",
459             this->TypenamePrefix + "'a"));
460     PrintType->append(this->UnitTypedef);
461     this->ValnameTypes.insert("print", PrintType,
462         this->InsertionScope);
463 }
464 {
465     auto PlusType = std::make_shared<FunctionTypedef>();
466     auto Tup = std::make_shared<ProductTypedef>();
467     Tup->append(this->IntTypedef);
468     Tup->append(this->IntTypedef);
469     PlusType->append(Tup);
470     PlusType->append(this->IntTypedef);
471     this->ValnameTypes.insert("+", PlusType, this->InsertionScope);
472 }
473 {
474     auto Type = std::make_shared<FunctionTypedef>();
475     auto Tup = std::make_shared<ProductTypedef>();
476     Tup->append(this->IntTypedef);
477     Tup->append(this->IntTypedef);
478     Type->append(Tup);
479     Type->append(this->IntTypedef);

```

```

467     this->ValnameTypes.insert("-", Type, this->InsertionScope);
468 }
469 {
470     auto Type = std::make_shared<FunctionTypedef>();
471     auto Tup = std::make_shared<ProductTypedef>();
472     Tup->append(this->IntTypedef);
473     Tup->append(this->IntTypedef);
474     Type->append(Tup);
475     Type->append(this->IntTypedef);
476     this->ValnameTypes.insert("*", Type, this->InsertionScope);
477 }
478 {
479     auto Type = std::make_shared<FunctionTypedef>();
480     auto Tup = std::make_shared<ProductTypedef>();
481     Tup->append(this->IntTypedef);
482     Tup->append(this->IntTypedef);
483     Type->append(Tup);
484     Type->append(this->IntTypedef);
485     this->ValnameTypes.insert("div", Type, this->InsertionScope);
486 }
487 {
488     auto Type = std::make_shared<FunctionTypedef>();
489     auto Tup = std::make_shared<ProductTypedef>();
490     Tup->append(this->IntTypedef);
491     Tup->append(this->IntTypedef);
492     Type->append(Tup);
493     Type->append(this->IntTypedef);
494     this->ValnameTypes.insert("mod", Type, this->InsertionScope);
495 }
496 {
497     auto Type = std::make_shared<FunctionTypedef>();
498     auto Tup = std::make_shared<ProductTypedef>();
499     auto Ref = getRefTypedef();
500     Ref->setArgument({0, 0}, std::make_shared<VariableTypedef>(
501         "'a", this->TypenamePrefix +
502         "'a"));
503     Tup->append(Ref);
504     Tup->append(
505         std::make_shared<VariableTypedef>("'a",
506             this->TypenamePrefix + "'a"));
507     Type->append(Tup);
508     Type->append(this->UnitTypedef);
509     this->ValnameTypes.insert("=", Type, this->InsertionScope);
510 }
511 {
512     auto Type = std::make_shared<FunctionTypedef>();
513     Type->append(this->IntTypedef);
514     Type->append(this->IntTypedef);
515     this->ValnameTypes.insert("~", Type, this->InsertionScope);
516 }
517 {
518     auto Type = std::make_shared<FunctionTypedef>();
519     Type->append(this->BoolTypedef);
520     Type->append(this->BoolTypedef);
521     this->ValnameTypes.insert("not", Type, this->InsertionScope);
522 }
523 {
524     auto Type = std::make_shared<FunctionTypedef>();
525     auto RefInstance = std::make_shared<DatatypeInstanceTypedef>(
526         *getRefTypedef(),
527         std::make_shared<VariableTypedef>("'a",
528             this->TypenamePrefix + "'a"));

```

```

526     RefInstance->setArgument({0, 0},
527         std::make_shared<VariableTypedef>(
528             "a",
529             this->TypenamePrefix
530             + "a"));
531
532     Type->append(
533         std::make_shared<VariableTypedef>("a",
534             this->TypenamePrefix + "a"));
535     Type->append(RefInstance);
536     this->ValnameTypes.insert("ref", Type, this->InsertionScope);
537 }
538 {
539     auto Type = std::make_shared<FunctionTypedef>();
540     auto Tup = std::make_shared<ProductTypedef>();
541     auto ArrayInstance = std::make_shared<DatatypeInstanceTypedef>(
542         *getArrayTypedef(),
543         std::make_shared<VariableTypedef>("a",
544             this->TypenamePrefix + "a"));
545     ArrayInstance->setArgument({0, 0},
546         std::make_shared<VariableTypedef>(
547             "a",
548             this->TypenamePrefix
549             + "a"));
550
551     Tup->append(this->IntTypedef);
552     Tup->append(
553         std::make_shared<VariableTypedef>("a",
554             this->TypenamePrefix + "a"));
555     Type->append(Tup);
556     Type->append(ArrayInstance);
557     this->ValnameTypes.insert("Array.array", Type,
558         this->InsertionScope);
559 }
560 {
561     auto Type = std::make_shared<FunctionTypedef>();
562     auto Tup = std::make_shared<ProductTypedef>();
563     auto Ary = getArrayTypedef();
564     Ary->setArgument({0, 0}, std::make_shared<VariableTypedef>(
565         "a", this->TypenamePrefix +
566         "a"));
567
568     Tup->append(Ary);
569     Tup->append(this->IntTypedef);
570     Tup->append(
571         std::make_shared<VariableTypedef>("a",
572             this->TypenamePrefix + "a"));
573     Type->append(Tup);
574     Type->append(this->UnitTypedef);
575     this->ValnameTypes.insert("Array.update", Type,
576         this->InsertionScope);
577 }
578 {
579     auto Ary = getArrayTypedef();
580     this->ValnameTypes.insert("Array.empty", Ary,
581         this->InsertionScope);
582 }
583 {
584     auto Type = std::make_shared<FunctionTypedef>();
585     auto Tup = std::make_shared<ProductTypedef>();
586     auto Ary = getArrayTypedef();
587     Ary->setArgument({0, 0}, std::make_shared<VariableTypedef>(
588         "a", this->TypenamePrefix +
589         "a"));

```

```

573     Tup->append(Ary);
574     Tup->append(this->IntTypedef);
575     Type->append(Tup);
576     Type->append(
577         std::make_shared<VariableTypedef>(" 'a",
578             this->TypenamePrefix + " 'a"));
579     this->ValnameTypes.insert("Array.get", Type,
580         this->InsertionScope);
581 }
582 {
583     auto Type = std::make_shared<FunctionTypedef>();
584     auto Ary = getArrayTypedef();
585     Ary->setArgument({0, 0}, std::make_shared<VariableTypedef>(
586         " 'a", this->TypenamePrefix +
587         " 'a"));
588     Type->append(Ary);
589     Type->append(this->IntTypedef);
590     this->ValnameTypes.insert("Array.length", Type,
591         this->InsertionScope);
592 }
593 {
594     auto Type = std::make_shared<FunctionTypedef>();
595     auto RefInstance = std::make_shared<DatatypeInstanceTypedef>(
596         *getRefTypedef(),
597         std::make_shared<VariableTypedef>(" 'a",
598             this->TypenamePrefix + " 'a"));
599     RefInstance->setArgument({0, 0},
600         std::make_shared<VariableTypedef>(
601             " 'a",
602             this->TypenamePrefix
603             + " 'a"));
604     Type->append(RefInstance);
605     Type->append(
606         std::make_shared<VariableTypedef>(" 'a",
607             this->TypenamePrefix + " 'a"));
608     this->ValnameTypes.insert("!", Type, this->InsertionScope);
609 }
610 {
611     auto Type = std::make_shared<FunctionTypedef>();
612     auto Tup = std::make_shared<ProductTypedef>();
613     Tup->append(this->IntTypedef);
614     Tup->append(this->IntTypedef);
615     Type->append(Tup);
616     Type->append(this->BoolTypedef);
617     this->ValnameTypes.insert("=", Type, this->InsertionScope);
618 }
619 {
620     auto Type = std::make_shared<FunctionTypedef>();
621     auto Tup = std::make_shared<ProductTypedef>();
622     Tup->append(this->IntTypedef);
623     Tup->append(this->IntTypedef);
624     Type->append(Tup);
625     Type->append(Tup);

```

```

626     Type->append(this->BoolTypedef);
627     this->ValnameTypes.insert("<", Type, this->InsertionScope);
628 }
629 {
630     auto Type = std::make_shared<FunctionTypedef>();
631     auto Tup = std::make_shared<ProductTypedef>();
632     Tup->append(this->IntTypedef);
633     Tup->append(this->IntTypedef);
634     Type->append(Tup);
635     Type->append(this->BoolTypedef);
636     this->ValnameTypes.insert(">", Type, this->InsertionScope);
637 }
638 {
639     auto Type = std::make_shared<FunctionTypedef>();
640     auto Tup = std::make_shared<ProductTypedef>();
641     Tup->append(this->IntTypedef);
642     Tup->append(this->IntTypedef);
643     Type->append(Tup);
644     Type->append(this->BoolTypedef);
645     this->ValnameTypes.insert("<=", Type, this->InsertionScope);
646 }
647 {
648     auto Type = std::make_shared<FunctionTypedef>();
649     auto Tup = std::make_shared<ProductTypedef>();
650     Tup->append(this->IntTypedef);
651     Tup->append(this->IntTypedef);
652     Type->append(Tup);
653     Type->append(this->BoolTypedef);
654     this->ValnameTypes.insert(">=", Type, this->InsertionScope);
655 }
656 }
657
658 void TypecheckVisitor::visit(Root *N) {
659     this->pushFuncVals();
660     N->setFunctionVals(this->peekFuncVals());
661     this->enterScope();
662
663     this->addDefaults();
664
665     this->enterScope();
666     for (auto C : *N)
667         this->doVisit(C);
668
669     this->leaveScope();
670     this->leaveScope();
671     this->popFuncVals();
672 }
673
674 void TypecheckVisitor::visit(ValDeclaration *N) {
675     bool PrevHasApply = this->ExprHasValnameApply;
676
677     this->doVisit(N->getDest());
678     auto DestType = this->getPrevPatternType();
679     if (!DestType)
680         errorExit(N->getDest()->getLocation(),
681                 "pattern missing type specification");
682     auto Match = this->PrevPatternmatch;
683     auto NameTypes = Match->getNameTypes();
684
685     for (auto &S : this->PatternIdenInsertionOrder) {
686         assert(NameTypes.count(S));
687         auto Def = NameTypes[S];

```

```

688         if (Def->getKind() == Typedef::SimpleBuildin)
689             this->peekFuncVals()->addVal(false);
690         else
691             this->peekFuncVals()->addVal(true);
692     }
693     this->PatternIdenInsertionOrder.clear();
694
695     this->ExprHasValnameApply = false;
696     this->doVisit(N->getSource());
697     auto SourceType = this->PrevType;
698
699     if (this->cannotBeAssignWith(DestType,
700         N->getSource()->getLocation(),
701         SourceType,
702         this->ExprHasValnameApply,
703         false))
704         errorExit(N->getLocation(),
705             "pattern and expression have conflicting types");
706
707     this->ValnameTypes.mergeWith(NameTypes, 0);
708     this->ConstnameTypes.erase(NameTypes, 0);
709     this->ExprHasValnameApply = PrevHasApply;
710 }
711
712 void TypecheckVisitor::visit(NonfixDeclaration *N) { // Nop.
713 }
714
715 void TypecheckVisitor::visit(InfixDeclaration *N) { // Nop.
716 }
717
718 void TypecheckVisitor::visit(InfixRDeclaration *N) { // Nop.
719 }
720
721 static Typemap::MapType mergeParamNameTypes(SourceLocation L,
722     const std::string
723         &FunName,
724     std::vector<Typemap::MapType>
725         &V) {
726     Typemap::MapType Ret;
727     for (auto &&M : V) {
728         for (auto &&P : M) {
729             if (Ret.count(P.first) || P.first == FunName)
730                 errorExit(L, "multiple declarations of identifier '" +
731                     P.first + "'");
732             Ret[P.first] = P.second;
733         }
734     }
735     return Ret;
736 }
737
738 void TypecheckVisitor::visit(FunPatternDeclaration *N) {
739     bool PrevHasApply = this->ExprHasValnameApply;
740     bool PrevFunParam = this->AddVariableTypeToTypenames;
741
742     auto IDParams = N->getIDParams();
743     auto TypePat = IDParams->asTypePattern();
744     if (!TypePat)
745         errorExit(N->getLocation(), "missing function return type");
746
747     auto ReturnType = TypePat->getType();
748     auto TypeLHS = TypePat->getPattern();

```



```

744 auto Apply = TypeLHS->asApplyPattern();
745 if (!Apply)
746     errorExit(TypeLHS->getLocation(), "function missing
747         parameter");
748 auto FunName = Apply->front()->asLongIdentifierPattern();
749 if (!FunName || FunName->getIDs()->size() != 1)
750     errorExit(N->getLocation(), "invalid function identifier");
751 auto FunID = FunName->getIDs()->toString();
752
753 auto FunType = std::make_shared<FunctionTypedef>();
754 std::vector<Typemap::MapType> ParamNameTypes;
755 for (size_t I = 1; I < Apply->size(); ++I) {
756     this->AddVariableTypeToTypenames = true;
757     this->doVisit((*Apply)[I]);
758     this->PrevType = this->getPrevPatternType();
759     if (!this->PrevType)
760         errorExit((*Apply)[I]->getLocation(), "missing type
761             specification");
762     if (this->PrevType->getKind() == Typedef::SimpleBuildin)
763         (*Apply)[I]->setHasSimpleType();
764     FunType->append(this->PrevType);
765     ParamNameTypes.push_back(this->PrevPatternmatch->getNameTypes());
766 }
767
768 auto PTypes = mergeParamNameTypes(N->getLocation(), FunID,
769     ParamNameTypes);
770 for (auto &S : this->PatternIdenInsertionOrder) {
771     assert(PTypes.count(S));
772     auto Def = PTypes[S];
773     if (Def->getKind() == Typedef::SimpleBuildin)
774         this->peekFuncVals()->addVal(false);
775     else
776         this->peekFuncVals()->addVal(true);
777 }
778 this->PatternIdenInsertionOrder.clear();
779
780 this->AddVariableTypeToTypenames = true;
781 this->ExprHasValnameApply = false;
782 this->doVisit(ReturnType);
783 auto RetTT = this->PrevType;
784 FunType->append(RetTT);
785
786 this->ValnameTypes.insert(FunID, FunType, 1);
787 this->ConstnameTypes.erase(FunID, 1);
788
789 this->enterScope();
790 this->ValnameTypes.mergeWith(PTypes, 0);
791 this->ConstnameTypes.erase(PTypes, 0);
792 this->doVisit(N->getDef());
793 this->leaveScope();
794
795 if (this->cannotBeAssignWith(this->PrevType,
796     ReturnType->getLocation(), RetTT,
797     this->ExprHasValnameApply, false,
798     true))
799     errorExit(N->getLocation(),
800         "conflicting return type and function expression");
801
802 this->PrevType.reset(FunType->clone());
803 this->PrevFunName = std::move(FunID);
804 this->ExprHasValnameApply = PrevHasApply;
805 this->AddVariableTypeToTypenames = PrevFunParam;

```

```

801 }
802
803 void TypecheckVisitor::visit(FunDeclaration *N) {
804     std::string FunName;
805     std::shared_ptr<Typedef> FunType;
806
807     this->pushFuncVals();
808     N->setFunctionVals(this->peekFuncVals());
809     this->enterScope();
810
811     for (auto &&F : *N) {
812         this->doVisit(F);
813         if (!FunType) {
814             FunName = this->PrevFunName;
815             FunType = this->PrevType;
816         } else {
817             if (this->PrevFunName != FunName)
818                 errorExit(N->getLocation(), "conflicting function case
819                                     identifiers");
820             if (*this->PrevType != *FunType)
821                 errorExit(N->getLocation(), "conflicting function case
822                                     types");
823         }
824     }
825     FunType->unmarkInScope();
826     this->leaveScope();
827     this->popFuncVals();
828     this->peekFuncVals()->addVal(true);
829 }
830
831 void TypecheckVisitor::visit(DatatypeBareInstanceDeclaration *N) {
832     auto Def = this->CurrDatatypeTypedef;
833     this->ConstnameTypes.insert(
834         *N->getID()->getID(),
835         std::make_shared<DatatypeInstanceTypedef>(*Def, nullptr),
836         this->InsertionScope);
837     this->ValnameTypes.erase(*N->getID()->getID(),
838         this->InsertionScope);
839 }
840
841 void TypecheckVisitor::visit(DatatypeTypedInstanceDeclaration *N) {
842     auto Def = this->CurrDatatypeTypedef;
843
844     this->doVisit(N->getType());
845     auto ParamType = this->PrevType;
846     if (!ParamType->isType())
847         errorExit(N->getType()->getLocation(), "not a type");
848     if (!ParamType->isTypeComplete())
849         errorExit(N->getType()->getLocation(), "incomplete type");
850
851     this->ConstnameTypes.insert(
852         *N->getID()->getID(),
853         std::make_shared<DatatypeInstanceTypedef>(*Def, ParamType),
854         this->InsertionScope);
855     this->ValnameTypes.erase(*N->getID()->getID(),
856         this->InsertionScope);
857 }
858
859 static bool vectorContains(const std::vector<std::string> V,
860                           const std::string S) {
861     for (auto &&E : V)
862         if (E == S)

```

```

859         return true;
860     return false;
861 }
862
863 void TypecheckVisitor::visit(BareDatatypeDeclaration *N) {
864     this->enterScope();
865
866     std::vector<std::string> Ps;
867     auto Typedef =
868         std::make_shared<DatatypeTypedef>(*N->getID()->getID(), Ps);
869
870     ++this->InsertionScope;
871
872     this->CurrDatatypeTypedef = Typedef;
873     this->TypenameTypes.insert(*N->getID()->getID(), Typedef,
874                               this->InsertionScope);
875
876     this->doVisit(N->getDef());
877
878     --this->InsertionScope;
879
880     this->leaveScope();
881 }
882
883 void TypecheckVisitor::visit(TypedDatatypeDeclaration *N) {
884     std::vector<std::string> Paramnames;
885     this->enterScope();
886     for (auto &&P : *N->getTypeParams()) {
887         const std::string &Name = *P->getID()->getID();
888         if (vectorContains(Paramnames, Name))
889             errorExit(P->getLocation(),
890                       "multiple uses of variable type name in type
891                       parameter list");
892         Paramnames.push_back(this->TypenamePrefix + Name);
893         this->TypenameTypes.insert(Name,
894                                   std::make_shared<VariableTypedef>(
895                                       Name,
896                                       this->TypenamePrefix
897                                       + Name),
898                                   this->InsertionScope);
899     }
900     auto Typedef =
901         std::make_shared<DatatypeTypedef>(*N->getID()->getID(),
902                                           std::move(Paramnames));
903
904     ++this->InsertionScope;
905
906     this->CurrDatatypeTypedef = Typedef;
907     this->TypenameTypes.insert(*N->getID()->getID(), Typedef,
908                               this->InsertionScope);
909
910     this->doVisit(N->getDef());
911
912     --this->InsertionScope;
913     this->leaveScope();
914 }
915
916 void TypecheckVisitor::visit(LongIdentifierType *N) {
917     auto T = TypenameTypes.get(N->getIDs()->toString());
918     if (!T)
919         errorExit(N->getLocation(), "unknown type " +
920                 N->getIDs()->toString());
921 }

```

```

914     this->PrevType = T;
915 }
916
917 void TypecheckVisitor::visit(VariableType *N) {
918     VariableTypedef *T;
919     std::shared_ptr<Typedef> ScopeVar1 =
920         this->TypenameTypes.get(*N->getID()->getID());
921     if (ScopeVar1) {
922         assert(ScopeVar1->getKind() == Typedef::Variable);
923         auto ScopeVar = static_cast<VariableTypedef>
924             *(>ScopeVar1.get());
925         T = new VariableTypedef(ScopeVar->getUserTypename(),
926                                 ScopeVar->getInternTypename());
927         T->setIsInScope(true);
928         this->PrevType.reset(T);
929     } else {
930         T = new VariableTypedef(*N->getID()->getID(),
931                                 this->TypenamePrefix +
932                                 *N->getID()->getID());
933         auto TT = std::shared_ptr<VariableTypedef>(T);
934         if (this->AddVariableTypeToTypenames) {
935             TT->setIsInScope(true);
936             this->TypenameTypes.insert(*N->getID()->getID(), TT, 0);
937         }
938         this->PrevType = TT;
939     }
940 }
941
942 void TypecheckVisitor::visit(SeqType *N) {
943     for (auto C : *N)
944         this->doVisit(C);
945 }
946
947 void TypecheckVisitor::visit(SeqVariableType *N) {
948     fatalExit("Typecheckvisitor should not vist SeqVariableType");
949 }
950
951 void TypecheckVisitor::visit(TupleType *N) {
952     auto Tup = std::make_shared<TupleTypedef>();
953     for (auto &&C : *N->getTypes()) {
954         this->doVisit(C);
955         Tup->append(this->PrevType);
956     }
957     this->PrevType = Tup;
958 }
959
960 void TypecheckVisitor::visit(ApplyType *N) {
961     std::shared_ptr<Typedef> Typ;
962     std::shared_ptr<Typedef> First;
963
964     for (auto &&C : *N) {
965         this->doVisit(C);
966         auto Next = this->PrevType;
967         if (Typ)
968             Next->setArgument(C->getLocation(), Typ);
969         else
970             First = Next;
971         Typ = Next;
972     }
973     if (First->isTypeMissingArgument())
974         errorExit(N->getLocation(), "missing type argument 1");
975 }

```

```

974     this->PrevType = Typ;
975 }
976
977 void TypecheckVisitor::visit(ProductType *N) {
978     auto T = std::make_shared<ProductTypedef>();
979     for (auto C : *N) {
980         this->doVisit(C);
981         T->append(this->PrevType);
982     }
983     this->PrevType = T;
984 }
985
986 void TypecheckVisitor::visit(FunctionType *N) {
987     auto T = std::make_shared<FunctionTypedef>();
988     for (auto C : *N) {
989         this->doVisit(C);
990         T->append(this->PrevType);
991     }
992     this->PrevType = T;
993 }

```

Listing 79: lib/Typecheck/Patternmatch.h

```

1  #ifndef LLVM_TOOLS_SSML_LIB_TYPECHECK_PATTERNMATCH_H
2  #define LLVM_TOOLS_SSML_LIB_TYPECHECK_PATTERNMATCH_H
3
4  #include "Typemap.h"
5
6  #include "ssml/Common/SourceLocation.h"
7
8  #include <string>
9  #include <memory>
10
11 namespace ssml {
12 namespace typecheck {
13 class Typedef;
14 class DatatypeInstanceTypedef;
15 class DatatypeTypedef;
16 } // End namespace typecheck.
17 } // End namespace ssml.
18
19 namespace ssml {
20 namespace typecheck {
21 class Patternmatch {
22 protected:
23     std::shared_ptr<Typedef> PatternType = nullptr;
24     SourceLocation Location;
25     Typemap::MapType NameTypes;
26     bool IsTypeMatched = false;
27     bool IsNameTypesReturned = false;
28
29 protected:
30     Patternmatch(SourceLocation);
31     void appendNameTypes(Typemap::MapType);
32
33 public:
34     virtual ~Patternmatch() = 0;
35     virtual void matchType(std::shared_ptr<Typedef>);
36     virtual Typemap::MapType getNameTypes();
37     std::shared_ptr<Typedef> getPatternType() { return
        this->PatternType; }

```

```

38     virtual std::shared_ptr<Typedef> getFreeMatchPatternType();
39 };
40 } // End namespace typecheck.
41 } // End namespace ssml.
42
43 namespace ssml {
44     namespace typecheck {
45         class WildcardPatternmatch : public Patternmatch {
46         public:
47             WildcardPatternmatch(SourceLocation);
48             void matchType(std::shared_ptr<Typedef>) override;
49         };
50     } // End namespace typecheck.
51 } // End namespace ssml.
52
53 namespace ssml {
54     namespace typecheck {
55         class LongIdentifierPatternmatch : public Patternmatch {
56         private:
57             const Typemap &ConstnameTypes;
58             std::string ID;
59         public:
60             LongIdentifierPatternmatch(SourceLocation, const Typemap
61                                     &ConstnameTypes,
62                                     std::string &&ID);
63             void matchType(std::shared_ptr<Typedef>) override;
64             std::shared_ptr<Typedef> getFreeMatchPatternType() override;
65         };
66     } // End namespace typecheck.
67 } // End namespace ssml.
68
69 namespace ssml {
70     namespace typecheck {
71         class TuplePatternmatch : public Patternmatch {
72         protected:
73             std::vector<std::shared_ptr<Patternmatch>> Matches;
74         public:
75             TuplePatternmatch(SourceLocation);
76             virtual void append(std::shared_ptr<Patternmatch>);
77             void matchType(std::shared_ptr<Typedef>) override;
78             Typemap::MapType getNameTypes() override;
79             std::shared_ptr<Typedef> getFreeMatchPatternType() override;
80         };
81     } // End namespace typecheck.
82 } // End namespace ssml.
83
84 namespace ssml {
85     namespace typecheck {
86         class ListPatternmatch : public TuplePatternmatch {
87         public:
88             ListPatternmatch(SourceLocation);
89             void append(std::shared_ptr<Patternmatch>) override;
90             void matchType(std::shared_ptr<Typedef>) override;
91             std::shared_ptr<Typedef> getFreeMatchPatternType() override;
92         };
93     } // End namespace typecheck.
94 } // End namespace ssml.
95
96 namespace ssml {
97     namespace typecheck {

```

```

99 class LiteralPatternmatch : public Patternmatch {
100 private:
101     std::shared_ptr<Typedef> LiteralType;
102
103 public:
104     LiteralPatternmatch(SourceLocation, std::shared_ptr<Typedef>
        LiteralType);
105     void matchType(std::shared_ptr<Typedef>) override;
106     std::shared_ptr<Typedef> getPreMatchPatternType() override;
107 };
108 } // End namespace typecheck.
109 } // End namespace ssml.
110
111 namespace ssml {
112 namespace typecheck {
113 class ApplyPatternmatch : public Patternmatch {
114 private:
115     std::shared_ptr<Patternmatch> Left;
116     std::shared_ptr<Patternmatch> Right;
117     SourceLocation LeftLocation;
118     SourceLocation RightLocation;
119
120 private:
121     std::shared_ptr<Typedef> match(
122         const std::vector<std::pair<std::string,
            std::shared_ptr<Typedef>>> &V,
123         std::shared_ptr<Typedef> Match);
124     std::shared_ptr<Typedef> simpleBuildinMatch(
125         const std::vector<std::pair<std::string,
            std::shared_ptr<Typedef>>> &V,
126         std::shared_ptr<Typedef> Match);
127     std::shared_ptr<Typedef> productMatch(
128         const std::vector<std::pair<std::string,
            std::shared_ptr<Typedef>>> &V,
129         std::shared_ptr<Typedef> Match);
130     std::shared_ptr<Typedef> functionMatch(
131         const std::vector<std::pair<std::string,
            std::shared_ptr<Typedef>>> &V,
132         std::shared_ptr<Typedef> Match);
133     std::shared_ptr<Typedef> datatypeMatch(
134         const std::vector<std::pair<std::string,
            std::shared_ptr<Typedef>>> &V,
135         std::shared_ptr<Typedef> Match);
136     std::shared_ptr<Typedef> variableMatch(
137         const std::vector<std::pair<std::string,
            std::shared_ptr<Typedef>>> &V,
138         std::shared_ptr<Typedef> Match);
139
140 public:
141     ApplyPatternmatch(SourceLocation LeftLocation, SourceLocation
        RightLocation,
142                     std::shared_ptr<Patternmatch> LHS,
143                     std::shared_ptr<Patternmatch> RHS);
144     void matchType(std::shared_ptr<Typedef>) override;
145     std::shared_ptr<Typedef> getPreMatchPatternType() override;
146     Typemap::MapType getNameTypes() override;
147 };
148 } // End namespace typecheck.
149 } // End namespace ssml.
150
151 #endif // LLVM_TOOLS_SSML_LIB_TYPECHECK_PATTERNMATCH_H

```

Listing 80: lib/Typecheck/TypecheckVisitor.h

```

1  #ifndef LLVM_TOOLS_SSML_LIB_TYPECHECK_TYPECHECKVISITOR_H
2  #define LLVM_TOOLS_SSML_LIB_TYPECHECK_TYPECHECKVISITOR_H
3
4  #include "Typemap.h"
5  #include "Typedef.h"
6
7  #include "ssml/AST/FunctionVals.h"
8  #include "ssml/AST/Visitor.h"
9
10 namespace ssml {
11 namespace ast {
12 class Node;
13 class Pattern;
14 } // End namespace ast.
15 } // End namespace ssml.
16
17 namespace ssml {
18 namespace typecheck {
19 class Patternmatch;
20 class Typedef;
21 } // End namespace typecheck.
22 } // End namespace ssml.
23
24 namespace ssml {
25 namespace typecheck {
26 class TypecheckVisitor : public ssml::ast::Visitor {
27 private:
28     Typemap TypenameTypes;
29     Typemap ValnameTypes;
30     Typemap ConstnameTypes;
31     size_t InsertionScope = 0;
32     /**
33      * TypenamePrefix is used to make sure user defined datatypes
34      * are distinct
35      * when declared in different scopes, even though the name is
36      * the same. E.g.:
37      * datatype A = ...
38      * val x:A = ...
39      * let
40      *   datatype A = ...
41      *   val y:A = x (* Error type of x != type of y *)
42      * in
43      *   ...
44      * end
45      */
46     std::string TypenamePrefix;
47
48     std::shared_ptr<Patternmatch> PrevPatternmatch;
49     std::shared_ptr<DatatypeTypedef> CurrDatatypeTypedef;
50     std::shared_ptr<Typedef> PrevType;
51
52     std::shared_ptr<Typedef> IntTypedef =
53         std::make_shared<SimpleBuildinTypedef>("int");
54     std::shared_ptr<Typedef> BoolTypedef =
55         std::make_shared<SimpleBuildinTypedef>("bool");
56     std::shared_ptr<Typedef> UnitTypedef =
57         std::make_shared<ProductTypedef>();
58
59     std::string PrevFunName;
60
61     bool ApplyExprWasVal = false;

```

```

59     bool ExprHasValnameApply = false;
60
61     bool AddVariableTypeToTypenames = false;
62
63     std::vector<std::string> PatternIdenInsertionOrder;
64
65     std::vector<std::shared_ptr<ssml::ast::FunctionVals>>
        FuncValsStack;
66
67 private:
68     void enterScope();
69     void leaveScope();
70     void addDefaults();
71     void doVisit(std::shared_ptr<ssml::ast::Node>);
72     void doVisit(ssml::ast::Node *);
73     bool cannotBeAssignWith(std::shared_ptr<Typedef> DestType,
74                             SourceLocation SourceL,
75                             std::shared_ptr<Typedef> SourceType,
76                             bool ExprHasValApplication, bool
77                                 ScopedVarIsVar,
78                                 bool IsRet = false);
79     std::shared_ptr<Typedef> getPrevPatternType();
80     void pushFuncVals() {
81         this->FuncValsStack.push_back(std::shared_ptr<ssml::ast::FunctionVals>(
82             new ssml::ast::FunctionVals()));
83     }
84     void popFuncVals() { this->FuncValsStack.pop_back(); }
85     std::shared_ptr<ssml::ast::FunctionVals> peekFuncVals() {
86         return this->FuncValsStack.back();
87     }
88 public:
89     void visit(ssml::ast::IntLiteral *) override;
90
91     void visit(ssml::ast::ShortIdentifier *) override;
92     void visit(ssml::ast::LongIdentifier *) override;
93     void visit(ssml::ast::SeqShortIdentifier *) override;
94     void visit(ssml::ast::SeqLongIdentifier *) override;
95
96     void visit(ssml::ast::Match *) override;
97     void visit(ssml::ast::SeqMatch *) override;
98
99     void visit(ssml::ast::SeqExpression *) override;
100    void visit(ssml::ast::LiteralExpression *) override;
101    void visit(ssml::ast::LongIdentifierExpression *) override;
102    void visit(ssml::ast::TupleExpression *) override;
103    void visit(ssml::ast::ListExpression *) override;
104    void visit(ssml::ast::ApplyExpression *) override;
105    void visit(ssml::ast::OrElseExpression *) override;
106    void visit(ssml::ast::AndAlsoExpression *) override;
107    void visit(ssml::ast::LetExpression *) override;
108    void visit(ssml::ast::IfExpression *) override;
109    void visit(ssml::ast::WhileExpression *) override;
110    void visit(ssml::ast::LambdaExpression *) override;
111
112    void visit(ssml::ast::SeqPattern *) override;
113    void visit(ssml::ast::LiteralPattern *) override;
114    void visit(ssml::ast::WildcardPattern *) override;
115    void visit(ssml::ast::LongIdentifierPattern *) override;
116    void visit(ssml::ast::TypePattern *) override;
117    void visit(ssml::ast::ApplyPattern *) override;
118    void visit(ssml::ast::ListPattern *) override;

```

```

119     void visit(ssml::ast::TuplePattern *) override;
120
121     void visit(ssml::ast::SeqDeclaration *) override;
122     void visit(ssml::ast::Root *) override;
123     void visit(ssml::ast::ValDeclaration *) override;
124     void visit(ssml::ast::NonfixDeclaration *) override;
125     void visit(ssml::ast::InfixDeclaration *) override;
126     void visit(ssml::ast::InfixRDeclaration *) override;
127     void visit(ssml::ast::FunPatternDeclaration *) override;
128     void visit(ssml::ast::FunDeclaration *) override;
129     void visit(ssml::ast::DatatypeBareInstanceDeclaration *)
130         override;
131     void visit(ssml::ast::DatatypeTypedInstanceDeclaration *)
132         override;
133
134     void visit(ssml::ast::LongIdentifierType *) override;
135     void visit(ssml::ast::VariableType *) override;
136     void visit(ssml::ast::SeqType *) override;
137     void visit(ssml::ast::SeqVariableType *) override;
138     void visit(ssml::ast::TupleType *) override;
139     void visit(ssml::ast::ApplyType *) override;
140     void visit(ssml::ast::ProductType *) override;
141     void visit(ssml::ast::FunctionType *) override;
142 };
143 } // End namespace typecheck.
144 } // End namespace ssml.
145
146 #endif // LLVM_TOOLS_SSML_LIB_TYPECHECK_TYPECHECKVISITOR_H

```

Listing 81: lib/Typecheck/Typedef.cpp

```

1  #include "Typedef.h"
2
3  #include "ssml/Common/ErrorMessage.h"
4  #include "ssml/Common/FatalExit.h"
5
6  #include "llvm/ADT/StringRef.h"
7
8  #include "llvm/Support/raw_ostream.h"
9
10 #include <map>
11
12 using namespace ssml;
13 using namespace ssml::typecheck;
14
15 static std::shared_ptr<Typedef>
16     &vectorGetRef(Typedef::VartypeVector &V,
17                  const std::string
18                  &S) {
19     for (auto &&P : V) {
20         if (P.first == S) {
21             assert(!P.second);
22             return P.second;
23         }
24     }
25     fatalError("cannot return ref!");
26     return V[0].second;
27 }

```

```

27 static std::shared_ptr<Typedef> vectorGet(const
    Typedef::VartypeVector &V,
28                                     const std::string &S) {
29     for (auto &&P : V) {
30         if (P.first == S) {
31             assert(!P.second);
32             return P.second;
33         }
34     }
35     return nullptr;
36 }
37
38 static std::shared_ptr<Typedef> vectorGet(const
    Typedef::VartypeVector *V,
39                                     const std::string &S) {
40     return vectorGet(*V, S);
41 }
42
43 Typedef::~Typedef() = default;
44
45 bool Typedef::canBeAssignWith(const Typedef &Y, const
    VartypeVector *V,
46                                     bool ScopedVarIsVar) const {
47     return !this->cannotBeAssignWith(Y, V, ScopedVarIsVar);
48 }
49
50 bool Typedef::cannotBeAssignWith(const Typedef &Y, const
    VartypeVector *V,
51                                     bool ScopedVarIsVar) const {
52     return !this->canBeAssignWith(Y, V, ScopedVarIsVar);
53 }
54
55 bool Typedef::isTypeMissingArgument() const { return false; }
56
57 bool Typedef::operator==(const Typedef &Y) const { return !(*this
    != Y); }
58
59 bool Typedef::operator!=(const Typedef &Y) const { return !(*this
    == Y); }
60
61 bool Typedef::isType() const { return true; }
62
63 std::shared_ptr<Typedef>
64 Typedef::funApplication(SourceLocation L, std::shared_ptr<Typedef>
    T) const {
65     errorExit(L, "cannot apply argument here 1");
66     return nullptr;
67 }
68
69 SimpleBuildinTypedef::SimpleBuildinTypedef(std::string N)
70     : Typename(std::move(N)) {}
71
72 SimpleBuildinTypedef *SimpleBuildinTypedef::clone() const {
73     return new SimpleBuildinTypedef(*this);
74 }
75
76 void SimpleBuildinTypedef::unmarkInScope() {}
77
78 void SimpleBuildinTypedef::setArgument(SourceLocation L,
79                                     std::shared_ptr<Typedef> A)
80     {
    errorExit(L, "type argument(s) applied to simple build in type

```

```

81         , " " +
82         this->Typename + " ");
83     }
84     std::shared_ptr<Typedef>
85     SimpleBuildinTypedef::withoutVariableTypedefs(const VartypeVector
86         &) {
87         return std::shared_ptr<Typedef>(this->clone());
88     }
89     Typedef::VartypeVector
90     SimpleBuildinTypedef::getVariableTypes(SourceLocation,
91         std::shared_ptr<Typedef>)
92         const {
93         return {};
94     }
95     static const Typedef &getVarTypedef(const Typedef &X,
96         const Typedef::VartypeVector
97         *V) {
98         if (!V || X.getKind() != Typedef::Variable)
99             return X;
100         auto &Var = static_cast<const VariableTypedef &>(X);
101         auto Ret = vectorGet(V, Var.getInternTypename());
102         if (!Ret)
103             return X;
104         return *Ret;
105     }
106     bool SimpleBuildinTypedef::canBeAssignWith(const Typedef &X,
107         const
108         Typedef::VartypeVector
109         *V,
110         bool ScopedVarIsVar)
111         const {
112         auto &Y = getVarTypedef(X, V);
113         if (this->getKind() != Y.getKind())
114             return false;
115         auto YY = static_cast<const SimpleBuildinTypedef &>(Y);
116         return this->Typename == YY.Typename;
117     }
118     bool SimpleBuildinTypedef::hasVariable() const { return false; }
119     bool SimpleBuildinTypedef::operator==(const Typedef &Y) const {
120         return this->canBeAssignWith(Y, nullptr, false);
121     }
122     bool SimpleBuildinTypedef::isTypeComplete() const { return true; }
123     Typedef::Kind SimpleBuildinTypedef::getKind() const {
124         return Typedef::SimpleBuildin;
125     }
126     #if 0
127     ProductTypedef::ProductTypedef(const ProductTypedef &Y) {
128         for (auto &&T : Y.Typedefs)
129             this->Typedefs->push_back(std::shared_ptr<Typedef>(T->clone()));
130     }
131     #endif
132     ProductTypedef *ProductTypedef::clone() const {

```

```

136     return new ProductTypedef(*this);
137 }
138
139 void ProductTypedef::setArgument(SourceLocation L,
140     std::shared_ptr<Typedef>) {
141     errorExit(L, "tuple does not take type argument(s)");
142 }
143
144 void ProductTypedef::unmarkInScope() {
145     for (auto T : this->Typedefs)
146         T->unmarkInScope();
147 }
148
149 Typedef::VartypeVector
150 ProductTypedef::getVariableTypes(SourceLocation L,
151     std::shared_ptr<Typedef> Def)
152     const {
153     std::vector<std::pair<std::string, std::shared_ptr<Typedef>>>
154         Ret;
155
156     #if 0
157     if (Def->getKind() != this->getKind())
158         errorExit(L, "unexpected argument(s)");
159     #else
160     if (Def->getKind() != this->getKind())
161         return {};
162     #endif
163
164     auto PTypes = static_cast<ProductTypedef *>(Def.get())->Typedefs;
165     auto Size = this->Typedefs.size();
166     if (Size != PTypes.size())
167         errorExit(L, "unexpected number of type arguments 1");
168
169     for (size_t I = 0; I < Size; ++I) {
170         auto &T = this->Typedefs[I];
171
172         assert(!T);
173         assert(!PTypes[I]);
174         auto M = T->getVariableTypes(L, PTypes[I]);
175         for (auto &&Def : M) {
176             auto Elem = vectorGet(Ret, Def.first);
177             Typedef::VartypeVector VTypes;
178             if (Elem)
179                 VTypes = Elem->getVariableTypes(L, Def.second);
180             if (Elem && Def.second->cannotBeAssignedWith(*Elem, &VTypes,
181                 false))
182                 errorExit(L, "contradicting types for type variable [1]");
183             if (!Elem)
184                 Ret.push_back(Def);
185             else
186                 vectorGetRef(Ret, Def.first) =
187                     Elem->withoutVariableTypedefs(VTypes);
188         }
189     }
190
191     return Ret;
192 }
193
194 void ProductTypedef::append(std::shared_ptr<Typedef> T) {
195     this->Typedefs.push_back(T);
196 }

```

```

193 std::shared_ptr<Typedef>
194 ProductTypedef::withoutVariableTypedefs(const VartypeVector &V) {
195     auto Ret = std::make_shared<ProductTypedef>();
196     for (auto &&T : this->Typedefs)
197         Ret->append(T->withoutVariableTypedefs(V));
198     return Ret;
199 }
200
201 static bool vecCanBeAssignWith(const
202     std::vector<std::shared_ptr<Typedef>> &LHS,
203     const
204         std::vector<std::shared_ptr<Typedef>>
205             &RHS,
206         const Typedef::VartypeVector *V,
207         bool ScopedVarIsVar) {
208     auto Size = LHS.size();
209     if (Size != RHS.size())
210         return false;
211     for (size_t I = 0; I < Size; ++I)
212         if (LHS[I]->cannotBeAssignWith(*RHS[I], V, ScopedVarIsVar))
213             return false;
214     return true;
215 }
216
217 bool ProductTypedef::canBeAssignWith(const Typedef &X,
218     const Typedef::VartypeVector
219         *V,
220     bool ScopedVarIsVar) const {
221     auto &Y = getVarTypedef(X, V);
222     if (this->getKind() != Y.getKind())
223         return false;
224     auto YY = static_cast<const ProductTypedef &>(Y);
225     return vecCanBeAssignWith(this->Typedefs, YY.Typedefs, V,
226         ScopedVarIsVar);
227 }
228
229 bool ProductTypedef::hasVariable() const {
230     for (auto &&T : this->Typedefs) {
231         if (T->hasVariable())
232             return true;
233     }
234     return false;
235 }
236
237 static bool vecEquals(const std::vector<std::shared_ptr<Typedef>>
238     &LHS,
239     const std::vector<std::shared_ptr<Typedef>>
240         &RHS) {
241     auto Size = LHS.size();
242     if (Size != RHS.size())
243         return false;
244     for (size_t I = 0; I < Size; ++I)
245         if (*LHS[I] != *RHS[I])
246             return false;
247     return true;
248 }
249
250 bool ProductTypedef::operator==(const Typedef &Y) const {
251     if (this->getKind() != Y.getKind())
252         return false;
253     auto YY = static_cast<const ProductTypedef &>(Y);
254     return vecEquals(this->Typedefs, YY.Typedefs);

```

```

248 }
249
250 bool ProductTypedef::isTypeComplete() const {
251     for (auto &&T : this->Typedefs)
252         if (!T->isTypeComplete())
253             return false;
254     return true;
255 }
256
257 Typedef::Kind ProductTypedef::getKind() const { return
    Typedef::Product; }
258
259 #if 0
260 FunctionTypedef::FunctionTypedef(const FunctionTypedef &Y) {
261     for (auto &&T : Y.Typedefs)
262         this->Typedefs->push_back(std::shared_ptr<Typedef>(T->clone()));
263 }
264 #endif
265
266 FunctionTypedef *FunctionTypedef::clone() const {
267     return new FunctionTypedef(*this);
268 }
269
270 void FunctionTypedef::append(std::shared_ptr<Typedef> T) {
271     if (this->Typedefs.size() < 2) {
272         this->Typedefs.push_back(T);
273     } else if (this->HasCreatedAppendFunc) {
274         assert(this->Typedefs.back()->getKind() == Typedef::Function);
275         auto Back = static_cast<ProductTypedef
            *>(this->Typedefs.back().get());
276         Back->append(T);
277     } else {
278         this->HasCreatedAppendFunc = true;
279         auto AppendFunc = std::make_shared<FunctionTypedef>();
280         auto First = this->Typedefs.back();
281         this->Typedefs.back() = AppendFunc;
282         AppendFunc->append(First);
283         AppendFunc->append(T);
284     }
285 }
286
287 Typedef::Kind FunctionTypedef::getKind() const { return
    Typedef::Function; }
288
289 bool FunctionTypedef::canBeAssignWith(const Typedef &X,
290                                         const Typedef::VartypeVector
291                                             *V,
292                                         bool ScopedVarIsVar) const {
293     auto &Y = getVarTypedef(X, V);
294     if (this->getKind() != Y.getKind())
295         return false;
296     auto YY = static_cast<const ProductTypedef &>(Y);
297
298     std::map<std::string, std::shared_ptr<const Typedef>>
299         VarTypeNameMap;
300
301     auto &LHS = this->getTypedefs();
302     auto &RHS = YY.getTypedefs();
303     auto Size = LHS.size();
304
305     if (Size != RHS.size())
306         return false;

```

```

305
306     for (size_t I = 0; I < Size; ++I) {
307         std::shared_ptr<const Typedef> RightType;
308
309         if (RHS[I]->getKind() == Typedef::Variable) {
310             auto R = static_cast<VariableTypedef *>(RHS[I].get());
311             if (!VarTypenameMap.count(R->getUserTypename())) {
312                 VarTypenameMap[R->getUserTypename()] = LHS[I];
313                 continue;
314             } else {
315                 RightType = VarTypenameMap[R->getUserTypename()];
316             }
317         } else {
318             RightType = RHS[I];
319         }
320
321         if (LHS[I]->cannotBeAssignedWith(*RightType, V, ScopedVarIsVar))
322             return false;
323     }
324
325     return true;
326 }
327
328 std::shared_ptr<Typedef>
329 FunctionTypedef::funApplication(SourceLocation L,
330                                std::shared_ptr<Typedef> T) const {
331     auto &Orig = this->getTypedefs();
332     if (Orig.size() != 2)
333         fatalExit("FunctionTypedef with wrong number of typedefs");
334
335     auto VarTypes = Orig.front()->getVariableTypes(L, T);
336     for (auto &&V : VarTypes) {
337         auto T = vectorGet(this->ApplicationVarTypes, V.first);
338         if (T && *T != *V.second)
339             errorExit(L, "argument has unexpected type [1]");
340         assert(!V.second);
341         if (!T)
342             this->ApplicationVarTypes.push_back(V);
343     }
344     if (Orig.front()->cannotBeAssignedWith(*T,
345         &this->ApplicationVarTypes, false) &&
346         T->cannotBeAssignedWith(*Orig.front(),
347             &this->ApplicationVarTypes, false))
348         errorExit(L, "argument has unexpected type [2]");
349
350     std::shared_ptr<Typedef> Ret =
351         std::shared_ptr<Typedef>(Orig.back()->clone());
352     return Ret->withoutVariableTypedefs(this->ApplicationVarTypes);
353 }
354
355 std::shared_ptr<Typedef>
356 FunctionTypedef::withoutVariableTypedefs(const VartypeVector &V) {
357     auto Ret = std::make_shared<FunctionTypedef>();
358     for (auto &&T : this->Typedefs)
359         Ret->append(T->withoutVariableTypedefs(V));
360     return Ret;
361 }
362
363 #if 0
364 TupleTypedef::TupleTypedef(const FunctionTypedef &Y) {
365     for (auto &&T : Y.Typedefs)
366         this->Typedefs->push_back(std::shared_ptr<Typedef>(T->clone()));
367 }

```

```

364 }
365 #endif
366
367 TupleTypedef *TupleTypedef::clone() const { return new
    TupleTypedef(*this); }
368
369 Typedef::Kind TupleTypedef::getKind() const { return
    Typedef::Tuple; }
370
371 void TupleTypedef::append(std::shared_ptr<Typedef> T) {
372     this->Typedefs.push_back(T);
373 }
374
375 void TupleTypedef::setArgument(SourceLocation L,
    std::shared_ptr<Typedef>) {
376     errorExit(L, "cannot apply type argument here");
377 }
378
379 void TupleTypedef::unmarkInScope() {
380     for (auto T : this->Typedefs)
381         T->unmarkInScope();
382 }
383
384 Typedef::VartypeVector
385 TupleTypedef::getVariableTypes(SourceLocation L,
386     std::shared_ptr<Typedef> Def) const
387 {
388     fatalExit("get variable types from tuple type");
389     return {};
390 }
391
392 bool TupleTypedef::canBeAssignWith(const Typedef &X,
393     const Typedef::VartypeVector *V,
394     bool ScopedVarIsVar) const {
395     auto &Y = getVarTypedef(X, V);
396     if (this->getKind() != Y.getKind())
397         return false;
398     auto YY = static_cast<const TupleTypedef &>(Y);
399     return vecCanBeAssignWith(this->Typedefs, YY.Typedefs, V,
400         ScopedVarIsVar);
401 }
402
403 bool TupleTypedef::hasVariable() const {
404     for (auto &&T : this->Typedefs) {
405         if (T->hasVariable())
406             return true;
407     }
408     return false;
409 }
410
411 bool TupleTypedef::operator==(const Typedef &Y) const {
412     if (this->getKind() != Y.getKind())
413         return false;
414     auto YY = static_cast<const TupleTypedef &>(Y);
415     return vecEquals(this->Typedefs, YY.Typedefs);
416 }
417
418 bool TupleTypedef::isTypeComplete() const {
419     for (auto &&T : this->Typedefs)
420         if (!T->isTypeComplete())
421             return false;
422     return true;

```

```

421 }
422
423 std::shared_ptr<Typedef>
424 TupleTypedef::withoutVariableTypedefs(const VartypeVector &V) {
425     auto Ret = std::make_shared<TupleTypedef>();
426     for (auto &&T : this->Typedefs)
427         Ret->append(T->withoutVariableTypedefs(V));
428     return Ret;
429 }
430
431 bool TupleTypedef::isType() const { return false; }
432
433 #if 0
434 DatatypeTypedef::DatatypeTypedef(const DatatypeTypedef &Y)
435     : Typename(Y.Typename), IsArgumentSet(Y.IsArgumentSet) {
436     for (auto &&P : Y.Params)
437         Params.push_back(P.first,
438             std::shared_ptr<Typedef>(P.second->clone()));
438 }
439 #endif
440
441 DatatypeTypedef::DatatypeTypedef(std::string Name,
442     std::vector<std::string>
443         Paramnames)
444     : Typename(std::move(Name)) {
445     for (auto &&N : Paramnames)
446         Params.push_back({std::move(N), nullptr});
447 }
448
449 DatatypeTypedef::DatatypeTypedef(
450     std::string Name,
451     const std::vector<std::pair<std::string,
452         std::shared_ptr<Typedef>>>
453         &Paramnames)
454     : Typename(std::move(Name)) {
455     for (auto &&N : Paramnames)
456         Params.push_back({N.first, nullptr});
457 }
458
459 DatatypeTypedef *DatatypeTypedef::clone() const {
460     return new DatatypeTypedef(*this);
461 }
462
463 Typedef::Kind DatatypeTypedef::getKind() const { return
464     Typedef::Datatype; }
465
466 void DatatypeTypedef::setArgument(SourceLocation L,
467     std::shared_ptr<Typedef> T) {
468     this->IsArgumentSet = true;
469
470     auto Size = this->Params.size();
471     if (Size == 0)
472         errorExit(L, "type does not take type arguments");
473
474     if (Size == 1) {
475         if (T->getKind() == Typedef::Tuple) {
476             auto TT = static_cast<TupleTypedef*>(T.get());
477             if (TT->getTypedefs().size() != 1)
478                 errorExit(L, "unexpected number of type arguments 2");
479             this->Params.back().second = TT->getTypedefs().front();
480         } else {
481             this->Params.back().second = T;
482         }
483     }
484 }

```

```

479     }
480     return;
481 }
482
483 if (T->getKind() != Typedef::Tuple)
484     errorExit(L, "expected more type argument(s)");
485 auto Types = static_cast<TupleTypedef *>(T.get())->getTypedefs();
486
487 if (Types.size() != Size)
488     errorExit(L, "unexpected number of type arguments 3");
489
490 for (size_t I = 0; I < Size; ++I)
491     this->Params[I].second = Types[I];
492 }
493
494 void DatatypeTypedef::unmarkInScope() {
495     for (auto &P : this->Params) {
496         P.second->unmarkInScope();
497     }
498 }
499
500 std::shared_ptr<Typedef>
501 DatatypeTypedef::withoutVariableTypedefs(const VartypeVector &V) {
502     auto Ret = std::make_shared<DatatypeTypedef>(this->Typename,
503         this->Params);
504     for (auto &&P : this->Params)
505         if (P.second)
506             Ret->setVariableType(P.first,
507                 P.second->withoutVariableTypedefs(V));
508     return Ret;
509 }
510
511 void DatatypeTypedef::setVariableType(const std::string &Name,
512     std::shared_ptr<Typedef> T) {
513     size_t Idx = 0;
514     size_t Size = this->Params.size();
515     for (Idx = 0; Idx < Size; ++Idx)
516         if (Name == this->Params[Idx].first)
517             break;
518     if (Idx >= Size)
519         fatalExit("type variable " + Name + " expected as datatype
520             parameter");
521     this->Params[Idx].second = T;
522 }
523
524 Typedef::VartypeVector
525 DatatypeTypedef::getVariableTypes(SourceLocation L,
526     std::shared_ptr<Typedef> T)
527     const {
528     std::vector<std::pair<std::string, std::shared_ptr<Typedef>>>
529     Ret;
530
531     const DatatypeTypedef *Def = nullptr;
532     if (T->getKind() == Typedef::DatatypeInstance)
533         Def = &static_cast<const DatatypeInstanceTypedef *>(T.get())
534             ->getDatatypedef();
535     else if (T->getKind() == Typedef::Datatype)
536         Def = static_cast<const DatatypeTypedef *>(T.get());
537 #if 0
538     else
539         errorExit(L, "unexpected type");
540 #else

```

```

536     else
537         return {};
538 #endif
539
540     auto Size = this->Params.size();
541     if (Size != Def->Params.size())
542         errorExit(L, "unexpected number of arguments");
543
544     for (size_t I = 0; I < Size; ++I) {
545         auto &T = this->Params[I].second;
546 #if 0
547         if (T->getKind() != Typedef::Variable && Def->Params[I].second
548             &&
549             T->cannotBeAssignWith(*Def->Params[I].second, nullptr,
550                 false))
551             errorExit(L, "invalid argument [2]");
552 #endif
553
554         if (!T)
555             continue;
556         if (!Def->Params[I].second)
557             continue;
558
559         auto M = T->getVariableTypes(L, Def->Params[I].second);
560         for (auto &&Def : M) {
561             auto Elem = vectorGet(Ret, Def.first);
562             if (Elem && Elem->cannotBeAssignWith(*Def.second, nullptr,
563                 false))
564                 errorExit(L, "contradicting types for type variable [2]");
565             if (!Elem && Def.second)
566                 Ret.push_back(Def);
567         }
568     }
569
570     return Ret;
571 }
572
573 bool DatatypeTypedef::canBeAssignWith(const Typedef &X,
574                                         const Typedef::VartypeVector
575                                             *V,
576                                         bool ScopedVarIsVar) const {
577     auto &Y = getVarTypedef(X, V);
578
579     if (Y.getKind() == Typedef::DatatypeInstance) {
580         auto &YY = static_cast<const DatatypeInstanceTypedef &>(Y);
581         return this->canBeAssignWith(YY.getDatatypepedef(), V,
582             ScopedVarIsVar);
583     } else if (Y.getKind() != Typedef::Datatype) {
584         return false;
585     }
586
587     auto &YY = static_cast<const DatatypeTypedef &>(Y);
588     auto Size = this->Params.size();
589     if (Size != YY.Params.size())
590         return false;
591     for (size_t I = 0; I < Size; ++I) {
592 #if 0
593         if (!this->Params[I].second)
594             fatalError("datatype typedef parameter was expected to be
595                 set!");
596 #endif
597     }
598
599     if (!this->Params[I].second || !YY.Params[I].second)

```

```

592         continue;
593     if (YY.Params[I].second->getKind() == Typedef::Variable) {
594         auto Var = static_cast<VariableTypedef>
                    *(YY.Params[I].second.get());
595         if (!Var->isInScope() || ScopedVarIsVar) {
596             continue;
597         }
598     }
599     if
        (this->Params[I].second->cannotBeAssignedWith(*YY.Params[I].second,
        V,
600                                                     ScopedVarIsVar))
601         return false;
602     }
603     return this->Typename == YY.Typename;
604 }
605
606 bool DatatypeTypedef::hasVariable() const {
607     for (auto &&P : this->Params) {
608         if (!P.second || P.second->hasVariable())
609             return true;
610     }
611     return false;
612 }
613
614 bool DatatypeTypedef::operator==(const Typedef &Y) const {
615     if (Y.getKind() == Typedef::DatatypeInstance) {
616         auto &YY = static_cast<const DatatypeInstanceTypedef &>(Y);
617         return *this == YY.getDatatype();
618     } else if (Y.getKind() != Typedef::Datatype) {
619         return false;
620     }
621
622     auto &YY = static_cast<const DatatypeTypedef &>(Y);
623     auto Size = this->Params.size();
624     if (Size != YY.Params.size())
625         return false;
626     for (size_t I = 0; I < Size; ++I) {
627         if (!this->Params[I].second)
628             fatalError("datatype typedef parameter was expected to be
                    set!");
629         if (!YY.Params[I].second)
630             continue;
631         else if (*this->Params[I].second != *YY.Params[I].second)
632             return false;
633     }
634     return this->Typename == YY.Typename;
635 }
636
637 bool DatatypeTypedef::isTypeComplete() const {
638     if (!this->Params.size())
639         return true;
640
641     if (!this->IsArgumentSet)
642         return false;
643
644     for (auto &&P : this->Params)
645         if (!P.second || !P.second->isTypeComplete())
646             return false;
647     return true;
648 }
649

```

```

650 bool DatatypeTypedef::isTypeMissingArgument() const {
651     return this->Params.size() && !this->IsArgumentSet;
652 }
653
654 VariableTypedef *VariableTypedef::clone() const {
655     return new VariableTypedef(*this);
656 }
657
658 VariableTypedef::VariableTypedef(std::string UserName, std::string
    InternName)
659     : UserTypeename(UserName), InternTypeename(InternName) {}
660
661 Typedef::Kind VariableTypedef::getKind() const { return
    Typedef::Variable; }
662
663 void VariableTypedef::setArgument(SourceLocation L,
    std::shared_ptr<Typedef>) {
664     errorExit(L, "variable type does not take type argument(s)");
665 }
666
667 void VariableTypedef::unmarkInScope() { this->IsInScope = false; }
668
669 Typedef::VartypeVector
670 VariableTypedef::getVariableTypes(SourceLocation L,
    std::shared_ptr<Typedef> T)
671     const {
672     return {{this->InternTypeename, T}};
673 }
674
675 bool VariableTypedef::canBeAssignWith(const Typedef &X,
    const Typedef::VartypeVector
676     *V,
677     bool ScopedVarIsVar) const {
678     auto &Y = getVarTypedef(X, V);
679
680     if (this->getKind() != Y.getKind())
681         return false;
682     auto YY = static_cast<const VariableTypedef &>(Y);
683     return this->InternTypeename == YY.InternTypeename;
684 }
685
686 bool VariableTypedef::hasVariable() const {
687     return this->IsInScope ? false : true;
688 }
689
690 std::shared_ptr<Typedef>
691 VariableTypedef::withoutVariableTypedefs(const VartypeVector &V) {
692     auto T = vectorGet(V, this->InternTypeename);
693     if (!T)
694         return std::shared_ptr<Typedef>(this->clone());
695     return T;
696 }
697
698 bool VariableTypedef::operator==(const Typedef &Y) const {
699     return this->canBeAssignWith(Y, nullptr, false);
700 }
701
702 bool VariableTypedef::isTypeComplete() const { return true; }
703
704 #if 0
705 DatatypeInstanceTypedef::DatatypeInstanceTypedef(
706     const DatatypeInstanceTypedef &Y)

```

```

707         : Datatypedef(Y.Datatypedef),
708           ArgumentTypedef(Y.ArgumentTypedef->clone()),
709           Argument(Y.Argument->clone()) {}
710 #endif
711 DatatypeInstanceTypedef::DatatypeInstanceTypedef(
712     const DatatypeTypedef &Datatype, std::shared_ptr<Typedef> Arg)
713     : Datatypedef(Datatype), ArgumentTypedef(Arg) {}
714
715 DatatypeInstanceTypedef *DatatypeInstanceTypedef::clone() const {
716     return new DatatypeInstanceTypedef(*this);
717 }
718
719 Typedef::Kind DatatypeInstanceTypedef::getKind() const {
720     return Typedef::DatatypeInstance;
721 }
722
723 namespace dummy_namespace {
724     size_t vectorCount(
725         const std::vector<std::pair<std::string,
726             std::shared_ptr<Typedef>>> &V,
727         const std::string &S) {
728         size_t Ret = 0;
729         for (auto &&P : V)
730             if (P.first == S)
731                 ++Ret;
732         return Ret;
733     }
734 } // End dummy_namespace namespace.
735
736 namespace dummy_namespace {
737     bool noDuplicates(
738         const std::vector<std::pair<std::string,
739             std::shared_ptr<Typedef>>> &V) {
740         for (auto &&P : V) {
741             if (vectorCount(V, P.first) > 1)
742                 return false;
743         }
744         return true;
745     }
746 } // End dummy_namespace namespace.
747
748 void DatatypeInstanceTypedef::setArgument(SourceLocation L,
749     std::shared_ptr<Typedef>
750     T) {
751     if (!this->ArgumentTypedef)
752         errorExit(L, "does not take arguments");
753     this->Argument.reset(T->clone());
754
755     auto VarTypes = this->ArgumentTypedef->getVariableTypes(L, T);
756
757     if (T->cannotBeAssignWith(*this->ArgumentTypedef, &VarTypes,
758         false))
759         errorExit(L, "argument type mismatch");
760
761     assert(dummy_namespace::noDuplicates(VarTypes));
762     for (auto &&P : VarTypes)
763         this->Datatypedef.setVariableType(P.first, P.second);
764 }
765
766 void DatatypeInstanceTypedef::unmarkInScope() {
767     this->Datatypedef.unmarkInScope();

```

```

764 }
765
766 std::shared_ptr<Typedef>
767 DatatypeInstanceTypedef::funApplication(SourceLocation L,
768                                         std::shared_ptr<Typedef>
769                                         T) const {
770     if (this->Argument)
771         errorExit(L, "cannot apply argument here 2");
772     auto Ret = std::shared_ptr<Typedef>(this->clone());
773     Ret->setArgument(L, T);
774     return Ret;
775 }
776
777 Typedef::VartypeVector
778 DatatypeInstanceTypedef::getVariableTypes(SourceLocation L,
779                                           std::shared_ptr<Typedef>
780                                           T) const {
781     return this->Datatypedef.getVariableTypes(L, T);
782 }
783
784 bool DatatypeInstanceTypedef::canBeAssignWith(const Typedef &X,
785                                               const
786                                               Typedef::VartypeVector
787                                               *V,
788                                               bool ScopedVarIsVar)
789     const {
790     auto &Y = getVarTypedef(X, V);
791     if (Y.getKind() == Typedef::Datatype) {
792         auto YY = static_cast<const DatatypeTypedef &>(Y);
793         return this->Datatypedef.canBeAssignWith(YY, V,
794         ScopedVarIsVar);
795     } else if (Y.getKind() == Typedef::DatatypeInstance) {
796         auto YY = static_cast<const DatatypeInstanceTypedef &>(Y);
797         return this->Datatypedef.canBeAssignWith(YY.Datatypedef, V,
798         ScopedVarIsVar);
799     }
800     return false;
801 }
802
803 bool DatatypeInstanceTypedef::hasVariable() const {
804     return this->Datatypedef.hasVariable();
805 }
806
807 bool DatatypeInstanceTypedef::operator==(const Typedef &Y) const {
808     if (Y.getKind() == Typedef::Datatype) {
809         auto YY = static_cast<const DatatypeTypedef &>(Y);
810         return YY == *this;
811     } else if (Y.getKind() == Typedef::DatatypeInstance) {
812         auto YY = static_cast<const DatatypeInstanceTypedef &>(Y);
813         return this->Datatypedef == YY.Datatypedef;
814     }
815     return false;
816 }
817
818 bool DatatypeInstanceTypedef::isTypeComplete() const {
819     if (this->ArgumentTypedef && !this->Argument)
820         return false;
821     if (this->ArgumentTypedef && !this->Argument->isTypeComplete())
822         return false;
823     auto &Params = this->Datatypedef.getParams();
824     for (auto &&P : Params) {
825         if (!P.second || !P.second->isTypeComplete())

```



```

819         return false;
820     }
821     return true;
822 }
823
824 bool DatatypeInstanceTypedef::isTypeMissingArgument() const {
825     if (!this->ArgumentTypedef)
826         return false;
827     if (this->Argument)
828         return false;
829     return true;
830 }
831
832 std::shared_ptr<Typedef>
833 DatatypeInstanceTypedef::withoutVariableTypedefs(const
834     VartypeVector &V) {
835     std::shared_ptr<DatatypeInstanceTypedef> Ret;
836     if (this->ArgumentTypedef)
837         Ret = std::shared_ptr<DatatypeInstanceTypedef>(new
838             DatatypeInstanceTypedef(
839                 static_cast<const DatatypeTypedef &>(
840                     *this->DatatypeTypedef.withoutVariableTypedefs(V)),
841                 std::shared_ptr<Typedef>(this->ArgumentTypedef->clone())));
842     else
843         Ret = std::shared_ptr<DatatypeInstanceTypedef>(new
844             DatatypeInstanceTypedef(
845                 static_cast<const DatatypeTypedef &>(
846                     *this->DatatypeTypedef.withoutVariableTypedefs(V)),
847                 std::shared_ptr<Typedef>(nullptr)));
848
849     if (this->Argument)
850         Ret->Argument.reset(this->Argument->clone());
851
852     return Ret;
853 }
854
855 std::shared_ptr<DatatypeTypedef> ssml::typecheck::getListTypedef()
856 {
857     return std::make_shared<DatatypeTypedef>("list",
858         std::vector<std::string>{'a'});
859 }
860
861 std::shared_ptr<DatatypeTypedef> ssml::typecheck::getRefTypedef() {
862     return std::make_shared<DatatypeTypedef>("ref",
863         std::vector<std::string>{'a'});
864 }
865
866 std::shared_ptr<DatatypeTypedef>
867 ssml::typecheck::getArrayTypedef() {
868     return std::make_shared<DatatypeTypedef>("array",
869         std::vector<std::string>{'a'});
870 }

```

Listing 82: lib/Typecheck/Patternmatch.cpp

```

1 #include "Patternmatch.h"
2 #include "Typedef.h"
3
4 #include "ssml/Common/FatalExit.h"
5 #include "ssml/Common/ErrorMessage.h"
6

```

```

7  #include "llvm/Support/raw_ostream.h"
8
9  #include <vector>
10
11 using namespace ssml::typecheck;
12
13 Patternmatch::Patternmatch(SourceLocation L) : Location(L) {}
14
15 Patternmatch::~Patternmatch() = default;
16
17 void Patternmatch::matchType(std::shared_ptr<Typedef> T) {
18     if (this->IsTypeMatched && *T != *this->PatternType)
19         errorExit(this->Location, "conflicting pattern type
20                     specifications");
21     this->IsTypeMatched = true;
22     this->PatternType = T;
23 }
24
25 std::shared_ptr<Typedef> Patternmatch::getFreeMatchPatternType() {
26     if (this->IsTypeMatched)
27         return this->PatternType;
28     return nullptr;
29 }
30
31 Typemap::MapType Patternmatch::getNameTypes() {
32     auto Pree = this->getFreeMatchPatternType();
33     if (!Pree || !Pree->isTypeComplete())
34         errorExit(this->Location, "missing type declaration");
35     if (this->IsNameTypesReturned)
36         fatalExit("Patternmatch Name types already returned");
37     this->IsNameTypesReturned = true;
38     return std::move(this->NameTypes);
39 }
40
41 void Patternmatch::appendNameTypes(Typemap::MapType M) {
42     for (auto &&T : M) {
43         if (this->NameTypes.count(T.first))
44             errorExit(this->Location, "multiple declarations of same
45                     name");
46         this->NameTypes[T.first] = T.second;
47     }
48 }
49
50 WildcardPatternmatch::WildcardPatternmatch(SourceLocation L)
51     : Patternmatch(L) {}
52
53 void WildcardPatternmatch::matchType(std::shared_ptr<Typedef> T) {
54     this->Patternmatch::matchType(T);
55 }
56
57 LongIdentifierPatternmatch::LongIdentifierPatternmatch(SourceLocation
58     L,
59     const
60     Typemap
61     &M,
62     std::string
63     &&ID)
64     : Patternmatch(L), ConstnameTypes(M), ID(std::move(ID)) {}
65
66 void
67     LongIdentifierPatternmatch::matchType(std::shared_ptr<Typedef>
68     T) {

```

```

61     this->Patternmatch::matchType(T);
62     auto CT = this->ConstnameTypes.get(this->ID);
63     if (!CT) {
64         this->NameTypes[std::move(this->ID)] = T;
65     } else {
66         if (!CT->isType())
67             fatalExit("unexpected value: does not have a type");
68         if (CT->isTypeMissingArgument())
69             errorExit(this->Location, "incomplete value, missing
              argument");
70         if (T->cannotBeAssignWith(*CT, nullptr, false))
71             errorExit(this->Location, "pattern does not match type");
72     }
73 }
74
75 std::shared_ptr<Typedef>
76     LongIdentifierPatternmatch::getPreeMatchPatternType() {
77     if (this->IsTypeMatched)
78         return this->PatternType;
79
80     auto CT = this->ConstnameTypes.get(this->ID);
81     if (!CT)
82         return nullptr;
83     return CT;
84 }
85
86 TuplePatternmatch::TuplePatternmatch(SourceLocation L) :
87     Patternmatch(L) {}
88
89 void TuplePatternmatch::append(std::shared_ptr<Patternmatch> M) {
90     Matches.push_back(M);
91 }
92
93 void TuplePatternmatch::matchType(std::shared_ptr<Typedef> T) {
94     this->Patternmatch::matchType(T);
95     if (T->getKind() != Typedef::Product)
96         errorExit(this->Location, "pattern does not match type");
97     auto &&Defs = static_cast<ProductTypedef>
98         *(T.get()->getTypedefs());
99     auto Size = this->Matches.size();
100     if (Defs.size() != Size)
101         errorExit(this->Location, "unexpected number of tuple
102             elements");
103     for (size_t I = 0; I < Size; ++I)
104         this->Matches[I]->matchType(Defs[I]);
105 }
106
107 std::shared_ptr<Typedef>
108     TuplePatternmatch::getPreeMatchPatternType() {
109     if (this->IsTypeMatched)
110         return this->PatternType;
111
112     auto Ret = std::make_shared<ProductTypedef>();
113     for (auto &&M : this->Matches) {
114         auto T = M->getPreeMatchPatternType();
115         if (!T)
116             return nullptr;
117         Ret->append(T);
118     }
119     return Ret;
120 }

```

```

117 TuplePatternmatch::getNameTypes() {
118     for (auto &&M : this->Matches)
119         this->appendNameTypes(M->getNameTypes());
120     return this->Patternmatch::getNameTypes();
121 }
122
123 ListPatternmatch::ListPatternmatch(SourceLocation L) :
124     TuplePatternmatch(L) {}
125
126 void ListPatternmatch::matchType(std::shared_ptr<Typedef> T) {
127     this->Patternmatch::matchType(T);
128
129     if (T->cannotBeAssignWith(*getListTypedef(), nullptr, false))
130         errorExit(this->Location, "expected 'list' type specificatin");
131     assert(T->getKind() == Typedef::Datatype);
132
133     auto List = static_cast<DatatypeTypedef &>(*T.get());
134     auto Type = List.getParams().front().second;
135     for (auto &&M : this->Matches)
136         M->matchType(Type);
137 }
138
139 void ListPatternmatch::append(std::shared_ptr<Patternmatch> M) {
140     auto T = M->getPreeMatchPatternType();
141     if (this->Matches.size()) {
142         auto Type = this->Matches.front()->getPreeMatchPatternType();
143         if (Type->cannotBeAssignWith(*T, nullptr, false) &&
144             T->cannotBeAssignWith(*Type, nullptr, false))
145             errorExit(this->Location, "contradicting pattern types in
146                 list pattern");
147     }
148     this->TuplePatternmatch::append(M);
149 }
150
151 std::shared_ptr<Typedef>
152 ListPatternmatch::getPreeMatchPatternType() {
153     if (this->IsTypeMatched)
154         return this->PatternType;
155
156     auto Ret = getListTypedef();
157     if (!this->Matches.size())
158         return Ret;
159
160     std::shared_ptr<Typedef> ListType;
161     for (auto &&M : this->Matches) {
162         auto T = M->getPreeMatchPatternType();
163         if (!T)
164             return nullptr;
165         if (!ListType)
166             ListType = T;
167         else if (ListType->cannotBeAssignWith(*T, nullptr, false) &&
168             T->cannotBeAssignWith(*ListType, nullptr, false))
169             return nullptr;
170         if (!T->isTypeComplete())
171             ListType = T;
172     }
173     Ret->setArgument(this->Location, ListType);
174     return Ret;
175 }
176
177 LiteralPatternmatch::LiteralPatternmatch(SourceLocation L,
178     std::shared_ptr<Typedef>

```

```

176         : Patternmatch(L), LiteralType(T) {}
177
178 void LiteralPatternmatch::matchType(std::shared_ptr<Typedef> T) {
179     this->Patternmatch::matchType(T);
180     if (T->cannotBeAssignWith(*this->LiteralType, nullptr, false))
181         errorExit(this->Location, "pattern does not match");
182 }
183
184 std::shared_ptr<Typedef>
185     LiteralPatternmatch::getPreMatchPatternType() {
186     return this->LiteralType;
187 }
188
189 ApplyPatternmatch::ApplyPatternmatch(SourceLocation L,
190                                     SourceLocation R,
191                                     std::shared_ptr<Patternmatch>
192                                         LHS,
193                                     std::shared_ptr<Patternmatch>
194                                         RHS)
195     : Patternmatch(L), Left(LHS), Right(RHS), RightLocation(R) {}
196
197 static std::shared_ptr<Typedef> vectorGetVariableTypedef(
198     const std::vector<std::pair<std::string,
199         std::shared_ptr<Typedef>>> &V,
200     const std::string &VarName) {
201     for (auto &P : V)
202         if (VarName == P.first)
203             return P.second;
204     ssml::fatalExit("didn't find variable type name " + VarName + "
205         in vector");
206     return nullptr;
207 }
208
209 std::shared_ptr<Typedef> ApplyPatternmatch::match(
210     const std::vector<std::pair<std::string,
211         std::shared_ptr<Typedef>>> &V,
212     std::shared_ptr<Typedef> Match) {
213     switch (Match->getKind()) {
214     case Typedef::SimpleBuildin:
215         return this->simpleBuildinMatch(V, Match);
216     case Typedef::Product:
217         return this->productMatch(V, Match);
218     case Typedef::Function:
219         return this->functionMatch(V, Match);
220     case Typedef::Tuple:
221         fatalExit("unexpected tuple type discovered as type
222             parameter");
223         break;
224     case Typedef::Datatype:
225         return this->datatypeMatch(V, Match);
226     case Typedef::Variable:
227         return this->variableMatch(V, Match);
228     case Typedef::DatatypeInstance:
229         fatalExit("unexpected datatype instance discovered as type
230             parameter");
231     }
232     return nullptr;
233 }
234
235 std::shared_ptr<Typedef> ApplyPatternmatch::simpleBuildinMatch(
236     const std::vector<std::pair<std::string,

```

```

228         std::shared_ptr<Typedef>>> &V,
229         std::shared_ptr<Typedef> Match) {
230     return Match;
231 }
232 std::shared_ptr<Typedef> ApplyPatternmatch::productMatch(
233     const std::vector<std::pair<std::string,
234         std::shared_ptr<Typedef>>> &V,
235     std::shared_ptr<Typedef> Match) {
236     auto &Prod = static_cast<ProductTypedef &>(*Match.get());
237     auto Ret = std::make_shared<ProductTypedef>();
238     auto Defs = Prod.getTypedefs();
239     for (auto &&D : Defs)
240         Ret->append(this->match(V, D));
241     return Ret;
242 }
243 std::shared_ptr<Typedef> ApplyPatternmatch::functionMatch(
244     const std::vector<std::pair<std::string,
245         std::shared_ptr<Typedef>>> &V,
246     std::shared_ptr<Typedef> Match) {
247     auto &Fun = static_cast<FunctionTypedef &>(*Match.get());
248     auto Ret = std::make_shared<FunctionTypedef>();
249     auto Defs = Fun.getTypedefs();
250     for (auto &&D : Defs)
251         Ret->append(this->match(V, D));
252     return Ret;
253 }
254 std::shared_ptr<Typedef> ApplyPatternmatch::datatypeMatch(
255     const std::vector<std::pair<std::string,
256         std::shared_ptr<Typedef>>> &V,
257     std::shared_ptr<Typedef> Match) {
258     auto &Datatype = static_cast<DatatypeTypedef &>(*Match.get());
259     auto &Params = Datatype.getParams();
260     auto Ret = std::shared_ptr<DatatypeTypedef>(
261         new DatatypeTypedef(Datatype.getUserTypeName(),
262             Datatype.getParams()));
263     auto Tup = std::make_shared<TupleTypedef>();
264     for (auto &&P : Params) {
265         if (!P.second)
266             fatalExit("unexpected unused datatype parameter");
267         Tup->append(this->match(V, P.second));
268     }
269     if (Params.size())
270         Ret->setArgument(this->RightLocation, Tup);
271     return Ret;
272 }
273 std::shared_ptr<Typedef> ApplyPatternmatch::variableMatch(
274     const std::vector<std::pair<std::string,
275         std::shared_ptr<Typedef>>> &V,
276     std::shared_ptr<Typedef> Match) {
277     auto &VarName =
278         static_cast<VariableTypedef
279             *>(Match.get())->getInternTypeName();
280     auto Def = vectorGetVariableTypedef(V, VarName);
281     return Def;
282 }
283 void ApplyPatternmatch::matchType(std::shared_ptr<Typedef> T) {
284     this->Patternmatch::matchType(T);

```

```

283     auto L = this->Left->getPreeMatchPatternType();
284     if (!L)
285         errorExit(this->Location, "unknown type name");
286     if (T->cannotBeAssignWith(*L, nullptr, false))
287         errorExit(this->Location, "pattern does not match");
288
289     if (L->getKind() != Typedef::DatatypeInstance)
290         errorExit(this->Location, "does not take argument");
291     auto LHS = static_cast<DatatypeInstanceTypedef *>(L.get());
292
293     auto Arg = LHS->getArgumentTypedef();
294     if (!Arg)
295         errorExit(this->Location, "does not take argument");
296
297     if (!T->getKind() == Typedef::Datatype)
298         fatalExit("expected match type to be datatype typedef");
299     auto TT = static_cast<DatatypeTypedef *>(T.get());
300
301     auto ArgDef = this->match(TT->getParams(),
302                               LHS->getArgumentTypedef());
303     LHS->setArgument(this->RightLocation, ArgDef);
304     if (T->cannotBeAssignWith(*LHS, nullptr, false))
305         errorExit(this->Location, "pattern does not match");
306     this->Right->matchType(ArgDef);
307 }
308
309 Typemap::MapType ApplyPatternmatch::getNameTypes() {
310     this->appendNameTypes(this->Right->getNameTypes());
311     return this->Patternmatch::getNameTypes();
312 }
313
314 std::shared_ptr<Typedef>
315 ApplyPatternmatch::getPreeMatchPatternType() {
316     if (this->IsTypeMatched)
317         return this->PatternType;
318
319     assert(!this->Left);
320     auto Type = this->Left->getPreeMatchPatternType();
321     if (!Type)
322         return nullptr;
323
324     auto Ret =
325         std::shared_ptr<Typedef>(Type->clone());
326     auto R = this->Right->getPreeMatchPatternType();
327     if (R)
328         Ret->setArgument(this->RightLocation, R);
329     return Ret;
330 }

```

Listing 83: lib/Typecheck/Typemap.cpp

```

1  #include "Typemap.h"
2
3  #include "Typedef.h"
4
5  using namespace ssml::typecheck;
6
7  void Typemap::insert(std::string K, ValueType V, size_t Scope) {
8      Maps[Maps.size() - 1 - Scope][std::move(K)] = V;
9  }

```

```
10
11 auto Typemap::get(const std::string &K) const -> ValueType {
12     for (auto It = Maps.rbegin(), End = Maps.rend(); It != End; ++It)
13         if (It->count(K))
14             return ValueType(It->find(K)->second->clone());
15     return nullptr;
16 }
17
18 void Typemap::mergeWith(MapType M, size_t Scope) {
19     for (auto &&T : M)
20         this->insert(std::move(T.first), T.second, Scope);
21 }
22
23 void Typemap::enterScope() {
24     Maps.push_back(MapType());
25 }
26
27 void Typemap::leaveScope() {
28     Maps.pop_back();
29 }
```