# Garbage Collection With LLVM

Andreas Lynge

January 25, 2016

# Contents

# 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 bitcode file. By further passing a number of bitcode files to LLVM; it can link them together into one executable while performing link-time optimization, or it can execute the bitcode 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 bitcode 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`:

```
%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:

```
define i32 @fib(i32 %n) {
  ; Signed compare whether %n is less than 2, and
  ; store boolean (1 bit integer) result in %1:
  %1 = icmp slt i32 %n, 2
  ; Branch to label Done if %n < 2, otherwise branch to Recurse:
  br i1 %1, label %Done, label %Recurse
; Label Recurse:
Recurse:
  ; Subtract 1 from %n and store result in %3:
  %3 = sub i32 %n, 1
  ; Recursive call fib(%n - 1) and store result in %4:
  %4 = tail call i32 @fib(i32 %3)
  ; Store %n-2 in %5:
  %5 = sub i32 %n, 2
  ; Store call fib(%n - 2) in %6:
  %6 = tail call i32 @fib(i32 %5)
  ; Store result of fib(%n-1) + fib(%n-2) in %7:
  %7 = add i32 %4, %6
  ; Return result in %7:
  ret i32 %7
; Label Done:
Done:
  ; Base case, returns 0 or 1:
  ret i32 %n
}
```

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.

| | | |
|---|---|---|
| ⟨*start*⟩ | → | ⟨*decl-seq*⟩ |
| ⟨*decl-seq*⟩ | → | ⟨*decl-seq*⟩ ⟨*scolon-opt*⟩ ⟨*decl*⟩ |
| | \| | ⟨*decl*⟩ |
| ⟨*decl*⟩ | → | **val** ⟨*val-decl*⟩ |
| | \| | **fun** ⟨*fun-decl*⟩ |
| | \| | **datatype** ⟨*datatype-decl*⟩ |
| | \| | **nonfix** ⟨*id-seq*⟩ |
| | \| | **infix** ⟨*int-lit-opt*⟩ ⟨*id-seq*⟩ |
| | \| | **infixr** ⟨*int-lit-opt*⟩ ⟨*id-seq*⟩ |
| ⟨*val-decl*⟩ | → | ⟨*pattern*⟩ = ⟨*expr*⟩ |
| ⟨*fun-decl*⟩ | → | ⟨*fun-decl*⟩ **\|** ⟨*val-decl*⟩ |
| | \| | ⟨*val-decl*⟩ |
| ⟨*datatype-decl*⟩ | → | ⟨*type-var-tuple-opt*⟩ ⟨*type-id*⟩ = ⟨*datatype-def*⟩ |
| ⟨*datatype-def*⟩ | → | ⟨*datatype-def*⟩ **\|** ⟨*id*⟩ ⟨*of-type-opt*⟩ |
| | \| | ⟨*id*⟩ ⟨*of-type-opt*⟩ |
| ⟨*of-type-opt*⟩ | → | **of** ⟨*type*⟩ |
| | \| | $\epsilon$ |
| ⟨*type-var-tuple-opt*⟩ | → | ⟨*type-var-tuple*⟩ |
| | \| | $\epsilon$ |
| ⟨*type-var-tuple*⟩ | → | **typevar** |
| | \| | **(** ⟨*type-var-comma-seq*⟩ **)** |
| ⟨*type-var-comma-seq*⟩ | → | ⟨*type-var-comma-seq*⟩ **, typevar** |
| | \| | **typevar** |
| ⟨*pattern*⟩ | → | ⟨*pattern*⟩ **:** ⟨*type*⟩ |
| | \| | ⟨*apply-pattern*⟩ |
| ⟨*apply-pattern*⟩ | → | ⟨*apply-pattern*⟩ ⟨*root-pattern*⟩ |
| | \| | ⟨*root-pattern*⟩ |
| ⟨*root-pattern*⟩ | → | **intlit** |
| | \| | _ |
| | \| | ⟨*long-id*⟩ |
| | \| | **op** ⟨*long-id*⟩ |
| | \| | **(** ⟨*pattern-comma-seq-opt*⟩ **)** |
| | \| | **[** ⟨*pattern-comma-seq-opt*⟩ **]** |
| ⟨*pattern-comma-seq-opt*⟩ | → | ⟨*pattern-comma-seq*⟩ |
| | \| | $\epsilon$ |
| ⟨*pattern-comma-seq*⟩ | → | ⟨*pattern-comma-seq*⟩ **,** ⟨*pattern*⟩ |
| | \| | ⟨*pattern*⟩ |
| ⟨*expr*⟩ | → | **let** ⟨*decl-seq*⟩ **in** ⟨*expr-scolon-seq*⟩ **end** |
| | \| | **if** ⟨*expr*⟩ **then** ⟨*expr*⟩ **else** ⟨*expr*⟩ |

| **while** ⟨*expr*⟩ **do** ⟨*expr*⟩
| **fn** ⟨*match-seq*⟩
| ⟨*expr*⟩ **andalso** ⟨*expr*⟩
| ⟨*expr*⟩ **orelse** ⟨*expr*⟩
| ⟨*apply-expr*⟩

⟨*apply-expr*⟩ → ⟨*apply-expr*⟩ ⟨*root-expr*⟩
| ⟨*root-expr*⟩

⟨*root-expr*⟩ → **intlit**
| ⟨*long-id*⟩
| **op** ⟨*long-id*⟩
| **(** ⟨*expr-comma-seq-opt*⟩ **)**
| **[** ⟨*expr-comma-seq-opt*⟩ **]**

⟨*expr-scolon-seq*⟩ → ⟨*expr-scolon-seq*⟩ **;** ⟨*expr*⟩
| ⟨*expr*⟩

⟨*expr-comma-seq-opt*⟩ → ⟨*expr-comma-seq*⟩
| $\epsilon$

⟨*expr-comma-seq*⟩ → ⟨*expr-comma-seq*⟩ **,** ⟨*expr*⟩
| ⟨*expr*⟩

⟨*match-seq*⟩ → ⟨*match-seq*⟩ **|** ⟨*match*⟩
| ⟨*match*⟩

⟨*match*⟩ → ⟨*pattern*⟩ => ⟨*expr*⟩

⟨*type*⟩ → ⟨*fun-type-2*⟩
| ⟨*prod-type-2*⟩
| ⟨*apply-type*⟩

⟨*fun-type-2*⟩ → ⟨*fun-type*⟩ -> ⟨*apply-type*⟩
| ⟨*fun-type*⟩ -> ⟨*prod-type-2*⟩

⟨*fun-type*⟩ → ⟨*fun-type*⟩ -> ⟨*apply-type*⟩
| ⟨*fun-type*⟩ -> ⟨*prod-type-2*⟩
| ⟨*apply-type*⟩
| ⟨*prod-type-2*⟩

⟨*prod-type-2*⟩ → ⟨*prod-type*⟩ * ⟨*apply-type*⟩

⟨*prod-type*⟩ → ⟨*prod-type*⟩ * ⟨*apply-type*⟩
| ⟨*apply-type*⟩

⟨*apply-type*⟩ → ⟨*apply-type*⟩ ⟨*root-type*⟩
| ⟨*root-type*⟩

⟨*root-type*⟩ → ⟨*long-type-id*⟩
| **typevar**
| **(** ⟨*type-comma-seq*⟩ **)**

⟨*type-comma-seq*⟩ → ⟨*type-comma-seq*⟩ **,** ⟨*type*⟩
| ⟨*type*⟩

$\langle \textit{scolon-opt} \rangle \rightarrow$ **;**
$\quad | \ \epsilon$

$\langle \textit{id-seq} \rangle \rightarrow \langle \textit{id-seq} \rangle\ \langle \textit{id} \rangle$
$\quad | \ \langle \textit{id} \rangle$

$\langle \textit{long-id} \rangle \rightarrow \langle \textit{long-id} \rangle$ **.** $\langle \textit{id} \rangle$
$\quad | \ \langle \textit{id} \rangle$

$\langle \textit{id} \rangle \rightarrow \langle \textit{type-id} \rangle$
$\quad |$ *****

$\langle \textit{long-type-id} \rangle \rightarrow \langle \textit{long-type-id} \rangle$ **.** $\langle \textit{type-id} \rangle$
$\quad | \ \langle \textit{type-id} \rangle$

$\langle \textit{type-id} \rangle \rightarrow$ **identifier**
$\quad |$ **operator**
$\quad |$ **=**

$\langle \textit{int-lit-opt} \rangle \rightarrow$ **intlit**
$\quad | \ \epsilon$

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:

```
fun factorial 0       : int = 1
  | factorial(n:int) : int = let
    fun run() : int = n * factorial(n-1)
  in
    run()
  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:

```
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:

```
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`:

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

And in the following case the compiler does not:

```
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:

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

Or equivalently:

```
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:

```
fun len []               : int = 0
  | 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:

```
fun len([]    : 'a list): int = 0
  | 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:

```
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:

```
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:

```
fun f(xs: 'a list): 'a list ref =
let
  val ys: 'a list ref = ref xs
in
  ys
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 + 2*3
```

into the form:

```
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:

```
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 `int`s.

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:

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

the compiler produces the error message:

```
<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:

```
<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:

```
struct function_value = {
  void **parent_frame;
  void *(*value)(void **frame, void *arg);
};
```

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:

```
struct function_value *fun = frame[X];
void *arg = ...;
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:

```
val 1 = 2
```

Another example of a failing pattern match is:

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

When generating code for a curried function like:

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

```
struct function_value *test_1(void **parent_frame, void *arg) {
  void **shared_frame = malloc(...);
  shared_frame[0] = parent_frame;
  shared_frame[1] = arg;
  struct function_value *fun = malloc(...);
  fun->parent_frame = shared_frame;
  fun->value = &test_2;
```

```
  return fun;
}
```

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:

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

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:

```
void *test_3(void **shared_frame, void *arg) {
  if ((long int)shared_frame[1] == 1 &&
      (long int)shared_frame[2] == 2 &&
      (long int)arg             == 3)
    return (void *)1;
  if ((long int)shared_frame[1] == 2 &&
      (long int)shared_frame[2] == 3 &&
      (long int)arg             == 4)
    return (void *)3;
  fprintf(stderr, "pattern match failed");
  exit(1);
}
```

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:

```
val a: int*int -> int = op+
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:

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

In the following example:

```
val a:ordering = LT
val b:ordering = EQ
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:

```
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:

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

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

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

In this case when defining the values:

```
val n:intopt = NoneInt
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:

```
val SomeInt 10 = s
```

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

```
if (s[0] != 1) {
  fprintf(stderr, "pattern match failed");
  exit(1);
}
if (s[1] != 10) {
  fprintf(stderr, "pattern match failed");
  exit(1);
}
```

And for the following `val` statement:

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

code like the following will be generated:

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

It is also possible to create parameterized types:

```
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:

```
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 is 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:

```
datatype nat = Zero | Succ of nat

fun maxImpl Zero        (_:nat)      (_:nat,y:nat) : nat = y
  | maxImpl(_:nat)       Zero        (x:nat,_:nat) : nat = x
  | maxImpl(Succ n: nat)(Succ m: nat)(xy: nat*nat) : nat =
      maxImpl n m xy

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.

```
define i64 @entry() ... {
  ...
  ; Allocate stack memory for pointer to @entry's frame:
  %locals = alloca i64*, align 8
  ...
  ; Dynamically allocate memory for @entry's frame:
  %1 = tail call i64* @allocate(i64 24)
  ; Store pointer to frame on the stack:
  store i64* %1, i64** %locals, align 8
  ...
  ; Allocate memory for function value for @_1maxImpl:
  %2 = tail call i64* @allocate(i64 16)
  ; Pointer to function value converted to i64:
  %3 = ptrtoint i64* %2 to i64
  ...
  ; Store pointer to @_maxImpl at index 0 in the function value:
  store i64 ptrtoint (i64 (i64, i64)* @_1maxImpl to i64), i64* %2,
      align 8
  ; Get pointer to index 1 of the function value:
  %4 = getelementptr i64, i64* %2, i64 1
  ; Get @entry's frame from the stack and store it in %frame11:
  %5 = bitcast i64** %locals to i64*
  %frame11 = load i64, i64* %5, align 8
  ; Store @entry's frame at index 1 in the function value:
  store i64 %frame11, i64* %4, align 8
  ; Get @entry's frame:
  %frame2 = load i64*, i64** %locals, align 8
  ; Get pointer to index 1 of @entry's frame:
  %6 = getelementptr i64, i64* %frame2, i64 1
  ; Store function value in @entry's frame:
  store i64 %3, i64* %6, align 8
  ; Allocate memory for function value for @_4max:
  %7 = tail call i64* @allocate(i64 16)
  ; Initialize and store function value for @_4max is analogous
  ; to how it was done for @_1maxImpl, thus it is left out.
```

```
  ...
  ; Return 0 to indicate execution was a success:
  ret i64 0
}
```

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`:

```
define i64 @_1maxImpl(i64, i64) ... {
  ; Allocate stack memory for temporary storage:
  %tempptr1 = alloca i64*, align 8
  ; Convert it from i64** to i64* for a technical reason
  ; related to GC:
  %temp2 = bitcast i64** %tempptr1 to i64*
  ...
  ; Allocate stack memory for storing frame shared among
  ; the maxImpl functions (shared frame):
  %locals = alloca i64*, align 8
  ; Store second function argument. This is the argument
  ; which is used for partially applying maxImpl:
  store i64 %1, i64* %temp2, align 8
  ...
  ; Dynamically allocate shared frame:
  %4 = tail call i64* @allocate(i64 24)
  ; Store shared frame on the stack:
  store i64* %4, i64** %locals, align 8
  ; Store first function argument (parent
  ; function's frame) in shared frame:
  store i64 %0, i64* %4, align 8
  ; Get second function argument from stack:
  %5 = load i64, i64* %temp2, align 8
  ; Get shared frame from stack:
  %frame3 = load i64*, i64** %locals, align 8
  ; Get pointer to index 1 of shared frame:
  %6 = getelementptr i64, i64* %frame3, i64 1
  ; Store second function argument at index 1 of shared frame:
  store i64 %5, i64* %6, align 8
  ; Allocate memory for function value for maxImpl
  ; partially applied 1 argument:
  %7 = tail call i64* @allocate(i64 16)
  ; Store pointer to @_2maxImpl (partially applied maxImpl)
  ; at index 0 of function value:
  store i64 ptrtoint (i64 (i64, i64)* @_2maxImpl to i64), i64* %7,
      align 8
  ; Get pointer to index 1 of function value:
  %8 = getelementptr i64, i64* %7, i64 1
  ; Get shared frame from stack and store it in %frame41:
  %9 = bitcast i64** %locals to i64*
  %frame41 = load i64, i64* %9, align 8
  ; Store shared frame in function value:
  store i64 %frame41, i64* %8, align 8
  ; Convert to function value to i64:
  %10 = ptrtoint i64* %7 to i64
  ; Return function value of partially applied maxImpl
  ; (@_2maxImpl):
  ret i64 %10
}

define i64 @_2maxImpl(i64, i64) ... {
```

```
  ...
  ; Allocate stack memory for shared frame:
  %locals = alloca i64*, align 8
  ...
  ; Get shared frame (first argument):
  %.cast = inttoptr i64 %0 to i64*
  ; Store shared frame on stack:
  store i64* %.cast, i64** %locals, align 8
  ; Get pointer to index 2 of shared frame:
  %3 = getelementptr i64, i64* %.cast, i64 2
  ; Store second argument at index 2 in shared frame:
  store i64 %1, i64* %3, align 8
  ; Allocate memory for function value for maxImpl
  ; partially applied 2 arguments:
  %4 = tail call i64* @allocate(i64 16)
  ; Store pointer to @_3maxImpl at index 0 of function value:
  store i64 ptrtoint (i64 (i64, i64)* @_3maxImpl to i64), i64* %4,
      align 8
  ; Get pointer to index 1 of function value:
  %5 = getelementptr i64, i64* %4, i64 1
  ; Get shared frame from stack and store it in %frame31:
  %6 = bitcast i64** %locals to i64*
  %frame31 = load i64, i64* %6, align 8
  ; Store shared frame at index 1 in function value:
  store i64 %frame31, i64* %5, align 8
  %7 = ptrtoint i64* %4 to i64
  ; Return function value:
  ret i64 %7
}

define i64 @_3maxImpl(i64, i64) ... {
  ; Allocate stack memory:
  %tempptr1 = alloca i64*, align 8
  ...
  %temp2 = bitcast i64** %tempptr1 to i64*
  ...
  ; Allocate stack memory for frame:
  %locals = alloca i64*, align 8
  ...
  ; Store second argument passed to this function
  ; on the stack:
  store i64 %1, i64* %temp2, align 8
  ...
  ; Dynamically allocate memory for frame:
  %11 = tail call i64* @allocate(i64 48)
  ; Store pointer to frame on the stack:
  store i64* %11, i64** %locals, align 8
  ; Store shared frame at index 0 in frame:
  store i64 %0, i64* %11, align 8
  ; Get frame:
  %frame3 = load i64*, i64** %locals, align 8
  %12 = bitcast i64* %frame3 to i64**
  ; Get shared frame:
  %13 = load i64*, i64** %12, align 8
  ; Get pointer to index 1 of shared frame:
  %14 = getelementptr i64, i64* %13, i64 1
  %15 = bitcast i64* %14 to i64**
  ; Get the argument which was passed to @_1maxImpl from
  ; index 1 of shared frame:
  %16 = load i64*, i64** %15, align 8
  ; Get first word of that argument:
  %17 = load i64, i64* %16, align 8
```

```
  ...
  ; Compare whether the first word of the argument is Zero or Succ:
  %19 = icmp eq i64 %18, 0
  ; Branch to label %20 if the argument which was passed
  ; to @_1maxImpl is Zero, otherwise branch to label %27:
  br i1 %19, label %20, label %27

; <label>:20
  ; Argument passed to @_1maxImpl was Zero.
  ; Thus, in this branch the index 1 from the tuple
  ; which was passed to this function is returned.
  ; Get tuple which was passed to this function (as
  ; second argument):
  %21 = load i64*, i64** %tempptr1, align 8
  %22 = getelementptr i64, i64* %21, i64 1
  %23 = load i64, i64* %22, align 8
  %24 = getelementptr i64, i64* %frame3, i64 1
  store i64 %23, i64* %24, align 8
  %frame6 = load i64*, i64** %locals, align 8
  %25 = getelementptr i64, i64* %frame6, i64 1
  %26 = load i64, i64* %25, align 8
  ret i64 %26

; <label>:27
  ; Argument passed to @_1maxImpl was Succ.
  ; Get pointer to index 2 of shared frame:
  %28 = getelementptr i64, i64* %13, i64 2
  %29 = bitcast i64* %28 to i64**
  ; Get argument which was passed to @_2maxImpl:
  %30 = load i64*, i64** %29, align 8
  ...
  ; Branch to %33 if argument which was passed to @_2maxImpl
  ; is Zero, else it is Succ, then branch to %39:
  br i1 %32, label %33, label %39

; <label>:33
  ; Argument passed to @_2maxImpl was Zero.
  ; Get tuple which was passed to @_2maxImpl function
  ; and return index 0 from that tuple:
  %34 = load i64*, i64** %tempptr1, align 8
  %35 = load i64, i64* %34, align 8
  %36 = getelementptr i64, i64* %frame3, i64 2
  store i64 %35, i64* %36, align 8
  %frame10 = load i64*, i64** %locals, align 8
  %37 = getelementptr i64, i64* %frame10, i64 2
  %38 = load i64, i64* %37, align 8
  ret i64 %38

; <label>:39
  ; Arguments passed to @_1maxImpl and @_2maxImpl were both Succ.
  ; Verify the first word of the argument passed to
  ; @_1maxImpl is 1:
  %40 = icmp eq i64 %18, 1
  ; If it is 1 then branch to %41, otherwise crash program:
  br i1 %40, label %41, label %fun_match_fail

; <label>:41
  ; Get index 1 of the argument which was passed to @_1maxImpl:
  %42 = getelementptr i64, i64* %16, i64 1
  %43 = load i64, i64* %42, align 8
  %44 = getelementptr i64, i64* %frame3, i64 3
  ; Store index 1 from the argument passed to @_1maxImpl in frame:
```

```
  store i64 %43, i64* %44, align 8
  %frame13 = load i64*, i64** %locals, align 8
  %45 = bitcast i64* %frame13 to i64**
  ; Get shared frame from frame:
  %46 = load i64*, i64** %45, align 8
  %47 = getelementptr i64, i64* %46, i64 2
  %48 = bitcast i64* %47 to i64**
  ; Get word at index 2 from shared frame, that is the argument
  ; which was passed to @_2maxImpl:
  %49 = load i64*, i64** %48, align 8
  ; Get first word of that:
  %50 = load i64, i64* %49, align 8
  ...
  ; Branch to %52 if the argument which was passed to
  ; @_2maxImpl was a Succ, otherwise crash the program:
  br i1 %51, label %52, label %fun_match_fail

; <label>:52
  ; Arguments passed to @_1maxImpl and @_2maxImpl are both Succ.
  %53 = getelementptr i64, i64* %49, i64 1
  %54 = load i64, i64* %53, align 8
  %55 = getelementptr i64, i64* %frame13, i64 4
  ; Store index 1 from the argument passed to @_2maxImpl in frame
  store i64 %54, i64* %55, align 8
  %56 = load i64, i64* %temp2, align 8
  %frame15 = load i64*, i64** %locals, align 8
  %57 = getelementptr i64, i64* %frame15, i64 5
  store i64 %56, i64* %57, align 8
  ; Allocate memory for function value for recursive call:
  %58 = tail call i64* @allocate(i64 16)
  %59 = ptrtoint i64* %58 to i64
  store i64 %59, i64* %temp17, align 8
  %60 = bitcast i64** %locals to i64***
  %frame1812345 = load i64**, i64*** %60, align 8
  %61 = load i64*, i64** %frame1812345, align 8
  %62 = load i64, i64* %61, align 8
  ; Store function pointer at index 0 in function value:
  store i64 ptrtoint (i64 (i64, i64)* @_1maxImpl to i64), i64*
      %58, align 8
  %63 = getelementptr i64, i64* %58, i64 1
  ; Store shared frame at index 1 in function value:
  store i64 %62, i64* %63, align 8
  %frame19 = load i64*, i64** %locals, align 8
  %64 = getelementptr i64, i64* %frame19, i64 3
  %65 = load i64, i64* %64, align 8
  store i64 %65, i64* %temp21, align 8
  ; Get function frame:
  %66 = load i64*, i64** %tempptr16, align 8
  %67 = bitcast i64* %66 to i64 (i64, i64)**
  %68 = load i64 (i64, i64)*, i64 (i64, i64)** %67, align 8
  %69 = getelementptr i64, i64* %66, i64 1
  %70 = load i64, i64* %69, align 8
  ; Call @_1maxImpl:
  %71 = tail call i64 %68(i64 %70, i64 %65)
  ; Returns function value, which we store on the stack:
  store i64 %71, i64* %temp23, align 8
  %72 = getelementptr i64, i64* %frame19, i64 4
  %73 = load i64, i64* %72, align 8
  store i64 %73, i64* %temp26, align 8
  %.cast = inttoptr i64 %71 to i64*
  %74 = inttoptr i64 %71 to i64 (i64, i64)**
  %75 = load i64 (i64, i64)*, i64 (i64, i64)** %74, align 8
```

```
  %76 = getelementptr i64, i64* %.cast, i64 1
  %77 = load i64, i64* %76, align 8
  ; Call @_2maxImpl:
  %78 = tail call i64 %75(i64 %77, i64 %73)
  ; Store returned function value on stack:
  store i64 %78, i64* %temp28, align 8
  %79 = getelementptr i64, i64* %frame19, i64 5
  %80 = load i64, i64* %79, align 8
  store i64 %80, i64* %temp31, align 8
  %.cast6 = inttoptr i64 %78 to i64*
  %81 = inttoptr i64 %78 to i64 (i64, i64)**
  %82 = load i64 (i64, i64)*, i64 (i64, i64)** %81, align 8
  %83 = getelementptr i64, i64* %.cast6, i64 1
  %84 = load i64, i64* %83, align 8
  ; Recursively call @_3maxImpl:
  %85 = tail call i64 %82(i64 %84, i64 %80)
  ret i64 %85

fun_match_fail:
  ; Should never get here.
  ; This function call will crash the program with the
  ; "pattern match failed" error message:
  tail call void @matchError()
  unreachable
}
```

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:

```
val shadow: int = 1

fun first(f1:int)(f2:int): unit =
  let
    val f: int = 20
    fun second(s1:int)(s2:int): unit =
```

```
      if s1 = f1 andalso s2 = f2
      then print(f)
      else print(~1)

    val f: int = 30
    fun third(t:int): unit =
      if t = f
      then print(shadow)
      else print(~1)

    val shadow: int = 2
  in
    second f1 f2;
    third f;
    print shadow
  end

val () = (first 3 4;
          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:

```
20
1
2
1
```

## 8.2 Common Functions

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

```
(* List we will use as input for functions *)
val input: int list = [1,2,3,4,5]

(* Define function composition *)
infixr 3 o;
fun (f: 'b -> 'c) o (g: 'a -> 'b) : 'a -> 'c = fn (x:'a) => f (g x)

fun even(x:int) : bool = x mod 2 = 0
val odd:   int -> bool = not o even

fun map(_: 'a -> 'b)([]     : 'a list): 'b list = []
  | map(f: 'a -> 'b)(x::xs : 'a list): 'b list = f x :: map f xs

fun filter(f: 'a -> bool)([]     : 'a list): 'a list = []
  | filter(f: 'a -> bool)(x::xs : 'a list): 'a list = let
    val tail: 'a list = filter f xs
  in
    if f x then x::tail else tail
  end

(* Map function partially applied with negation op~ *)
```

```
val negmap: int list -> int list = map op~

(* Results of using the functions *)
val duped: int list = map (fn x:int => x*2) input
val neged: int list = negmap input
val evens: int list = filter even input
val odds : int list = filter odd input

(* Print results *)
fun printList([]     : 'a list): unit = ()
  | printList(x::xs  : 'a list): unit = (print x; printList xs)
val () = (printList duped;
          printList neged;
          printList evens;
          printList odds)
```

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

```
2
4
6
8
10
-1
-2
-3
-4
-5
2
4
1
3
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:

```
(* Ordering for comparing elements in a set *)
datatype ordering = LT | EQ | GT

(* A node in a set is a leaf or a node with 2 children *)
datatype 'a setnode = Leaf
                    | Node of 'a * 'a setnode * 'a setnode

(* Set contains comparison function and root node. *)
datatype 'a set = Set of ('a*'a -> ordering) * 'a setnode

(* Get an empty set by supplying a comparison function. *)
fun empty(cmp: 'a*'a -> ordering): 'a set = Set (cmp, Leaf)

```

```
(* insert element in set. *)
fun insert(x:'a)(Set(cmp, node): 'a set): 'a set =
  let
    infix cmp
    fun ins(Leaf: 'a setnode): 'a setnode =
        Node (x, Leaf, Leaf)
      | ins(Node(y, left, right): 'a setnode): 'a setnode =
        (fn  LT              => Node(y, ins left, right)
          |  EQ              => Node(y, left, right)
          | (_ : ordering) => Node(y, left, ins right)) (x cmp y)
  in
    Set(op cmp, ins node)
  end

(* Convert list into set. *)
fun fromList(cmp: 'a*'a -> ordering)([]    : 'a list): 'a set =
    empty cmp
  | fromList(cmp: 'a*'a -> ordering)(x::xs: 'a list): 'a set =
    insert x (fromList cmp xs)

(* Apply function f to each element in set (in-order). *)
fun app(f: 'a -> 'b)(Set(_, node): 'a set): unit =
  let
    fun inord(Leaf                    : 'a setnode): unit = ()
      | inord(Node (x, left, right): 'a setnode): unit = (
        inord left;
        f x;
        inord right)
  in
    inord node
  end

(* An integer set for testing *)
val set: int set =
  let
    fun cmp(x:int,y:int): ordering = if x < y then      LT
                                     else if x = y then EQ
                                     else               GT
  in
    fromList cmp [5, 3, 1, 6, ~3, 1]
  end

(* Print values of an in-order traversal. *)
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:

```
-3
1
3
5
6
```

## 8.4 Foldl with Ref

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

```
fun null([]: 'a list): bool = true
  | null(_ : 'a list): bool = false

fun hd(x::_ : 'a list): 'a = x
fun tl(_::xs: 'a list): 'a list = xs

fun foldl(f: 'a * 'b -> 'b)(i: 'b)(xs: 'a list): 'b =
  let
    val acc : 'b ref = ref i
    val rest: 'a list ref = ref xs
  in
    while not (null (!rest)) do
      let
        val head: 'a = hd (!rest)
      in
        rest := tl(!rest);
        acc  := f(head, !acc)
      end;
    !acc
  end

val sum: int list -> int = foldl op+ 0

val () = print (sum [1,2,3])
```

Executing that program yields the output:

```
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`:

```
fun foldl(f: 'a * 'b -> 'b)(i: 'b)(a: 'a array): 'b =
  let
    val n   : int     = Array.length a
    val idx: int ref = ref 0
    val acc: 'b  ref = ref i
  in
    while !idx < n do (
      acc := f(Array.get(a, !idx), !acc);
      idx := !idx + 1
    );
    !acc
  end

val sum: int array -> int = foldl op+ 0

val a: int array = Array.array (3, 0)
val () = (Array.update (a, 0, 1);
          Array.update (a, 1, 2);
          Array.update (a, 2, 3))
```

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

When executing the program it outputs the expected:

```
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:

```
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:

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

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:

```
stackptr[0] = (int64_t)llvm_gc_root_chain;
stackptr[1] = stack_frame_descriptor;
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:

```
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 `ssmlgcmap`. 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):

```
ssmlgcmap_entry_1: # Entry of 1st function in program.
  .short L         # Stack frame size.
```

```
  .short M          # Number of stack roots.
  .short I_1        # Stack index of 1st stack root.
  .short I_2        # Stack index of 2nd stack root.
  ...
  .short I_M        # Stack index of Mth stack root.
ssmlgcmap_entry_2: # Entry for 2nd function in program.
  ...
...
ssmlgcmap_entry_N: # Entry for Nth function in program.
  ...
```

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:

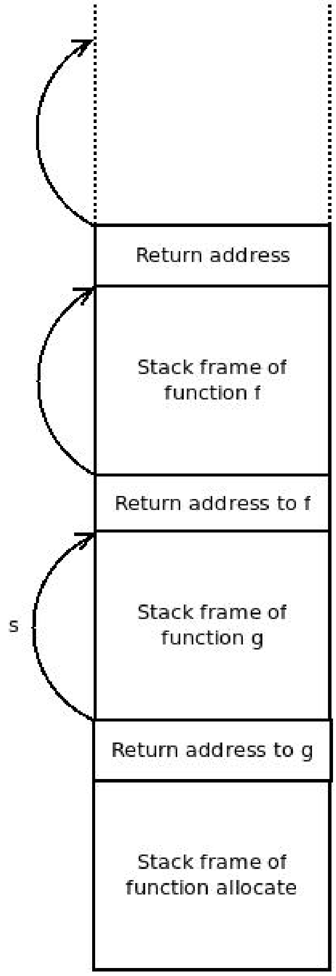
```c
struct lookup_entry {
  void *fun_addr;
  void *ssmlgc_entry;
}

const uint64_t ssml_gclookup_size = N; // Number of functions
const struct lookup_entry ssml_gclookup_map[N+1] = {
  {&user_fun_1, &ssmlgcmap_entry_1},
  {&user_fun_2, &ssmlgcmap_entry_2},
  ...
  {&user_fun_N, &ssmlgcmap_entry_N},
  {&code_end, &0}
}
```

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 `ssmlgcmap_entry_k` entry is stored. However, if implementing this, then given stack frame pointer `f` we know that address of the `ssmlgcmap_entry_k` entry is located at `f[c]` for some constant index `c`. This solution would be almost free, since address of `ssmlgcmap_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 `ssmlgcmap_entry_k` entry as argument when calling a function such that the callee have the `ssmlgcmap_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:

```
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:

```
int64_t *pop(size_t i) {
  // Get memory block from beginning of free list at index i:
  int64_t *m = free_lists[i];
  // Remove the memory block from the free list:
  free_list[i] = (int64_t*)m[2];
  // Now head of the free list is next memory block, possibly NULL.
  if (free_list[i]) {
    // If there was a next memory block:
    // We remove its previous pointer:
    free_list[i][1] = 0;
    // And we remove the next pointer from m:
    m[2] = 0;
  }
  return m;
}
```

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:

```
int64_t *split(size_t i, size_t j) {
  // Remove memory block from free_lists[i+j]:
  int64_t *first = pop(i+j);
  if (j == 0)
    // Done. Return memory block from free_lists[i].
    return first;
  // Otherwise we need to split memory block first.
  // Get size of the memory block:
  int64_t old_size = first[0];
  // When splitting the memory block we half the size of the
  // memory block. Compute size of the two new memory blocks:
  int64_t new_size = old_size / 2;
  // Save size in first memory block:
  first[0] = new_size;
  // The size of the memory blocks are in bytes, but we index
  // an int64_t array. Compute index of second new memory block:
  int64_t second_index = new_size/sizeof(int64_t);
  // Get pointer to second memory block:
  int64_t *second = &first[second_index];
  // Store size of second memory block:
```

```
  second[0] = new_size;
  // Insert second in free_lists[i+j-1]:
  push(second, i+j-1);
  // Insert first in free_lists[i+j-1]:
  push(first, i+j-1);
  // Continue until first has size fitting free_lists[i].
  return split(i, j-1);
}
```

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:

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

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:

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

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

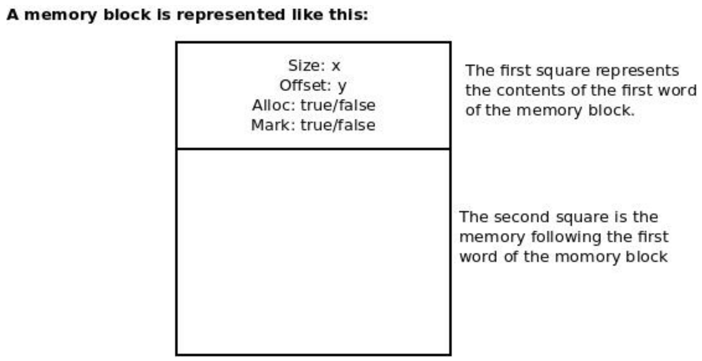
Note that the address returned to the user is not going to be the beginning of the memory block. When memory block pointed to by `int64_t* b`, which will be used to serve the users memory allocation, has been removed from its free then pointer `b+1` is returned to the user. This makes sure the user does 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.

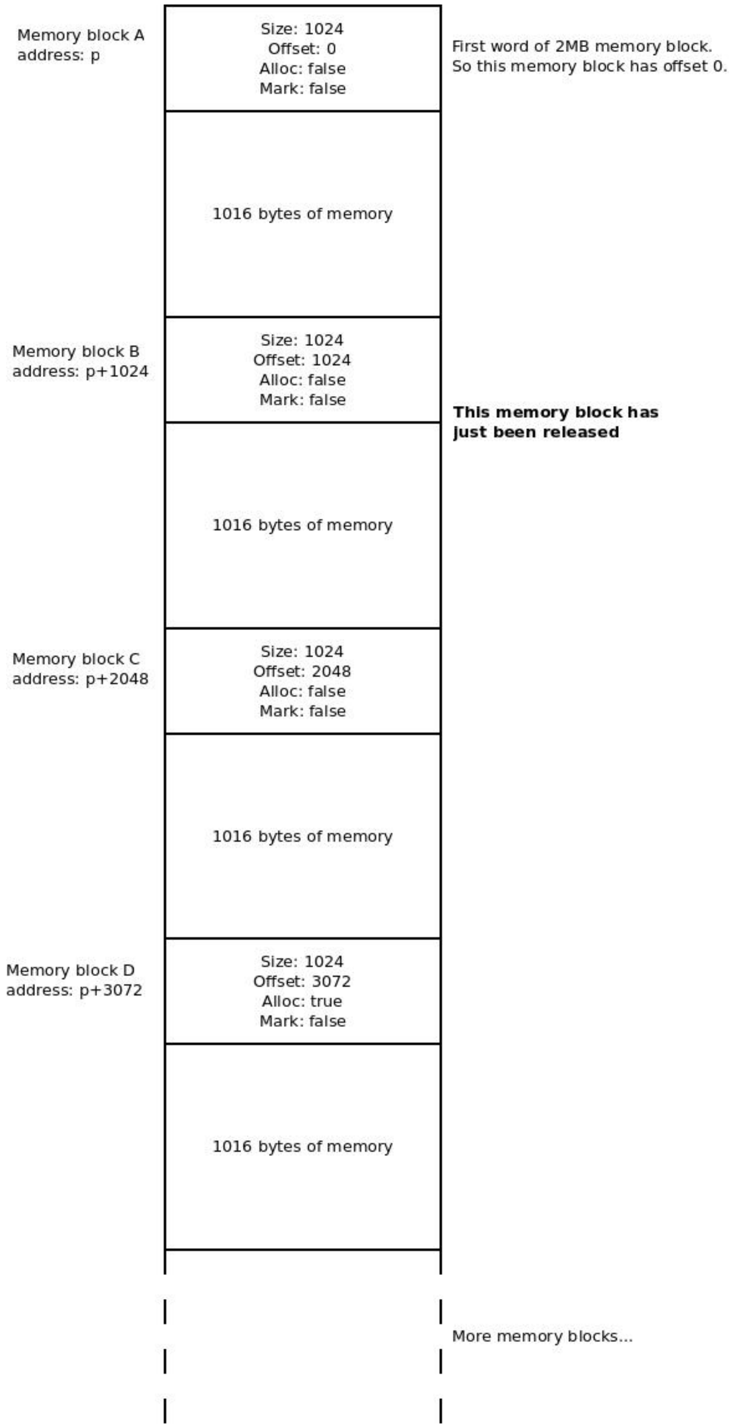
Say that we have a pointer to a memory block `b`. In the previous examples we assumed that we could get the size by indexing `b[0]`. In reality `b[0]` contains its size, and its byte offset to the 2MB block it originates from, and an allocated bit indicating whether the block is free or allocated, and 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 ($y$ mod $(2 \cdot x) = 0$) 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:

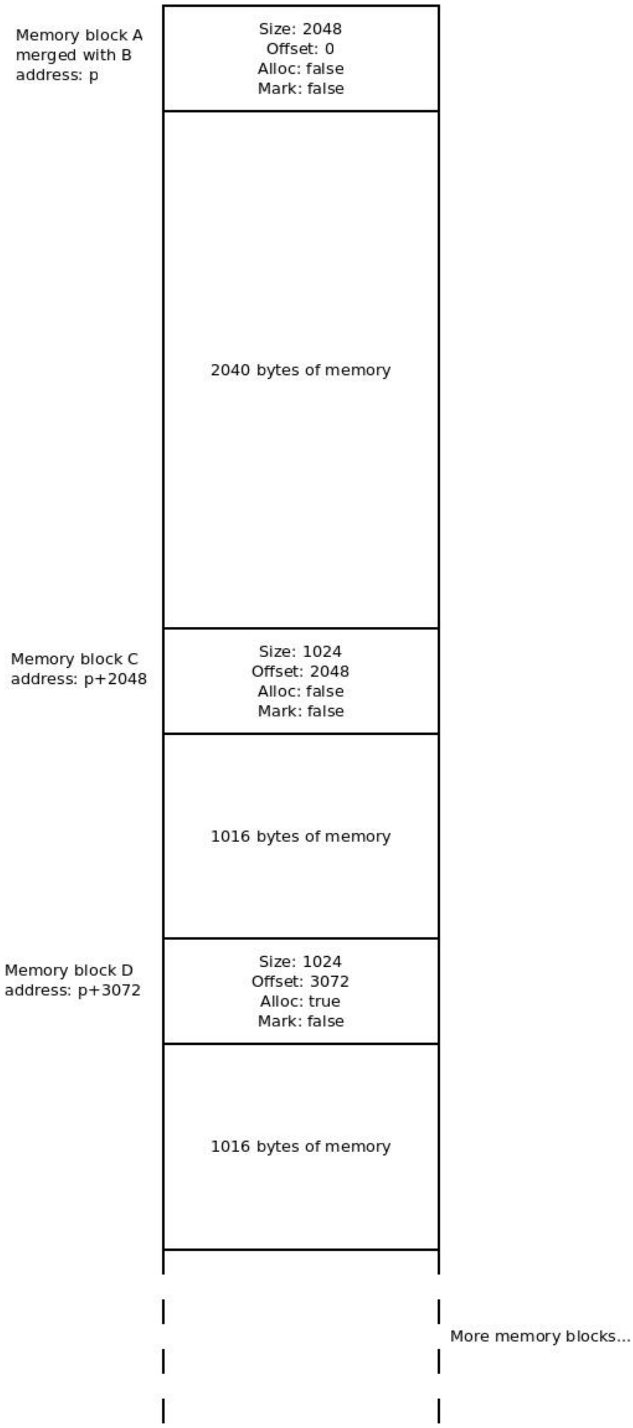



Say that a GC has been triggered and that memory block B in the following illustration has been released:




Using the formula to find the buddy of memory block B, we have $z = p + 1024$, $x = 1024$, $y = 1024$. Thus: $(y \bmod (2 \cdot x) = 1024 \bmod 2028 = 1024)$. 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:

```
// Function markall marks object Root and all objects
// reachable from Root. If Root is not dynamically allocated
// or if it is not a pointer, then the function does nothing.
static void markall(int64_t *Root) {
  // If Root is not a dynamically allocated pointer or
  // if Root is already marked, then return:
  if (!istraced(Root) || ismarked(Root))
    return;

  // Linked list containing reachable unmarked objects:
  std::forward_list<int64_t *> Objects;
```

```
  // Initialliy Root is unmarked:
  Objects.push_front(Root);

  do {
    // Get Obj from front of list:
    int64_t *Obj = Objects.front();
    // Remove front of list:
    Objects.pop_front();

    // If allocated bit is not set, then it is an error:
    assert(isAllocated(Obj));

    // Mark Obj, and return the word size of Obj:
    int64_t Size = mark(Obj);
    // For each word in Obj:
    for (int64_t I = 0; I < Size; ++I) {
      // Get Child from index I of Obj:
      int64_t *Child = (int64_t *)Obj[I];
      // If Child is a dynamically allocated object and
      // it has not been marked, then put it in list with
      // unmakred reachable objects:
      if (istraced(Child) && !ismarked(Child))
        Objects.push_front(Child);
    }
    // While there are reachable unmarked objects:
  } while (!Objects.empty());
}
```

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 unmerked 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:

```
int64_t *b = m;
// End address of 2MB block.
uint64_t end = (uint64_t)b + pow(2, 21);
// While memory block b is within the bounds of m:
while ((uint64_t)b < end) {
  // Use memory block b:
  ...
  // Get size of b masking out mark, allocated and offset bits.
  uint64_t size = b[0] & 0x7FFFFF;
  // Get next memory block:
  b = (uint64_t)b + size;
}
```

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

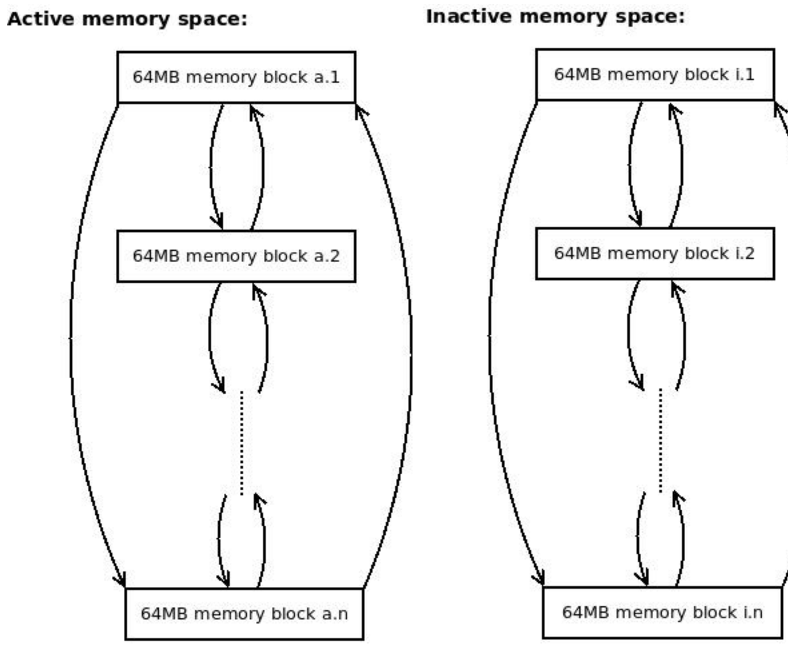
Like with the mark and sweep collector the copying collector is specific to the `x86-64` architecture.

## 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:




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:

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

```
}
// Returns true if there is free memory in active_block to
// serve a memory allocation of parameter total bytes,
// otherwise returns false:
bool active_offset_in_bounds(uint64_t total) {
  // Get address &active_block[active_index] after an allocation
  // of total bytes.
  uint64_t end = (uint64_t)(active_block + active_index) + total;
  // Get address of active_block:
  uint64_t start = (uint64_t)active_block;
  // Return whether an allocation of total bytes is inside
  // the bounds of active_block:
  return (end-start) <= pow(2,26);
}
// Allocate parameter total bytes from active_block:
int64_t *active_allocate_curr_block(uint64_t total) {
  // Get beginning of memory chunk which will be allocated:
  int64_t *chunk = active_block;
  // Store index where next allocation will begin:
  active_index += total / sizeof(int64_t);
  // Store size of chunk in first word:
  chunk[0] = total;
  // Return pointer to chunk the user can use (this does not
  // include the first word of the chunk used for the size):
  return chunk + 1;
}
// Try to allocate parameter total bytes of memory from the
// memory block after active_block. If that is not possible
// get the necessary memory by triggering a GC. If that does
// not work also, then ask OS for more memory.
int64_t *active_allocate_next_block(uint64_t total) {
  // Make the active block be the next block pointed
  // to by the current active block:
  active_block = (int64_t*)active_block[1];
  if (active_block != active_head)
    // Then the "new" active block is unused.
    // Allocate the memory from that block:
    return active_allocate_curr_block(total);
  // There are no unused memory blocks left and the last
  // memory block was unable to serve the memory allocation.
  // Now we trigger a GC to see if we can free memory
  // such that we can serve the allocation without
  // asking the OS for more memory:
  triggerGC();
  // After the garbage collection we check if we can
  // get a memory block from the new active memory block
  // (which before the GC was an inactive memory block):
  if (active_offset_in_bounds(total))
    return active_allocate_curr_block(total);
  // Otherwise, there might be a free memory
  // block after the new active block:
  active_block = (int64_t*)active_block[1];
  if (active_block != active_head) {
    // After the GC there is an unused memory block.
    // Allocate the memory from that block:
    return active_allocate_curr_block(total);
  }
  // We were unable to free enough memory to serve
  // the memory allocation. We ask the OS for 2 new
  // memory blocks:
  active_block = malloc(pow(2,26));
  int64_t *new_inactive = malloc(pow(2,26));
```

```
  // Insert the two new memory blocks in the lists:
  insert(active_head, active_block);
  insert(inactive_head, new_inactive);
  // Now we got memory to serve the allocation:
  return active_allocate_curr_block(total);
}
// Function to insert memory block parameter new_block
// before memory block parameter list_node in the doubly
// linked list of memory blocks:
void insert(int64_t *list_node, int64_t *new_block) {
  // Get list_node's prev pointer:
  int64_t *prev = (int64_t*)list_node[0];
  // new_block's prev pointer will become list_node's prev pointer:
  new_block[0] = (int64_t)prev;
  // new_block's prev pointer's next pointer will become
  // new_block instead of list_node:
  prev[1] = (int64_t)new_block;
  // new_block's next pointer will become list_node:
  new_block[1] = (int64_t)list_node;
  // list_node's prev pointer will become new_block:
  list_node[0] = (int64_t)new_block;
}
// Function called when a GC is necessary. This function
// will copy reachable memory from active space to inactive space
// and afterwards swap which space is active and inactive:
void triggerGC() {
  // For a C++ implementation, see appendix runtime/CopyingGC.cpp
  // A lot of the workings of this function is also described later
}
```

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:

```
// Function memoryCopy will forward (copy) parameter Object and
// the objects directly and indirectly reachable from Object
// from inactive to active memory space.
// The function returns the new address of Object after
// it has benn copied to inactive memory space.
static int64_t *memoryCopy(int64_t *Object) {
  // Ret is destination memory location for Object.
  int64_t Ret;

  // List containing destination addresses and objects
  // which will be copied and assigned to its destination
  // memory address afterwards:
  std::forward_list<std::pair<int64_t *, int64_t *>> MemObjStack;
  // Push Object and its destination address:
  MemObjStack.emplace_front(&Ret, Object);

  do {
    // Pop destination address and associated object from stack:
    auto P = MemObjStack.front();
    MemObjStack.pop_front();
    int64_t *DestMem = P.first;
    int64_t *Obj = P.second;

    if (!istraced(Obj)) {
      // Obj is not a heap pointer, so should not be forwarded.
      // Just store its value to its new memory field.
      *DestMem = (int64_t)Obj;
      continue;
    }

    // Get the word size of Obj:
    int64_t Size = getObjectSize(Obj);
    // Word size of Obj must never be 0:
    assert(Size);
```

```

    if (!isSmallMemory(Obj)) {
      // If Obj is a big memory object >= 16kB,
      // then the object is an array. We do not forward the object,
      // and we only forward its elements if they are non-scalar
      // and if they have not yet been forwarded.

      // Index 0 of arrays contain the array’s size.
      // Index 1 of arrays contain the first element.
      // If first element is non-scalar, then so are the rest.
      if (!isForwarded(Obj) && istraced((int64_t *)Obj[1])) {
        // Array contains non-scalar elements which have not yet
        // been forwarded. Add them to list so they can be copied:
        for (int64_t I = 1; I < Size - 1; ++I) {
          int64_t *Child = (int64_t *)Obj[I];
          MemObjStack.emplace_front(&Obj[I], Child);
        }
      }
      // Store Obj to its destination memory location:
      *DestMem = (int64_t)Obj;
      // Mark Obj as forwarded:
      markForwarded(Obj);
      continue;
    }

    if (isForwarded(Obj)) {
      // Obj has already been forwarded.
      // Store its forwarded address to its destination address:
      *DestMem = *Obj;
      continue;
    }

    // If we get here; Obj is a heap allocated pointer and has not
    // yet been forwarded. So we forward it now and store its
    // forwarded address to its destination address.

    // Allocate memory from inactive heap. This is the new inactive
    // memory address of Obj. This will be destination memory for
    // fields from Obj:
    int64_t *Forward = memoryForwardAlloc(Size * sizeof(int64_t));

    // Put fields from Obj in list. Together with
    // destination address:
    for (int64_t I = 0; I < Size - 1; ++I) {
      int64_t *Child = (int64_t *)Obj[I];
      MemObjStack.emplace_front(&Forward[I], Child);
    }

    // Insert forwarded address of Obj at index 0 of Obj
    // and mark Obj forwarded:
    *Obj = (int64_t)Forward;
    markForwarded(Obj);

    // Store forwarded memory address of Obj to destination
    // address:
    *DestMem = (int64_t)Forward;
  // While there are reachable objects not yet copied...
  } while (!MemObjStack.empty());

  // Return inactive memory address of copy of Object:
  return (int64_t *)Ret;
}
```

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:

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

The generated LLVM assembly is as follows:

```
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

@FirstGenBegin = external externally_initialized constant i64
@FirstGenEnd = external externally_initialized constant i64

; Function Attrs: nounwind
declare void @llvm.gcroot(i8**, i8*) #0 gc "ssml"

declare i64* @allocate(i64) gc "ssml"

; Function Attrs: noreturn
declare void @matchError() #1

declare void @rememberset_insert(i64*)

define i64 @entry() gc "ssml" {
entrylabel:
  %tempptr = alloca i64*, align 8
  %0 = bitcast i64** %tempptr to i8**
  tail call void @llvm.gcroot(i8** %0, i8* null)
  %temp = bitcast i64** %tempptr to i64*
  %tempptr1 = alloca i64*, align 8
```

```
  %1 = bitcast i64** %tempptr1 to i8**
  tail call void @llvm.gcroot(i8** %1, i8* null)
  %temp2 = bitcast i64** %tempptr1 to i64*
  %locals = alloca i64*, align 8
  store volatile i64 1, i64* %temp, align 8
  %2 = bitcast i64** %locals to i8**
  tail call void @llvm.gcroot(i8** %2, i8* null)
  %3 = tail call i64* @allocate(i64 16)
  store volatile i64* %3, i64** %locals, align 8
  %frame = load volatile i64*, i64** %locals, align 8
  %4 = ptrtoint i64* %frame to i64
  %5 = load i64, i64* @FirstGenEnd, align 8
  %6 = icmp uge i64 %4, %5
  %7 = load i64, i64* @FirstGenBegin, align 8
  %8 = icmp ult i64 %4, %7
  %or.cond.i = or i1 %6, %8
  br i1 %or.cond.i, label %9, label %ssmlMemorybarrier.exit

; <label>:9                  ; preds = %entrylabel
  tail call void @rememberset_insert(i64* %frame)
  br label %ssmlMemorybarrier.exit

ssmlMemorybarrier.exit:     ; preds = %entrylabel, %9
  store volatile i64 0, i64* %frame, align 8
  %10 = tail call i64* @allocate(i64 16)
  %11 = ptrtoint i64* %10 to i64
  store volatile i64 %11, i64* %temp2, align 8
  %12 = load i64, i64* @FirstGenEnd, align 8
  %13 = icmp uge i64 %11, %12
  %14 = load i64, i64* @FirstGenBegin, align 8
  %15 = icmp ult i64 %11, %14
  %or.cond.i.1 = or i1 %13, %15
  br i1 %or.cond.i.1, label %16, label %ssmlMemorybarrier.exit2

; <label>:16                 ; preds = %ssmlMemorybarrier.exit
  tail call void @rememberset_insert(i64* %10)
  br label %ssmlMemorybarrier.exit2

ssmlMemorybarrier.exit2:    ; preds = %ssmlMemorybarrier.exit, %16
  store volatile i64 ptrtoint (i64 (i64, i64)* @_1head to i64),
      i64* %10, align 8
  %17 = getelementptr i64, i64* %10, i64 1
  %frame3 = load volatile i64*, i64** %locals, align 8
  %18 = ptrtoint i64* %frame3 to i64
  %19 = and i64 %18, 1
  %20 = icmp eq i64 %19, 0
  br i1 %20, label %21, label %ssmlMemorybarrier.exit4

; <label>:21                 ; preds = %ssmlMemorybarrier.exit2
  %22 = load i64, i64* @FirstGenEnd, align 8
  %23 = icmp uge i64 %11, %22
  %24 = load i64, i64* @FirstGenBegin, align 8
  %25 = icmp ult i64 %11, %24
  %or.cond.i.3 = or i1 %23, %25
  br i1 %or.cond.i.3, label %26, label %ssmlMemorybarrier.exit4

; <label>:26                 ; preds = %21
  tail call void @rememberset_insert(i64* %10)
  br label %ssmlMemorybarrier.exit4

ssmlMemorybarrier.exit4:    ; preds = %ssmlMemorybarrier.exit2,
    %21, %26
```

```
  store volatile i64 %18, i64* %17, align 8
  %frame4 = load volatile i64*, i64** %locals, align 8
  %27 = getelementptr i64, i64* %frame4, i64 1
  %28 = and i64 %11, 1
  %29 = icmp eq i64 %28, 0
  br i1 %29, label %30, label %ssmlMemorybarrier.exit6

; <label>:30                  ; preds = %ssmlMemorybarrier.exit4
  %31 = ptrtoint i64* %frame4 to i64
  %32 = load i64, i64* @FirstGenEnd, align 8
  %33 = icmp uge i64 %31, %32
  %34 = load i64, i64* @FirstGenBegin, align 8
  %35 = icmp ult i64 %31, %34
  %or.cond.i.5 = or i1 %33, %35
  br i1 %or.cond.i.5, label %36, label %ssmlMemorybarrier.exit6

; <label>:36                  ; preds = %30
  tail call void @rememberset_insert(i64* %frame4)
  br label %ssmlMemorybarrier.exit6

ssmlMemorybarrier.exit6:    ; preds = %ssmlMemorybarrier.exit4,
   %30, %36
  store volatile i64 %11, i64* %27, align 8
  ret i64 0
}

define i64 @_1head(i64, i64) gc "ssml" {
entrylabel:
  %tempptr = alloca i8*, align 8
  tail call void @llvm.gcroot(i8** nonnull %tempptr, i8* null)
  %tempptr1 = alloca i64*, align 8
  %2 = bitcast i64** %tempptr1 to i8**
  tail call void @llvm.gcroot(i8** %2, i8* null)
  %temp2 = bitcast i64** %tempptr1 to i64*
  %tempptr3 = alloca i64*, align 8
  %3 = bitcast i64** %tempptr3 to i8**
  tail call void @llvm.gcroot(i8** %3, i8* null)
  %temp4 = bitcast i64** %tempptr3 to i64*
  %locals = alloca i64*, align 8
  store volatile i64 %0, i64* %temp2, align 8
  store volatile i64 %1, i64* %temp4, align 8
  %4 = bitcast i64** %locals to i8**
  tail call void @llvm.gcroot(i8** %4, i8* null)
  %5 = tail call i64* @allocate(i64 24)
  store volatile i64* %5, i64** %locals, align 8
  %6 = load volatile i64, i64* %temp2, align 8
  %frame = load volatile i64*, i64** %locals, align 8
  %7 = and i64 %6, 1
  %8 = icmp eq i64 %7, 0
  br i1 %8, label %9, label %ssmlMemorybarrier.exit

; <label>:9                 ; preds = %entrylabel
  %10 = ptrtoint i64* %frame to i64
  %11 = load i64, i64* @FirstGenEnd, align 8
  %12 = icmp uge i64 %10, %11
  %13 = load i64, i64* @FirstGenBegin, align 8
  %14 = icmp ult i64 %10, %13
  %or.cond.i = or i1 %12, %14
  br i1 %or.cond.i, label %15, label %ssmlMemorybarrier.exit

; <label>:15                ; preds = %9
  tail call void @rememberset_insert(i64* %frame)
```

```
  br label %ssmlMemorybarrier.exit

ssmlMemorybarrier.exit:   ; preds = %entrylabel, %9, %15
  store volatile i64 %6, i64* %frame, align 8
  %16 = load volatile i64, i64* %temp4, align 8
  %17 = inttoptr i64 %16 to i64*
  %18 = load volatile i64, i64* %17, align 8
  %19 = icmp ult i64 %18, 2
  br i1 %19, label %20, label %fun_match_fail

; <label>:20              ; preds = %ssmlMemorybarrier.exit
  %21 = getelementptr i64, i64* %17, i64 1
  %22 = load volatile i64, i64* %21, align 8
  %23 = inttoptr i64 %22 to i64*
  %24 = load volatile i64, i64* %23, align 8
  %frame5 = load volatile i64*, i64** %locals, align 8
  %25 = getelementptr i64, i64* %frame5, i64 1
  %26 = and i64 %24, 1
  %27 = icmp eq i64 %26, 0
  br i1 %27, label %28, label %ssmlMemorybarrier.exit2

; <label>:28              ; preds = %20
  %29 = ptrtoint i64* %frame5 to i64
  %30 = load i64, i64* @FirstGenEnd, align 8
  %31 = icmp uge i64 %29, %30
  %32 = load i64, i64* @FirstGenBegin, align 8
  %33 = icmp ult i64 %29, %32
  %or.cond.i.1 = or i1 %31, %33
  br i1 %or.cond.i.1, label %34, label %ssmlMemorybarrier.exit2

; <label>:34                                       ; preds = %28
  tail call void @rememberset_insert(i64* %frame5)
  br label %ssmlMemorybarrier.exit2

ssmlMemorybarrier.exit2:  ; preds = %20, %28, %34
  store volatile i64 %24, i64* %25, align 8
  %35 = getelementptr i64, i64* %23, i64 1
  %36 = load volatile i64, i64* %35, align 8
  %frame6 = load volatile i64*, i64** %locals, align 8
  %37 = getelementptr i64, i64* %frame6, i64 2
  %38 = and i64 %36, 1
  %39 = icmp eq i64 %38, 0
  br i1 %39, label %40, label %ssmlMemorybarrier.exit4

; <label>:40               ; preds = %ssmlMemorybarrier.exit2
  %41 = ptrtoint i64* %frame6 to i64
  %42 = load i64, i64* @FirstGenEnd, align 8
  %43 = icmp uge i64 %41, %42
  %44 = load i64, i64* @FirstGenBegin, align 8
  %45 = icmp ult i64 %41, %44
  %or.cond.i.3 = or i1 %43, %45
  br i1 %or.cond.i.3, label %46, label %ssmlMemorybarrier.exit4

; <label>:46                                       ; preds = %40
  tail call void @rememberset_insert(i64* %frame6)
  br label %ssmlMemorybarrier.exit4

ssmlMemorybarrier.exit4:   ; preds = %ssmlMemorybarrier.exit2,
   %40, %46
  store volatile i64 %36, i64* %37, align 8
  %frame7 = load volatile i64*, i64** %locals, align 8
  %47 = getelementptr i64, i64* %frame7, i64 1
```

```
  %48 = load volatile i64, i64* %47, align 8
  ret i64 %48

fun_match_fail:              ; preds = %ssmlMemorybarrier.exit
  tail call void @matchError()
  unreachable
}

attributes #0 = { nounwind }
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:

```
fun genlist(size: int): int list =
let
  fun gen(0: int): int list = []
    | gen(n: int): int list = size::gen(n-1)
in
  gen size
end

fun allocList(listSize: int): unit =
let
```

```
  val _: int list = genlist listSize
in
  ()
end

fun allocLists(_:        int)(0:            int): int = 0
  | allocLists(listSize: int)(numberLists: int): int =
  let
    val () = allocList listSize
  in
    1 + allocLists listSize (numberLists-1)
  end

val _:int = allocLists 3000 3000
val _:int = allocLists 3000 3000
val _:int = allocLists 3000 3000
val _:int = allocLists 3000 3000
val _:int = allocLists 3000 3000
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:

```
fun getList(size:int): int list =
let
  val n: int ref = ref 0
  val ret: int list ref = ref []
in
  while (!n < size) do (
    ret := !n :: !ret;
    n := !n + 1
  );
  !ret
end

fun alloc1(): unit =
let
  val a: int array =
    let
      val x: int array = Array.array(300000, 0)
      val y: int array = Array.array(300000, 0)
      val z: int array = Array.array(300000, 0)
    in
      x
    end

  val a: int array =
    let
      val x: int array = Array.array(300000, 0)
      val y: int array = Array.array(300000, 0)
      val z: int array = Array.array(300000, 0)
    in
      x
    end

  val size: int = 2100077
  val list: int list = getList size
```

```
in
  ()
end

fun alloc2(): unit =
let
  val size: int = 1300089
  val list: int list = getList size
  val list: int list = getList size
  val list: int list = getList size
in
  ()
end

fun run(): unit =
let
  val () = alloc1 ()
  val () = alloc1 ()
  val () = alloc1 ()

  val x: int array = Array.array(300000, 0)

  val () = alloc2 ()
  val () = alloc2 ()
  val () = alloc2 ()
  val () = alloc2 ()
  val () = alloc2 ()

  val () = alloc1 ()
  val () = alloc1 ()
  val () = alloc1 ()
  val x: int array = Array.array(300000, 0)
  val () = alloc1 ()
  val () = alloc1 ()
  val () = alloc1 ()

  val () = alloc2 ()
  val () = alloc2 ()
  val () = alloc2 ()
  val () = alloc2 ()
  val () = alloc2 ()
in
  ()
end

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:

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

```

```
fun head(x::_: 'a list): 'a      = x

fun tail(_::x: 'a list): 'a list = x

fun null([]: 'a list): bool = true
  | null(_ : 'a list): bool = false

fun foldl(_: 'a*'b -> 'b)(acc: 'b)([]: 'a list): 'b =
    acc
  | foldl(f: 'a*'b -> 'b)(acc: 'b)(x::xs: 'a list): 'b =
    foldl f (f(x,acc)) xs

fun map(f: 'a -> 'b): 'a list -> 'b list =
  foldl (fn (x: 'a, acc: 'b list) => f x :: acc) []

fun filter(f: 'a -> bool): 'a list -> 'a list =
  foldl (fn (x: 'a, acc: 'a list) => if f x then x::acc else acc)
      []

infixr 5 @
fun (xs: 'a list) @ (ys: 'a list): 'a list = let
  fun cat([]   : 'a list,
          ys   : 'a list,
          cont: 'a list -> 'a list): 'a list =
      cont ys
    | cat(x::xs: 'a list,
          ys    : 'a list,
          cont : 'a list -> 'a list): 'a list =
      cat(xs, ys, fn acc: 'a list => cont (x::acc))
in
  cat(xs, ys, id)
end

fun concat(xs: 'a list list): 'a list = foldl op@ [] xs

fun range(x:int)(y:int): int list = let
  fun rng(acc: int list)(x:int)(y:int): int list =
      if x > y
      then acc
      else rng (y::acc) x (y-1)
in
  rng [] x y
end

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:

```
fun subsetSum(_: int)([]: int list):bool =
    false
  | subsetSum(result: int)(set: int list):bool = let
  fun recurse(h: int list, []: int list):bool =
      sum h = result
    | recurse(h: int list, t::ts: int list):bool =
      if subsetSum result (h@ts)
      then true
```

```
      else recurse(t::h, ts)
in
  recurse([], set)
end

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:

```
fun combinate(xs: 'a list): 'a list list = let
  fun sublists(acc: 'a list list)(_: 'a list)([]: 'a list): 'a
      list list =
      acc
    | sublists(acc: 'a list list)(hs: 'a list)(t::ts: 'a list): 'a
        list list =
      sublists ((hs@ts)::acc) (t::hs) ts
in
  xs :: concat (map combinate (sublists [] [] xs))
end

fun subsetSum(result:int)(set: int list): bool = let
  val cs : int list list = combinate set
```

```
  val res: int list list = filter (fn x: int list => sum x =
      result) cs
in
  (fn []: int list list => false | _: int list list => true) res
end

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.

```
fun subsetSum(_: int)([]: int list):bool =
    false
  | subsetSum(result: int)(set: int list):bool = let
  val ret: bool ref = ref(sum set = result)
  val hs : int list ref = ref []
  val ts : int list ref = ref set
in
  while not(!ret) andalso not(null(!ts)) do let
    val t: int = head(!ts)
  in
    ts := tail(!ts);
    if subsetSum result (!hs @ !ts)
    then ret := true
    else (hs := t :: !hs)
  end;
  !ret
end

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

```
#include <stdlib.h>
#include <stdio.h>
#include <inttypes.h>

void *print(int64_t i) {
  if (i & 1)
    printf("%" PRId64 "\n", i >> 1);
  else
    printf("%p\n", (void *)i);
  return 0;
}

void *framePrint(void *Frame, int64_t i) {
  (void)Frame;
  return print(i);
}

int64_t framePrintVal[2];

void initFramePrintVal() {
  framePrintVal[0] = (int64_t)framePrint;
  framePrintVal[1] = 0;
}
```

Listing 2: runtime/Error.c

```
#include <stdlib.h>
#include <stdio.h>

void error(const char *Msg) {
  fprintf(stderr, "runtime error: %s\n", Msg);
  exit(1);
}
```

Listing 3: runtime/MatchError.c

```c
#include <stdio.h>
#include <stdlib.h>

void error(const char *);

void matchError() {
  error("pattern match failed");
}
```

Listing 4: runtime/StdLib.c

```c

#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

#ifdef SSML_GENERATIONAL_GC
void rememberset_insert(int64_t *DestBase);
#endif

int64_t *allocateFrame(int64_t N, void *Frame);
void error(const char *);

static inline int64_t mark(int64_t V) { return (V << 1) + 1; }

static inline int64_t unmark(int64_t V) { return V >> 1; }

int64_t arrayempty[1] = {0};

int64_t doarrayupdate(int64_t *Ary, int64_t N, int64_t V) {
  N = unmark(N);
  if (N < 0 || N >= unmark(Ary[0]))
    error("array index out of bounds");
  Ary[N + 1] = V;
#ifdef SSML_GENERATIONAL_GC
  if (!(V & 1))
    rememberset_insert(Ary);
#endif
  return 0;
}

int64_t arrayupdate(int64_t Env, int64_t A[3]) {
  return doarrayupdate((int64_t *)A[0], A[1], A[2]);
}
int64_t frameArrayupdate[2];

int64_t plus(int64_t Env, int64_t A[2]) {
  return mark(unmark(A[0]) + unmark(A[1]));
}
int64_t framePlus[2];

int64_t minus(int64_t Env, int64_t A[2]) {
  return mark(unmark(A[0]) - unmark(A[1]));
}
int64_t frameMinus[2];

int64_t multiply(int64_t Env, int64_t A[2]) {
  return mark(unmark(A[0]) * unmark(A[1]));
}
int64_t frameMultiply[2];
```

```
int64_t division(int64_t Env, int64_t A[2]) {
  return mark(unmark(A[0]) / unmark(A[1]));
}
int64_t frameDivision[2];

int64_t modulo(int64_t Env, int64_t A[2]) {
  return mark(unmark(A[0]) % unmark(A[1]));
}
int64_t frameModulo[2];

int64_t refassign(int64_t Env, int64_t A[2]) {
  int64_t *Dst = (int64_t *)A[0];
  int64_t Src = A[1];
  *Dst = Src;
#ifdef SSML_GENERATIONAL_GC
  if (!(Src & 1))
    rememberset_insert(Dst);
#endif
  return 0;
}
int64_t frameRefassign[2];

static int64_t doarrayFrame(int64_t N, int64_t V, void *Frame) {
  int64_t I;
  int64_t *Ret;

  int64_t Un = unmark(N);

  if (Un < 1)
    error("array size must be greater than 0");

  Ret = allocateFrame(Un * sizeof(int64_t) + sizeof(int64_t),
      Frame);
  Ret[0] = N;
  for (I = 1; I < Un; ++I)
    Ret[I] = V;
  return (int64_t)Ret;
}

int64_t doarray(int64_t N, int64_t V) {
  return doarrayFrame(N, V, __builtin_frame_address(0));
}

int64_t array(int64_t Env, int64_t A[2]) {
  return doarrayFrame(A[0], A[1], __builtin_frame_address(0));
}
int64_t frameArray[2];

int64_t doarrayget(int64_t *Ary, int64_t N) {
  N = unmark(N);
  if (N < 0 || N >= unmark(Ary[0]))
    error("array index out of bounds");
  return Ary[1 + N];
}

int64_t arrayget(int64_t Env, int64_t A[2]) {
  return doarrayget((int64_t *)A[0], A[1]);
}
int64_t frameArrayget[2];

int64_t negate(int64_t Env, int64_t Val) { return
```

```
    mark(-unmark(Val)); }
int64_t frameNegate[2];

int64_t not(int64_t Env, int64_t Val) {
  return Val == mark(1) ? mark(0) : mark(1);
}
int64_t frameNot[2];

int64_t ref(int64_t Env, int64_t Val) {
  int64_t *Ret = allocateFrame(sizeof(int64_t), __builtin_frame_address(0));
  *Ret = Val;
  return (int64_t)Ret;
}
int64_t frameRef[2];

int64_t arraylength(int64_t Env, int64_t Val) { return *(int64_t *)Val; }
int64_t frameArraylength[2];

int64_t deref(int64_t Env, int64_t Val) { return *(int64_t *)Val; }
int64_t frameDeref[2];

int64_t equals(int64_t Env, int64_t A[2]) {
  return mark((int64_t)(A[0] == A[1] ? 1 : 0));
}
int64_t frameEquals[2];

int64_t notEquals(int64_t Env, int64_t A[2]) {
  return mark((int64_t)(A[0] != A[1] ? 1 : 0));
}
int64_t frameNotEquals[2];

int64_t less(int64_t Env, int64_t A[2]) {
  return mark((int64_t)(A[0] < A[1] ? 1 : 0));
}
int64_t frameLess[2];

int64_t greater(int64_t Env, int64_t A[2]) {
  return mark((int64_t)(A[0] > A[1] ? 1 : 0));
}
int64_t frameGreater[2];

int64_t lessEquals(int64_t Env, int64_t A[2]) {
  return mark((int64_t)(A[0] <= A[1] ? 1 : 0));
}
int64_t frameLessEquals[2];

int64_t greaterEquals(int64_t Env, int64_t A[2]) {
  return mark((int64_t)(A[0] >= A[1] ? 1 : 0));
}
int64_t frameGreaterEquals[2];

void initStdLib() {
  frameArrayupdate[0] = (int64_t)arrayupdate;
  frameArrayupdate[1] = 0;

  framePlus[0] = (int64_t)plus;
  framePlus[1] = 0;
  frameMinus[0] = (int64_t)minus;
  frameMinus[1] = 0;
  frameMultiply[0] = (int64_t)multiply;
```

```
  frameMultiply[1] = 0;
  frameDivision[0] = (int64_t)division;
  frameDivision[1] = 0;
  frameModulo[0] = (int64_t)modulo;
  frameModulo[1] = 0;
  frameRefassign[0] = (int64_t)refassign;
  frameRefassign[1] = 0;
  frameArray[0] = (int64_t)array;
  frameArray[1] = 0;
  frameArrayget[0] = (int64_t)arrayget;
  frameArrayget[1] = 0;

  frameNegate[0] = (int64_t)negate;
  frameNegate[1] = 0;
  frameNot[0] = (int64_t) not;
  frameNot[1] = 0;
  frameRef[0] = (int64_t)ref;
  frameRef[1] = 0;
  frameDeref[0] = (int64_t)deref;
  frameDeref[1] = 0;
  frameArraylength[0] = (int64_t)arraylength;
  frameArraylength[1] = 0;

  frameEquals[0] = (int64_t)equals;
  frameEquals[1] = 0;
  frameNotEquals[0] = (int64_t)notEquals;
  frameNotEquals[1] = 0;
  frameLess[0] = (int64_t)less;
  frameLess[1] = 0;
  frameGreater[0] = (int64_t)greater;
  frameGreater[1] = 0;
  frameLessEquals[0] = (int64_t)lessEquals;
  frameLessEquals[1] = 0;
  frameGreaterEquals[0] = (int64_t)greaterEquals;
  frameGreaterEquals[1] = 0;
}

void patternCheck(int64_t Val) {
  if (Val > 1000) {
    fprintf(stderr, "VAL %p\n", (void *)Val);
    error("VAL TOO BIG!");
  }
}
```

Listing 5: runtime/GenerationalGC.cpp

```
#ifdef SSML_GENERATIONAL_GC

#include "GC.h"

#include "ssml/Common/Timer.h"
#include "llvm/ADT/DenseSet.h"

#include <strings.h>
#include <iostream>
#include <stdint.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <forward_list>

#define MMAP_ALLOC
```

```

using namespace ssml;

extern "C" void error(const char *);

static uint64_t TriggerGCCount;
static uint64_t TotalGCTime;
static uint64_t BestGCTime = UINT64_MAX;
static uint64_t WhorstGCTime;
static uint64_t TotalGCRoots;
static uint64_t GCMemoryUsage;
static uint64_t FullGCCount;
static uint64_t FirstGenGCCount;

extern "C" void setFirstGenBegin(uint64_t);
extern "C" void setFirstGenEnd(uint64_t);

extern "C" void ssmlGCPrintStatistics() {
  std::cout << "**** Generational GC Statistics ****\n";
#ifdef SSML_SHADOW_STACK_GC
  std::cout << "Used Shadow Stack GC strategy\n";
#else
  std::cout << "Used Binary Map GC strategy\n";
#endif
  std::cout << "Total GCs:             " << TriggerGCCount << '\n';
  if (!TriggerGCCount)
    return;

  double Tot = (double)TotalGCTime / 1000;
  double Avg = ((double)TotalGCTime / TriggerGCCount) / 1000;
  double Whorst = (double)WhorstGCTime / 1000;
  double Best = (double)BestGCTime / 1000;
  double AvgRoots = (double)TotalGCRoots / TriggerGCCount;
  uint64_t Avail = MAX_ALLOC_SIZE - GCMemoryUsage;

  std::cout << "Total GC Time:         " << Tot << "ms\n";
  std::cout << "Average GC Time:       " << Avg << "ms\n";
  std::cout << "Whorst GC Time:        " << Whorst << "ms\n";
  std::cout << "Best GC Time:          " << Best << "ms\n";
  std::cout << "Average number Roots:  " << AvgRoots << '\n';
  std::cout << "Number full GCs:       " << FullGCCount << '\n';
  std::cout << "First generation GCs:  " << FirstGenGCCount << '\n';
  std::cout << "Total Memory:          " << MAX_ALLOC_SIZE << "B\n";
  std::cout << "GC Memory Usage:       " << GCMemoryUsage << "B\n";
  std::cout << "Total Free Memory:     " << Avail << "B\n";
  std::cout << "************************************" << std::endl;
}

static llvm::DenseSet<int64_t *> RememberSet;

static int64_t getObjectSize(int64_t *Obj);
static bool isFirstGenPointer(int64_t *Obj);

extern "C" void rememberset_insert(int64_t *DestBase) {
  assert(!isFirstGenPointer(DestBase));
  assert(getObjectSize(DestBase));
  RememberSet.insert(DestBase);
}

// When allocating more memory than this, use a mark sweep
```

```
    algorithm instead.
static const int64_t BigAllocateThreshold = 16384; // 2^14 bytes.
// Both heaps are increased by MemoryIncrease when more memory is needed.
static const int64_t MemoryIncrease = 2097152 * 32; // 2 * 32 MB.
// First generation memory size (2 * 2 MB):
#define FIRST_GEN_SIZE (2097152 * 2)

class HeapBlock {
public:
  // Block[PrevIndex] pointer to prev block.
  // Block[NextIndex] pointer to next block.
  // Block[FreeIndex] index to next free block.
  int64_t *Block;

public:
  static const uint64_t PrevIndex = 0;
  static const uint64_t NextIndex = 1;
  static const uint64_t FreeIndex = 2;
  static const uint64_t InitialFreeIndex = 3;
  static const int64_t EndIndex = MemoryIncrease / sizeof(int64_t);
  static_assert(MemoryIncrease >= 4 * BigAllocateThreshold,
                "MemoryIncrease too small");
  static_assert(MemoryIncrease % sizeof(int64_t) == 0, "bad MemoryIncrease");

public:
  explicit HeapBlock(int64_t *Block, HeapBlock Prev, HeapBlock Next);
  HeapBlock(int64_t *InitializedBlock);

  int64_t *allocate(int64_t Size);

  operator bool();

  HeapBlock getPrev() { return (int64_t *)this->Block[PrevIndex]; };
  HeapBlock getNext() { return (int64_t *)this->Block[NextIndex]; };

  void setPrev(HeapBlock B) { this->Block[PrevIndex] = (int64_t)B.Block; };
  void setNext(HeapBlock B) { this->Block[NextIndex] = (int64_t)B.Block; };

  void zero();

  bool operator==(HeapBlock Oth) { return this->Block == Oth.Block; }
  bool operator!=(HeapBlock Oth) { return !(*this == Oth); }

  uint64_t getNumberFreeBytes() {
    return MemoryIncrease - this->Block[FreeIndex] * sizeof(int64_t);
  }
};

HeapBlock::HeapBlock(int64_t *Block, HeapBlock Prev, HeapBlock Next)
    : Block(Block) {
  this->Block[PrevIndex] = (int64_t)Prev.Block;
  this->Block[NextIndex] = (int64_t)Next.Block;
```

```
  this->Block[FreeIndex] = InitialFreeIndex;
}

HeapBlock::HeapBlock(int64_t *InitializedBlock) :
    Block(InitializedBlock) {}

int64_t *HeapBlock::allocate(int64_t WordSize) {
  assert(WordSize > 0);
  int64_t NextIndex = this->Block[HeapBlock::FreeIndex] + WordSize;
  if (NextIndex > HeapBlock::EndIndex)
    return nullptr;
  int64_t PrevIndex = this->Block[HeapBlock::FreeIndex];
  this->Block[FreeIndex] = NextIndex;
  return &this->Block[PrevIndex];
}

void HeapBlock::zero() {
  assert(this->Block);
  bzero(this->Block + 2, MemoryIncrease - 2 * sizeof(int64_t));
  this->Block[FreeIndex] = HeapBlock::InitialFreeIndex;
}

HeapBlock::operator bool() { return this->Block; }

struct FirstGenMemoryBlock {
  int64_t *Block;
  uint64_t Index;
};

FirstGenMemoryBlock FirstGenBlock;

struct BigMemoryNode {
  BigMemoryNode *Prev;
  BigMemoryNode *Next;
  int64_t *MemoryBlock;
};

BigMemoryNode BigMemoryList = {&BigMemoryList, &BigMemoryList,
    nullptr};

HeapBlock ActiveHeapBlockList(0);
HeapBlock InactiveHeapBlockList(0);
HeapBlock ActiveHeapBlock(0);

static bool activeHasCapacity(uint64_t Bytes) {
  assert(Bytes < MemoryIncrease - sizeof(int64_t) * 64);
  if (ActiveHeapBlock.getNumberFreeBytes() >= Bytes)
    return true;
  if (ActiveHeapBlock.getNext() != ActiveHeapBlockList)
    return true;
  return false;
}

static bool activeHasFirstGenCapacity() {
  return activeHasCapacity(FIRST_GEN_SIZE);
}

static const uint64_t SmallMemoryBit = (uint64_t)1 << 63;
static void markSmallMemory(int64_t *Obj) { Obj[-1] |=
    SmallMemoryBit; }

static bool isSmallMemory(int64_t *Obj) { return Obj[-1] &
```

```
    SmallMemoryBit; }

static const uint64_t ForwardBit = (uint64_t)1 << 62;
static void markForwarded(int64_t *Obj) { Obj[-1] |= ForwardBit; }

static void unmarkForwarded(int64_t *Obj) { Obj[-1] &=
    ~ForwardBit; }

static bool isForwarded(int64_t *Obj) { return Obj[-1] &
    ForwardBit; }

static int64_t getObjectSize(int64_t *Obj) {
  return Obj[-1] & ~(SmallMemoryBit | ForwardBit);
}

static void bigMemoryNodeAppend(BigMemoryNode *Node, BigMemoryNode
    *Prev,
                                BigMemoryNode *Next) {
  Node->Prev = Prev;
  Node->Next = Next;
  Prev->Next = Node;
  Next->Prev = Node;
}

static void bigMemoryNodeRemove(BigMemoryNode *Node) {
  assert(Node != &BigMemoryList);
  Node->Prev->Next = Node->Next;
  Node->Next->Prev = Node->Prev;
}

static void bigMemoryNodeAppend(BigMemoryNode *Node) {
  bigMemoryNodeAppend(Node, BigMemoryList.Prev, &BigMemoryList);
}

static int64_t *heapBlockListAllocate1(int64_t N, HeapBlock
    ListHead) {
  assert(N % sizeof(int64_t) == 0);
  N = N / sizeof(int64_t);
  assert(N > 0);

  int64_t *Mem = ActiveHeapBlock.allocate(N);
  if (Mem)
    goto out;

  ActiveHeapBlock = ActiveHeapBlock.getNext();
  if (ActiveHeapBlock == ListHead)
    return nullptr;

  Mem = ActiveHeapBlock.allocate(N);
  assert(Mem);

out:
  Mem[0] = N;
  auto Ret = Mem + 1;
  markSmallMemory(Ret);
  return Ret;
}

static int64_t *newBlock(uint64_t Size) {
  GCMemoryUsage += Size;
  if (GCMemoryUsage > MAX_ALLOC_SIZE)
    error("out of memory");
```

```
#ifdef MMAP_ALLOC
  void *Mem = mmap(nullptr, Size, PROT_READ | PROT_WRITE,
                   MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
  if (Mem == MAP_FAILED) {
    perror("unable to allocate memory");
    error("out of memory");
  }
#else
  void *Mem = calloc(1, Size);
  if (!Mem) {
    perror("unable to allocate memory");
    error("out of memory");
  }
#endif
  return (int64_t *)Mem;
}

static void increaseMemory() {
  auto ActivePrev = ActiveHeapBlockList.getPrev();
  auto Block =
      HeapBlock(newBlock(MemoryIncrease), ActivePrev,
          ActiveHeapBlockList);
  ActivePrev.setNext(Block);
  ActiveHeapBlockList.setPrev(Block);

  auto InactivePrev = InactiveHeapBlockList.getPrev();
  Block =
      HeapBlock(newBlock(MemoryIncrease), InactivePrev,
          InactiveHeapBlockList);
  InactivePrev.setNext(Block);
  InactiveHeapBlockList.setPrev(Block);
}

static int64_t *fullMemoryForwardAlloc(uint64_t NumBytes) {
  int64_t *Forward = heapBlockListAllocate1(NumBytes,
      InactiveHeapBlockList);
  if (!Forward) {
    increaseMemory();
    ActiveHeapBlock = InactiveHeapBlockList.getPrev();
    Forward = heapBlockListAllocate1(NumBytes,
        InactiveHeapBlockList);
    assert(Forward);
  }
  assert(Forward[-1]);
  return Forward;
}

static int64_t *fullMemoryCopy(int64_t *Object) {
  // Ret is new memory field of Object.
  int64_t Ret;

  std::forward_list<std::pair<int64_t *, int64_t *>> MemObjStack;
  // Push new memory field and associated object.
  MemObjStack.emplace_front(&Ret, Object);

  do {
    // Pop new memory field and associated object from stack.
    auto P = MemObjStack.front();
    MemObjStack.pop_front();
    int64_t *DestMem = P.first;
    int64_t *Obj = P.second;
```

```

    if (!istraced(Obj)) {
      // Obj is not a heap pointer, so should not be forwarded.
      // Just store its value to its new memory field.
      *DestMem = (int64_t)Obj;
      continue;
    }

    int64_t Size = getObjectSize(Obj);
    assert(Size);

    if (!isSmallMemory(Obj)) {
      // Then the object is an array. We do not forward the object.
      // We only forward its elements if they are non-scalar and
      // they have not been forwarded yet.

      // Index 1 of arrays contain the first element.
      // If first element is non-scalar, then so are the rest.
      if (!isForwarded(Obj) && istraced((int64_t *)Obj[1])) {
        // Array contains non-scalar elements which have not yet been forwarded.
        // Add them to list so they can be copied:
        for (int64_t I = 1; I < Size - 1; ++I) {
          int64_t *Child = (int64_t *)Obj[I];
          MemObjStack.emplace_front(&Obj[I], Child);
        }
      }
      // Store Obj to its new memory location:
      *DestMem = (int64_t)Obj;
      markForwarded(Obj);
      continue;
    }

    if (isForwarded(Obj)) {
      // Obj has already been forwarded.
      // Store its forwarded address to new memory field.
      *DestMem = *Obj;
      continue;
    }

    // If we get here; Obj is a heap allocated pointer and has not yet been
    // forwarded. So we forward is now and store its forwarded address in
    // the new memory field.

    // 1. Allocate memory from inactive heap. This is destination memory for
    // fields from Obj.
    int64_t *Forward = fullMemoryForwardAlloc(Size * sizeof(int64_t));

    // 2. Put fields from Obj in stack. Together with memory destination.
    for (int64_t I = 0; I < Size - 1; ++I) {
      int64_t *Child = (int64_t *)Obj[I];
      MemObjStack.emplace_front(&Forward[I], Child);
    }

    // 3. Insert forward addres in Obj and mark Obj forwarded.
    *Obj = (int64_t)Forward;
    markForwarded(Obj);
```

```

    // 4. Store new memory address of Obj to new memory field.
    *DestMem = (int64_t)Forward;
  } while (!MemObjStack.empty());

  assert(!isFirstGenPointer((int64_t *)Ret));
  return (int64_t *)Ret;
}

static void bigMemoryCopy() {
  auto Block = BigMemoryList.Next;
  auto Prev = &BigMemoryList;
  while (Block != &BigMemoryList) {
    auto New = (BigMemoryNode
        *)fullMemoryForwardAlloc(sizeof(BigMemoryNode) +
                                                        sizeof(int64_t));
    assert(getObjectSize((int64_t *)New));
    Prev->Next = New;
    New->Prev = Prev;
    New->MemoryBlock = Block->MemoryBlock;
    Block = Block->Next;
    Prev = New;
  }
  Prev->Next = &BigMemoryList;
  BigMemoryList.Prev = Prev;
}

static void heapBlockListZero(HeapBlock List) {
  auto B = List;
  B.zero();
  for (B = B.getNext(); B != List; B = B.getNext())
    B.zero();
}

static void firstGenZero() {
  bzero(FirstGenBlock.Block, FIRST_GEN_SIZE);
  FirstGenBlock.Index = 0;
}

static void bigRelease(int64_t *Block) {
  uint64_t Size = getObjectSize(Block) * sizeof(int64_t);
  GCMemoryUsage -= Size;
#ifdef MMAP_ALLOC
  if (munmap(Block - 1, Size)) {
    perror("unable to deallocate memory block");
    error("will not continue with unexpected deallocation error");
  }
#else
  free(Block - 1);
#endif
}

static void sweep() {
  for (auto B = BigMemoryList.Next; B != &BigMemoryList;) {
    auto Block = B;
    auto Mem = B->MemoryBlock;
    B = B->Next;
    if (!isForwarded(Mem)) {
      bigMemoryNodeRemove(Block);
      bigRelease(Mem);
    } else {
      unmarkForwarded(Mem);
```

```
    }
  }
}

static void triggerFullGC(void *PrevFrame) {
// std::cout << " ************** TRIGGER FULL GC **************
    \n";

#ifndef SSML_SHADOW_STACK_GC
  int64_t *PrevF = (int64_t *)PrevFrame;
  assert(isfunc(PrevF[1]));
  FunctionFrame FirstFrame = {&PrevF[2]};
#endif

#ifdef SSML_GC_STATISTICS
  ++FullGCCount;
  uint64_t NumberRoots = 0;
  Timer Tim;
  Tim.start();
#endif // SSML_GC_STATISTICS

  RememberSet.clear();

  // Now the active heap block is beginning of inactive heap.
  ActiveHeapBlock = InactiveHeapBlockList;

#ifdef SSML_SHADOW_STACK_GC
  for (StackEntry *R = llvm_gc_root_chain; R; R = R->Next) {
    assert(R->Map->NumMeta == 0);
    auto Roots = R->Roots;
    for (unsigned I = 1, E = R->Map->NumRoots; I != E; ++I) {
      auto Root = (int64_t *)Roots[I];
      auto Copy = fullMemoryCopy(Root);
      assert(istraced(Copy) ? !isFirstGenPointer(Copy) : true);
      Roots[I] = Copy;

      assert(istraced(Root) && isSmallMemory(Root) ? *Root ==
          (int64_t)Copy
                                                    : true);
      assert(istraced(Root) ? getObjectSize(Root) ==
          getObjectSize(Copy)
                            : true);
#ifdef SSML_GC_STATISTICS
      ++NumberRoots;
#endif // SSML_GC_STATISTICS
    }

    // Mark stack frame as containing pointers to old generation.
    R->Roots[0] = (void *)1;
  }

#else // !defined(SSML_SHADOW_STACK_GC)
  FunctionFrame Frame = FirstFrame;
  for (;;) {
    FrameDescr *Descr = getGCEntry(Frame.getFunction());
    int16_t *Offsets = Descr->RootOffsets;

    for (int64_t I = 1, Count = Descr->RootCount; I < Count; ++I) {
      int16_t RootIndex = Offsets[I];
      int64_t *Root = Frame.getRoot(RootIndex);
      int64_t *Copy = fullMemoryCopy(Root);
      Frame.setRoot(RootIndex, Copy);
```

```
      assert(istraced(Root) && isFirstGenPointer(Root) &&
          isSmallMemory(Root)
                 ? *Root == (int64_t)Copy
                 : true);
      assert(istraced(Root) ? getObjectSize(Root) ==
          getObjectSize(Copy)
                            : true);
#ifdef SSML_GC_STATISTICS
      ++NumberRoots;
#endif // SSML_GC_STATISTICS
    }

    if (isEntryFunction(Frame.getFunction()))
      break;
    // Mark stack frame as containing pointers to old generation.
    Frame.setRoot(Offsets[0], (int64_t *)1);
    Frame = Frame.nextFrame(Descr);
  }

#endif // SSML_SHADOW_STACK_GC

  bigMemoryCopy();
  sweep();

  std::swap(ActiveHeapBlockList, InactiveHeapBlockList);
  heapBlockListZero(InactiveHeapBlockList);
  firstGenZero();

  if (ActiveHeapBlock.getNext() == ActiveHeapBlockList &&
      !activeHasCapacity(MemoryIncrease / 2) &&
      GCMemoryUsage + MemoryIncrease <= MAX_ALLOC_SIZE)
    increaseMemory();

#ifdef SSML_GC_STATISTICS
  TotalGCRoots += NumberRoots;

  uint64_t Time = Tim.time();
  if (Time > WhorstGCTime)
    WhorstGCTime = Time;
  if (Time < BestGCTime)
    BestGCTime = Time;
  TotalGCTime += Time;
  ++TriggerGCCount;
#endif // SSML_GC_STATISTICS
}

static int64_t *doHeapBlockListAllocate(int64_t N) {
  int64_t *Mem = heapBlockListAllocate1(N, ActiveHeapBlockList);
  assert(Mem);
  assert(Mem[-1]);
  return Mem;
}

static int64_t *heapBlockListAllocate(int64_t N) {
  return doHeapBlockListAllocate(N);
}

static int64_t *firstGenAllocateWords(int64_t Words) {
  assert(FirstGenBlock.Index + Words <= FIRST_GEN_SIZE /
      sizeof(int64_t));
  int64_t *Ret = &FirstGenBlock.Block[FirstGenBlock.Index];
```

```
  FirstGenBlock.Index += Words;
  *Ret = Words;
  ++Ret;
  markSmallMemory(Ret);
  return Ret;
}

static bool isFirstGenPointer(int64_t *Obj) {
  uint64_t P = (uint64_t)Obj;
  uint64_t Begin = (uint64_t)FirstGenBlock.Block;
  uint64_t End = (uint64_t)FirstGenBlock.Block + FIRST_GEN_SIZE;
  return P >= Begin && P < End;
}

static int64_t *firstGenMemoryCopy(int64_t *Object) {
  // Ret is new memory field of Object.
  int64_t Ret;

  std::forward_list<std::pair<int64_t *, int64_t *>> MemObjStack;
  // Push new memory field and associated object.
  MemObjStack.emplace_front(&Ret, Object);

  do {
    // Pop new memory field and associated object from stack.
    auto P = MemObjStack.front();
    MemObjStack.pop_front();
    int64_t *DestMem = P.first;
    int64_t *Obj = P.second;

    if (!istraced(Obj)) {
      // Obj is not a heap pointer, so should not be forwarded.
      // Just store its value to its new memory field.
      *DestMem = (int64_t)Obj;
      continue;
    }

    if (!isFirstGenPointer(Obj)) {
      // Obj is not pointer to first generation.
      // Thus we don't forward it.
      *DestMem = (int64_t)Obj;
      continue;
    }

    if (isForwarded(Obj)) {
      // Obj has already been forwarded.
      // Store its forwarded address to new memory field.
      *DestMem = *Obj;
      continue;
    }

    // If we get here; Obj is a heap allocated pointer from first generation.
    // It has not yet been forwarded, so we forward is now and store its
    // forwarded address in the new memory field.

    int64_t Size = getObjectSize(Obj);
    assert(Size);
    assert(Size < BigAllocateThreshold);

    // Put fields from Obj in stack. Together with memory destination.
```

```
    int64_t *Forward = heapBlockListAllocate(Size * sizeof(int64_t));
    for (int64_t I = 0; I < Size - 1; ++I) {
      int64_t *Child = (int64_t *)Obj[I];
      MemObjStack.emplace_front(&Forward[I], Child);
    }

    // Insert forward addres in Obj and mark Obj forwarded.
    *Obj = (int64_t)Forward;
    markForwarded(Obj);

    // Store new memory address of Obj to new memory field.
    *DestMem = (int64_t)Forward;
  } while (!MemObjStack.empty());

  return (int64_t *)Ret;
}

static void firstGenRememberedCopy(int64_t *Obj) {
  assert(istraced(Obj));
  assert(!isFirstGenPointer(Obj));

  int64_t Size = getObjectSize(Obj);
  assert(Size);

  for (int64_t I = 0; I < Size - 1; ++I) {
    int64_t *Child = (int64_t *)Obj[I];
    Obj[I] = (int64_t)firstGenMemoryCopy(Child);
  }
}

static void triggerFirstGenGC(void *PrevFrame) {
// std::cout << " ************** FIRST GEN GC ************** \n";
#ifndef SSML_SHADOW_STACK_GC
  int64_t *PrevF = (int64_t *)PrevFrame;
  assert(isfunc(PrevF[1]));
  FunctionFrame FirstFrame = {&PrevF[2]};
#endif

#ifdef SSML_GC_STATISTICS
  ++FirstGenGCCount;
  uint64_t NumberRoots = 0;
  Timer Tim;
  Tim.start();
#endif // SSML_GC_STATISTICS

  for (auto P : RememberSet)
    firstGenRememberedCopy(P);
  RememberSet.clear();

#ifdef SSML_SHADOW_STACK_GC
  for (StackEntry *R = llvm_gc_root_chain; R; R = R->Next) {
    assert(R->Map->NumMeta == 0);
    auto Roots = R->Roots;
    for (unsigned I = 1, E = R->Map->NumRoots; I != E; ++I) {
      auto Root = (int64_t *)Roots[I];
      auto Copy = firstGenMemoryCopy(Root);
      Roots[I] = Copy;

      assert(istraced(Root) && isFirstGenPointer(Root) &&
          isSmallMemory(Root)
                 ? *Root == (int64_t)Copy
```

```
                    : true);
      assert(istraced(Root) ? getObjectSize(Root) == getObjectSize(Copy)
                                         : true);
#ifdef SSML_GC_STATISTICS
      ++NumberRoots;
#endif // SSML_GC_STATISTICS
    }

    if (R->Roots[0]) {
      assert(R->Roots[0] == (void *)1);
      // Is this stack frame is marked as old stack frame; break now.
      break;
    }
    // Mark stack frame as containing pointers to old generation.
    R->Roots[0] = (void *)1;
  }

#else // !defined(SSML_SHADOW_STACK_GC)
  FunctionFrame Frame = FirstFrame;
  for (;;) {
    FrameDescr *Descr = getGCEntry(Frame.getFunction());
    int16_t *Offsets = Descr->RootOffsets;
    for (int64_t I = 1, Count = Descr->RootCount; I < Count; ++I) {
      int16_t RootIndex = Offsets[I];
      int64_t *Root = Frame.getRoot(RootIndex);
      int64_t *Copy = firstGenMemoryCopy(Root);
      Frame.setRoot(RootIndex, Copy);

      assert(istraced(Root) && isFirstGenPointer(Root) && isSmallMemory(Root)
                    ? *Root == (int64_t)Copy
                    : true);
      assert(istraced(Root) ? getObjectSize(Root) == getObjectSize(Copy)
                                         : true);
#ifdef SSML_GC_STATISTICS
      ++NumberRoots;
#endif // SSML_GC_STATISTICS
    }

    if (Frame.getRoot(Offsets[0])) {
      assert(Frame.getRoot(Offsets[0]) == (int64_t *)1);
      // Is this stack frame is marked as old stack frame; break now.
      break;
    }
    // Mark stack frame as containing pointers to old generation.
    Frame.setRoot(Offsets[0], (int64_t *)1);
    Frame = Frame.nextFrame(Descr);
  }
#endif // SSML_SHADOW_STACK_GC

  firstGenZero();

#ifdef SSML_GC_STATISTICS
  TotalGCRoots += NumberRoots;

  uint64_t Time = Tim.time();
  if (Time > WhorstGCTime)
    WhorstGCTime = Time;
```

```
  if (Time < BestGCTime)
    BestGCTime = Time;
  TotalGCTime += Time;
  ++TriggerGCCount;
#endif // SSML_GC_STATISTICS
}

static int64_t *firstGenAllocate(int64_t N, void *PrevFrame) {
  int64_t Words = N / sizeof(int64_t);
  if (FirstGenBlock.Index + Words <= FIRST_GEN_SIZE /
      sizeof(int64_t)) {
    return firstGenAllocateWords(Words);
  }
  if (activeHasFirstGenCapacity()) {
    triggerFirstGenGC(PrevFrame);
  } else {
    triggerFullGC(PrevFrame);
    if (!activeHasFirstGenCapacity())
      increaseMemory();
  }
  assert(FirstGenBlock.Index == 0);
  return firstGenAllocateWords(Words);
}

static int64_t *bigAllocate(int64_t N, void *PrevFrame) {
  assert(N % sizeof(int64_t) == 0);

  N = N + sizeof(int64_t);

  static const uint64_t NodeSize = sizeof(BigMemoryNode) +
      sizeof(int64_t);

  if (N + GCMemoryUsage > MAX_ALLOC_SIZE ||
      !activeHasCapacity(NodeSize))
    triggerFullGC(PrevFrame);

  if (!activeHasCapacity(NodeSize))
    increaseMemory();

  // Allocate node for holding big memory block.
  BigMemoryNode *Node = (BigMemoryNode
      *)heapBlockListAllocate(NodeSize);
  bigMemoryNodeAppend(Node);

  // Allocate memory. Need room for size of object; add size of
      int64_t.
  int64_t *Mem = newBlock(N);
  *Mem = N / sizeof(int64_t);

  Node->MemoryBlock = ++Mem;
  return Mem;
}

extern "C" void ssmlGCInit() {
  int64_t *Block = newBlock(MemoryIncrease);
  ActiveHeapBlockList = HeapBlock(Block, Block, Block);

  ActiveHeapBlock = ActiveHeapBlockList;

  Block = newBlock(MemoryIncrease);
  InactiveHeapBlockList = HeapBlock(Block, Block, Block);

```

```
  FirstGenBlock.Block = newBlock(FIRST_GEN_SIZE);
  setFirstGenBegin((uint64_t)FirstGenBlock.Block);
  setFirstGenEnd((uint64_t)FirstGenBlock.Block + FIRST_GEN_SIZE);
}

extern "C" int64_t *allocateFrame(int64_t N, void *Frame) {
  assert(N % sizeof(int64_t) == 0);

  if (!N)
    return nullptr;

  // Need to store size of object also; add one more word to
      memory size.
  N = N + sizeof(int64_t);
  if (N <= BigAllocateThreshold) {
    auto Ret = firstGenAllocate(N, Frame);
    return Ret;
  }
  return bigAllocate(N, Frame);
}

extern "C" int64_t *allocate(int64_t N) {
  return allocateFrame(N, __builtin_frame_address(0));
}

#endif // SSML_GENERATIONAL_GC
```

Listing 6: runtime/CopyingGC.cpp

```
#ifdef SSML_COPYING_GC

#include "GC.h"
#include "ssml/Common/Timer.h"

#include <strings.h>
#include <iostream>
#include <stdint.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <forward_list>

#define MMAP_ALLOC

using namespace ssml;

extern "C" void error(const char *);

static uint64_t TriggerGCCount;
static uint64_t TotalGCTime;
static uint64_t BestGCTime = UINT64_MAX;
static uint64_t WhorstGCTime;
static uint64_t TotalGCRoots;
static uint64_t GCMemoryUsage;

extern "C" void ssmlGCPrintStatistics() {
  std::cout << "****** Copying GC Statistics ******\n";
#ifdef SSML_SHADOW_STACK_GC
  std::cout << "Used Shadow Stack GC strategy\n";
#else
  std::cout << "Used Binary Map GC strategy\n";
#endif
```

```cpp
  std::cout << "Total GCs:             " << TriggerGCCount << '\n';
  if (!TriggerGCCount)
    return;

  double Tot = (double)TotalGCTime / 1000;
  double Avg = ((double)TotalGCTime / TriggerGCCount) / 1000;
  double Whorst = (double)WhorstGCTime / 1000;
  double Best = (double)BestGCTime / 1000;
  double AvgRoots = (double)TotalGCRoots / TriggerGCCount;
  uint64_t Avail = MAX_ALLOC_SIZE - GCMemoryUsage;

  std::cout << "Total GC Time:         " << Tot << "ms\n";
  std::cout << "Average GC Time:       " << Avg << "ms\n";
  std::cout << "Whorst GC Time:        " << Whorst << "ms\n";
  std::cout << "Best GC Time:          " << Best << "ms\n";
  std::cout << "Average number Roots:  " << AvgRoots << '\n';
  std::cout << "Total Memory:          " << MAX_ALLOC_SIZE << "B\n";
  std::cout << "GC Memory Usage:       " << GCMemoryUsage << "B\n";
  std::cout << "Total Free Memory:     " << Avail << "B\n";
  std::cout << "***********************************" << std::endl;
}

// When allocating more memory than this, use a mark sweep algorithm instead.
static const int64_t BigAllocateThreshold = 16384; // 2^14 bytes.
// static const int64_t BigAllocateThreshold = 1024;
// Both heaps are increased by MemoryIncrease when more memory is needed.
static const int64_t MemoryIncrease = 2097152 * 32; // 2 * 32 MB.

class HeapBlock {
private:
  // Block[PrevIndex] pointer to prev block.
  // Block[NextIndex] pointer to next block.
  // Block[FreeIndex] index to next free block.
  int64_t *Block;

public:
  static const uint64_t PrevIndex = 0;
  static const uint64_t NextIndex = 1;
  static const uint64_t FreeIndex = 2;
  static const uint64_t InitialFreeIndex = 3;
  static const int64_t EndIndex = MemoryIncrease / sizeof(int64_t);
  static_assert(MemoryIncrease >= 4 * BigAllocateThreshold,
                "MemoryIncrease too small");
  static_assert(MemoryIncrease % sizeof(int64_t) == 0, "bad MemoryIncrease");

public:
  explicit HeapBlock(int64_t *Block, HeapBlock Prev, HeapBlock Next);
  HeapBlock(int64_t *InitializedBlock);

  int64_t *allocate(int64_t Size);

  operator bool();

  HeapBlock getPrev() { return (int64_t *)this->Block[PrevIndex]; };
  HeapBlock getNext() { return (int64_t *)this->Block[NextIndex]; };
```

```
  void setPrev(HeapBlock B) { this->Block[PrevIndex] = (int64_t)B.Block; };
  void setNext(HeapBlock B) { this->Block[NextIndex] = (int64_t)B.Block; };

  void zero();

  bool operator==(HeapBlock Oth) { return this->Block == Oth.Block; }
  bool operator!=(HeapBlock Oth) { return !(*this == Oth); }

  uint64_t getNumberFreeBytes() {
    return MemoryIncrease - this->Block[FreeIndex] * sizeof(int64_t);
  }
};

HeapBlock::HeapBlock(int64_t *Block, HeapBlock Prev, HeapBlock Next)
    : Block(Block) {
  this->Block[PrevIndex] = (int64_t)Prev.Block;
  this->Block[NextIndex] = (int64_t)Next.Block;
  this->Block[FreeIndex] = InitialFreeIndex;
}

HeapBlock::HeapBlock(int64_t *InitializedBlock) : Block(InitializedBlock) {}

int64_t *HeapBlock::allocate(int64_t WordSize) {
  assert(WordSize > 0);
  int64_t NextIndex = this->Block[HeapBlock::FreeIndex] + WordSize;
  if (NextIndex > HeapBlock::EndIndex)
    return nullptr;
  int64_t PrevIndex = this->Block[HeapBlock::FreeIndex];
  this->Block[FreeIndex] = NextIndex;
  return &this->Block[PrevIndex];
}

void HeapBlock::zero() {
  assert(this->Block);
  bzero(this->Block + 2, MemoryIncrease - 2 * sizeof(int64_t));
  this->Block[FreeIndex] = HeapBlock::InitialFreeIndex;
}

HeapBlock::operator bool() { return this->Block; }

struct BigMemoryNode {
  BigMemoryNode *Prev;
  BigMemoryNode *Next;
  int64_t *MemoryBlock;
};

BigMemoryNode BigMemoryList = {&BigMemoryList, &BigMemoryList, nullptr};

HeapBlock ActiveHeapBlockList(0);
HeapBlock InactiveHeapBlockList(0);
HeapBlock ActiveHeapBlock(0);

static bool activeHasCapacity(uint64_t Bytes) {
  assert(Bytes < MemoryIncrease - sizeof(int64_t) * 64);
```

```
  if (ActiveHeapBlock.getNumberFreeBytes() >= Bytes)
    return true;
  if (ActiveHeapBlock.getNext() != ActiveHeapBlockList)
    return true;
  return false;
}

static const uint64_t SmallMemoryBit = (uint64_t)1 << 63;
static void markSmallMemory(int64_t *Obj) { Obj[-1] |= SmallMemoryBit; }

static bool isSmallMemory(int64_t *Obj) { return Obj[-1] & SmallMemoryBit; }

static const uint64_t ForwardBit = (uint64_t)1 << 62;
static void markForwarded(int64_t *Obj) { Obj[-1] |= ForwardBit; }

static void unmarkForwarded(int64_t *Obj) { Obj[-1] &= ~ForwardBit; }

static bool isForwarded(int64_t *Obj) { return Obj[-1] & ForwardBit; }

static int64_t getObjectSize(int64_t *Obj) {
  return Obj[-1] & ~(SmallMemoryBit | ForwardBit);
}

static void bigMemoryNodeAppend(BigMemoryNode *Node, BigMemoryNode *Prev,
                                BigMemoryNode *Next) {
  Node->Prev = Prev;
  Node->Next = Next;
  Prev->Next = Node;
  Next->Prev = Node;
}

static void bigMemoryNodeRemove(BigMemoryNode *Node) {
  assert(Node != &BigMemoryList);
  Node->Prev->Next = Node->Next;
  Node->Next->Prev = Node->Prev;
}

static void bigMemoryNodeAppend(BigMemoryNode *Node) {
  bigMemoryNodeAppend(Node, BigMemoryList.Prev, &BigMemoryList);
}

static int64_t *heapBlockListAllocate1(int64_t N, HeapBlock ListHead) {
  assert(N % sizeof(int64_t) == 0);
  N = N / sizeof(int64_t);
  assert(N > 0);

  int64_t *Mem = ActiveHeapBlock.allocate(N);
  if (Mem)
    goto out;

  ActiveHeapBlock = ActiveHeapBlock.getNext();
  if (ActiveHeapBlock == ListHead)
    return nullptr;

  Mem = ActiveHeapBlock.allocate(N);
  assert(Mem);
```

```
out:
  Mem[0] = N | SmallMemoryBit;
  return Mem + 1;
}

static int64_t *newBlock(uint64_t Size) {
  GCMemoryUsage += Size;
  if (GCMemoryUsage > MAX_ALLOC_SIZE)
    error("out of memory");

#ifdef MMAP_ALLOC
  void *Mem = mmap(nullptr, Size, PROT_READ | PROT_WRITE,
                   MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
  if (Mem == MAP_FAILED) {
    perror("unable to allocate memory");
    error("out of memory");
  }
#else
  void *Mem = calloc(1, Size);
  if (!Mem) {
    perror("unable to allocate memory");
    error("out of memory");
  }
#endif

  return (int64_t *)Mem;
}

static void increaseMemory() {
  auto ActivePrev = ActiveHeapBlockList.getPrev();
  auto Block =
      HeapBlock(newBlock(MemoryIncrease), ActivePrev,
          ActiveHeapBlockList);
  ActivePrev.setNext(Block);
  ActiveHeapBlockList.setPrev(Block);

  auto InactivePrev = InactiveHeapBlockList.getPrev();
  Block =
      HeapBlock(newBlock(MemoryIncrease), InactivePrev,
          InactiveHeapBlockList);
  InactivePrev.setNext(Block);
  InactiveHeapBlockList.setPrev(Block);
}

static int64_t *memoryForwardAlloc(uint64_t NumBytes) {
  int64_t *Forward = heapBlockListAllocate1(NumBytes,
      InactiveHeapBlockList);
  if (!Forward) {
    // This actually can happen if the copied objects are put in
        the new
    // memory spaces in a less space efficient way.
    increaseMemory();
    ActiveHeapBlock = InactiveHeapBlockList.getPrev();
    Forward = heapBlockListAllocate1(NumBytes,
        InactiveHeapBlockList);
    assert(Forward);
  }
  assert(Forward[-1]);
  return Forward;
}
```

```
static int64_t *memoryCopy(int64_t *Object) {
  // Ret is new memory field of Object.
  int64_t Ret;

  std::forward_list<std::pair<int64_t *, int64_t *>> MemObjStack;
  // Push new memory field and associated object.
  MemObjStack.emplace_front(&Ret, Object);

  do {
    // Pop new memory field and associated object from stack.
    auto P = MemObjStack.front();
    MemObjStack.pop_front();
    int64_t *DestMem = P.first;
    int64_t *Obj = P.second;

    if (!istraced(Obj)) {
      // Obj is not a heap pointer, so should not be forwarded.
      // Just store its value to its new memory field.
      *DestMem = (int64_t)Obj;
      continue;
    }

    int64_t Size = getObjectSize(Obj);
    assert(Size);

    if (!isSmallMemory(Obj)) {
      // Then the object is an array. We do not forward the object.
      // We only forward its elements if they are non-scalar and
      // they have not been forwarded yet.

      // Index 1 of arrays contain the first element.
      // If first element is non-scalar, then so are the rest.
      if (!isForwarded(Obj) && istraced((int64_t *)Obj[1])) {
        // Array contains non-scalar elements which have not yet been forwarded.
        // Add them to list so they can be copied:
        for (int64_t I = 1; I < Size - 1; ++I) {
          int64_t *Child = (int64_t *)Obj[I];
          MemObjStack.emplace_front(&Obj[I], Child);
        }
      }
      // Store Obj to its new memory location:
      *DestMem = (int64_t)Obj;
      markForwarded(Obj);
      continue;
    }

    if (isForwarded(Obj)) {
      // Obj has already been forwarded.
      // Store its forwarded address to new memory field.
      *DestMem = *Obj;
      continue;
    }

    // If we get here; Obj is a heap allocated pointer and has not yet been
    // forwarded. So we forward is now and store its forwarded address in
    // the new memory field.

    // 1. Allocate memory from inactive heap. This is destination memory for
```

```
    // fields from Obj.
    int64_t *Forward = memoryForwardAlloc(Size * sizeof(int64_t));

    // 2. Put fields from Obj in stack. Together with memory destination.
    for (int64_t I = 0; I < Size - 1; ++I) {
      int64_t *Child = (int64_t *)Obj[I];
      MemObjStack.emplace_front(&Forward[I], Child);
    }

    // 3. Insert forward addres in Obj and mark Obj forwarded.
    *Obj = (int64_t)Forward;
    markForwarded(Obj);

    // 4. Store new memory address of Obj to new memory field.
    *DestMem = (int64_t)Forward;
  } while (!MemObjStack.empty());

  return (int64_t *)Ret;
}

static void bigMemoryCopy() {
  auto Block = BigMemoryList.Next;
  auto Prev = &BigMemoryList;
  while (Block != &BigMemoryList) {
    auto New = (BigMemoryNode *)memoryForwardAlloc(sizeof(BigMemoryNode) +
                                                   sizeof(int64_t));
    Prev->Next = New;
    New->Prev = Prev;
    New->MemoryBlock = Block->MemoryBlock;
    Block = Block->Next;
    Prev = New;
  }
  Prev->Next = &BigMemoryList;
  BigMemoryList.Prev = Prev;
}

static void heapBlockListZero(HeapBlock List) {
  auto B = List;
  B.zero();
  for (B = B.getNext(); B != List; B = B.getNext())
    B.zero();
}

static void bigRelease(int64_t *Block) {
  uint64_t Size = getObjectSize(Block) * sizeof(int64_t);
  GCMemoryUsage -= Size;
#ifdef MMAP_ALLOC
  if (munmap(Block - 1, Size)) {
    perror("unable to deallocate memory block");
    error("will not continue with unexpected deallocation error");
  }
#else
  free(Block - 1);
#endif
}

static void sweep() {
  for (auto B = BigMemoryList.Next; B != &BigMemoryList;) {
    auto Block = B;
    auto Mem = B->MemoryBlock;
```

```
    B = B->Next;
    if (!isForwarded(Mem)) {
      bigMemoryNodeRemove(Block);
      bigRelease(Mem);
    } else {
      unmarkForwarded(Mem);
    }
  }
}

static void triggerGC(FunctionFrame FirstFrame) {
// std::cout << " ************** TRIGGER GC ************** \n";
#ifdef SSML_GC_STATISTICS
  uint64_t NumberRoots = 0;
  Timer Tim;
  Tim.start();
#endif // SSML_GC_STATISTICS

  // Now the active heap block is beginning of inactive heap.
  ActiveHeapBlock = InactiveHeapBlockList;

#ifdef SSML_SHADOW_STACK_GC
  for (StackEntry *R = llvm_gc_root_chain; R; R = R->Next) {
    assert(R->Map->NumMeta == 0);
    auto Roots = R->Roots;
    for (unsigned I = 0, E = R->Map->NumRoots; I != E; ++I) {
      auto Root = (int64_t *)Roots[I];
      auto Copy = memoryCopy(Root);
      Roots[I] = Copy;

      assert(istraced(Root) && isSmallMemory(Root) ? *Root ==
          (int64_t)Copy
                                                    : true);
      assert(istraced(Root) ? getObjectSize(Root) ==
          getObjectSize(Copy)
                            : true);
#ifdef SSML_GC_STATISTICS
      ++NumberRoots;
#endif // SSML_GC_STATISTICS
    }
  }
#else // !defined(SSML_SHADOW_STACK_GC)
  FunctionFrame Frame = FirstFrame;
  for (;;) {
    FrameDescr *Descr = getGCEntry(Frame.getFunction());
    int16_t *Offsets = Descr->RootOffsets;

    for (int64_t I = 0, Count = Descr->RootCount; I < Count; ++I) {
      int16_t RootIndex = Offsets[I];
      int64_t *Root = Frame.getRoot(RootIndex);
      int64_t *Copy = memoryCopy(Root);
      Frame.setRoot(RootIndex, Copy);

      assert(istraced(Root) && isSmallMemory(Root) ? *Root ==
          (int64_t)Copy
                                                    : true);
#ifdef SSML_GC_STATISTICS
      ++NumberRoots;
#endif // SSML_GC_STATISTICS
    }

    if (isEntryFunction(Frame.getFunction()))
```

```
      break;
    Frame = Frame.nextFrame(Descr);
  }
#endif // SSML_SHADOW_STACK_GC

  bigMemoryCopy();
  sweep();

  std::swap(ActiveHeapBlockList, InactiveHeapBlockList);
  heapBlockListZero(InactiveHeapBlockList);

  if (ActiveHeapBlock.getNext() == ActiveHeapBlockList &&
      !activeHasCapacity(MemoryIncrease / 2) &&
      GCMemoryUsage + MemoryIncrease <= MAX_ALLOC_SIZE)
    increaseMemory();

#ifdef SSML_GC_STATISTICS
  TotalGCRoots += NumberRoots;

  uint64_t Time = Tim.time();
  if (Time > WhorstGCTime)
    WhorstGCTime = Time;
  if (Time < BestGCTime)
    BestGCTime = Time;
  TotalGCTime += Time;
  ++TriggerGCCount;
#endif // SSML_GC_STATISTICS
}

static int64_t *doHeapBlockListAllocate(int64_t N, void
    *PrevFrame) {
  int64_t *Mem = heapBlockListAllocate1(N, ActiveHeapBlockList);
  if (Mem) {
    assert(Mem[-1]);
    return Mem;
  }

  // If we get here there is not enough allocated memory left.
  // Trigger a GC and see if we get enough new memory.
  int64_t *Prev = (int64_t *)PrevFrame;
#ifndef SSML_SHADOW_STACK_GC
  assert(isfunc(Prev[1]));
#endif
  triggerGC({&Prev[2]});

  // Now after GC, we might have some free memory.
  Mem = heapBlockListAllocate1(N, ActiveHeapBlockList);
  if (Mem) {
    assert(Mem[-1]);
    return Mem;
  }

  // Still, after GC there is not enough memory.
  // Allocate a new memory block.
  increaseMemory();
  ActiveHeapBlock = ActiveHeapBlockList.getPrev();

  // Now, we must have memory. Otherwise the program should have
      crashed.
  Mem = heapBlockListAllocate1(N, ActiveHeapBlockList);
  assert(Mem);
  assert(Mem[-1]);
```

```
  return Mem;
}

static int64_t *heapBlockListAllocate(int64_t N, void *PrevFrame) {
  auto Ret = doHeapBlockListAllocate(N, PrevFrame);
  markSmallMemory(Ret);
  return Ret;
}

static int64_t *bigAllocate(int64_t N, void *PrevFrame) {
  assert(N % sizeof(int64_t) == 0);

  N = N + sizeof(int64_t);

  if (N + GCMemoryUsage > MAX_ALLOC_SIZE) {
    int64_t *Prev = (int64_t *)PrevFrame;
#ifndef SSML_SHADOW_STACK_GC
    assert(isfunc(Prev[1]));
#endif
    triggerGC({&Prev[2]});
  }

  // Allocate node for holding big memory block.
  BigMemoryNode *Node =
      (BigMemoryNode *)allocateFrame(sizeof(BigMemoryNode), PrevFrame);
  bigMemoryNodeAppend(Node);

  int64_t *Mem = newBlock(N);
  *Mem = N / sizeof(int64_t);

  Node->MemoryBlock = ++Mem;
  return Mem;
}

extern "C" void ssmlGCInit() {
  int64_t *Block = newBlock(MemoryIncrease);
  ActiveHeapBlockList = HeapBlock(Block, Block, Block);

  ActiveHeapBlock = ActiveHeapBlockList;

  Block = newBlock(MemoryIncrease);
  InactiveHeapBlockList = HeapBlock(Block, Block, Block);
}

extern "C" int64_t *allocateFrame(int64_t N, void *Frame) {
  assert(N % sizeof(int64_t) == 0);

  if (!N)
    return nullptr;

  // Need to store size of object also; add one more word to memory size.
  N = N + sizeof(int64_t);
  if (N <= BigAllocateThreshold) {
    auto Ret = heapBlockListAllocate(N, Frame);
    return Ret;
  }
  return bigAllocate(N, Frame);
}

extern "C" int64_t *allocate(int64_t N) {
```

```
  return allocateFrame(N, __builtin_frame_address(0));
}

#endif // SSML_COPYING_GC
```

Listing 7: runtime/MarkSweepAlloc.cpp

```
#ifdef SSML_MARK_SWEEP_GC

#include "GC.h"
#include "ssml/Common/Timer.h"

#include <stdlib.h>
#include <assert.h>
#include <stdint.h>
#include <errno.h>
#include <sys/mman.h>
#include <strings.h>

#include <algorithm>
#include <iostream>
#include <set>
#include <forward_list>

#pragma GCC diagnostic ignored "-pedantic"

using namespace ssml;

#define MMAP_ALLOC

#define UNUSED __attribute__((unused))

extern "C" void ssmlGCInit() {}

static uint64_t BigAllocSize;
static uint64_t NormalAllocSize;
static uint64_t NormalAllocUsage;
static const uint64_t GCTriggerIncrease = MAX_ALLOC_SIZE / 16;
static uint64_t GCTriggerPoint = GCTriggerIncrease;

static uint64_t TriggerGCCount;
static uint64_t TotalGCTime;
static uint64_t BestGCTime = UINT64_MAX;
static uint64_t WhorstGCTime;
static uint64_t TotalGCRoots;

extern "C" void ssmlGCPrintStatistics() {
  std::cout << "**** Mark Sweep: GC Statistics ****\n";
#ifdef SSML_SHADOW_STACK_GC
  std::cout << "Used Shadow Stack GC strategy\n";
#else
  std::cout << "Used Binary Map GC strategy\n";
#endif
  std::cout << "Total GCs:             " << TriggerGCCount << '\n';
  if (!TriggerGCCount)
    return;

  double Tot = (double)TotalGCTime / 1000;
  double Avg = ((double)TotalGCTime / TriggerGCCount) / 1000;
  double Whorst = (double)WhorstGCTime / 1000;
  double Best = (double)BestGCTime / 1000;
```

```cpp
  double AvgRoots = (double)TotalGCRoots / TriggerGCCount;

  uint64_t Avail = MAX_ALLOC_SIZE - BigAllocSize - GCTriggerPoint;

  std::cout << "Total GC Time:          " << Tot << "ms\n";
  std::cout << "Average GC Time:        " << Avg << "ms\n";
  std::cout << "Whorst GC Time:         " << Whorst << "ms\n";
  std::cout << "Best GC Time:           " << Best << "ms\n";
  std::cout << "Average number Roots:   " << AvgRoots << '\n';
  std::cout << "Total Memory:           " << MAX_ALLOC_SIZE << "B\n";
  std::cout << "GC Memory Usage:        " << NormalAllocSize << "B\n";
  std::cout << "GC Threshold:           " << GCTriggerPoint << "B\n";
  std::cout << "Big Memory Usage:       " << BigAllocSize << "B\n";
  std::cout << "Total Free Memory:      " << Avail << "B\n";
  std::cout << "***********************************" << std::endl;
}

namespace {
class MemoryBlock {
  friend MemoryBlock splitFreeList(int64_t FreeListIndex);
  friend MemoryBlock popFreeList(int64_t FreeListIndex);
  friend void removeFreeList(int64_t FreeListIndex, MemoryBlock B);
  friend void pushFreeList(int64_t FreeListIndex, MemoryBlock B);

public:
  static const int64_t ObjectMarkBit = (int64_t)1 << 63;
  static const int64_t ObjectMarkMask = ~ObjectMarkBit;
  static const int64_t IsAllocatedBit = (int64_t)1 << 62;
  static const int64_t IsAllocatedMask = ~IsAllocatedBit;
  // SizeBitmask assumes FreeListMaxIndex <= 20.
  static const int64_t SizeBitmask = 0x7fffff;
  static const int OffsetShift = std::ceil(std::log2(SizeBitmask));
  static const int64_t OffsetMask = 0x7fffffffffffffff;
  // Bits indices 61-23 are free for offset bytes.
  // Smallest block is 3 words + 1 word for size and offset.
  static const int MinimumWordSize = 3;

private:
  // When block is in a free list. Block[1] is prev block in the free list
  // and Block[2] is next block in free list.
  int64_t *Block;

private:
  // Split this block into two equally sized blocks.
  // Assumes the block is nulled.
  std::pair<MemoryBlock, MemoryBlock> split();
  // Set pointer to prev block in free list.
  void setPrev(MemoryBlock N);
  // Set pointer to next block in free list.
  void setNext(MemoryBlock N);

public:
  // Initialize with nullptr.
  explicit MemoryBlock();
  // Assumes Block is an initialized memory block, already contains word size
  // and byte offset.
  explicit MemoryBlock(int64_t *Block);
```

```
  // Assumes Block is nulled. It will override the size/offset field.
  explicit MemoryBlock(int64_t *Block, int64_t WordSize, int64_t ByteOffset);
  // Returns number of words in memory words
  // (excluding the first word in the block).
  int64_t getWordSize();
  // Returns byte offset of this block to the base block.
  int64_t getOffset();
  // Address of first usable word of this block. (&this->Block[1]).
  int64_t *get();
  // Address of this block.
  int64_t *getRaw();
  // Get prev memory block in free list.
  MemoryBlock getPrev();
  // Get next memory block in free list.
  MemoryBlock getNext();
  // Returns true if all usable memory is zeroed. Used for debugging.
  bool isNulled();

  operator bool();
};
} // End annonymous namespace.

// Mark object starting at Obj[-1] as visited. Return word size of object.
static int64_t mark(int64_t *Obj);

// Mark object starting at Obj[-1] as not visited.
static void unmark(int64_t *Obj);

// Return whether object starting at Obj[-1] is marked as visited.
static bool ismarked(int64_t *Obj);

// Recursively mark object starting at Root[-1] and all objects directly or
// indirectly pointed to.
static void markall(int64_t *Root);

// Free heap allocated object starting at Obj[-1]. Return memory block which
// have been freed. The returned block is not the same block as Obj when
// freeing of Obj results in a merge with its buddy block(s).
static MemoryBlock release(int64_t *Obj);

// Sweep phase in mark & sweep algorithm.
static void sweep();

// Free list index 16 is for 2MB blocks.
static const int64_t FreeListMaxIndex = 16;
static_assert(FreeListMaxIndex < 20, "FreeLists size too big");

static const int64_t FreeListCount = FreeListMaxIndex + 1;
// Index 0 contains blocks of size 32 bytes.
// Index 1 contains blocks of size 64 bytes.
// ...
// Index FreeListMaxIndex contains blocks of size MemoryIncrease bytes.
static MemoryBlock FreeLists[FreeListCount];
// Index 0 of each block contains the block size in words.
```

```
// Index 1 contains address of the base block.
// Index 2 is pointer to next block in the free list.

// Memory increase for each base block allocation.
static const int64_t MemoryIncrease = 32 * std::pow(2, FreeListMaxIndex);

std::pair<MemoryBlock, MemoryBlock> MemoryBlock::split() {
  auto Size = this->getWordSize() + 1;
  auto Off = this->getOffset();
  assert(this->getWordSize() > MemoryBlock::MinimumWordSize);

  auto NewSize = Size >> 1;
  MemoryBlock First(this->Block, NewSize - 1, Off);

  auto NewOff = Off + NewSize * sizeof(int64_t);
  MemoryBlock Second(&this->Block[NewSize], NewSize - 1, NewOff);

  assert(First.getWordSize() == Size / 2 - 1);
  assert(First.getWordSize() == Second.getWordSize());
  assert(First.getWordSize() >= MemoryBlock::MinimumWordSize);
  assert(First.isNulled());
  assert(Second.isNulled());

  return {First, Second};
}

// Merge memory block A and memory block B into 1 block.
// Assumes the blocks are not currently in a free list
// And that the blocks are nulled.
static MemoryBlock memoryBlockMergeNulled(MemoryBlock A, MemoryBlock B) {
  assert(A.getWordSize() >= MemoryBlock::MinimumWordSize);
  assert(A.getWordSize() == B.getWordSize());

  if ((uint64_t)A.getRaw() > (uint64_t)B.getRaw())
    std::swap(A, B);

  assert((uint64_t)A.getRaw() + (A.getWordSize() + 1) * sizeof(int64_t) ==
         (uint64_t)B.getRaw());

  int64_t NewWordSize = (A.getWordSize() + 1) * 2 - 1;
  int64_t Offset = A.getOffset();

  assert((NewWordSize + 1) * sizeof(int64_t) % 32 == 0);

  B.getRaw()[0] = 0;
  MemoryBlock Ret(A.getRaw(), NewWordSize, Offset);
  assert(Ret.isNulled());
  return Ret;
}

MemoryBlock::MemoryBlock() : Block(nullptr) {}

MemoryBlock::MemoryBlock(int64_t *Block) : Block(Block) {}

MemoryBlock::MemoryBlock(int64_t *Block, int64_t WordSize, int64_t ByteOffset)
    : Block(Block) {
  this->Block[0] = WordSize;
  this->Block[0] |= ByteOffset << MemoryBlock::OffsetShift;
```

```
}

int64_t MemoryBlock::getWordSize() {
  return this->Block[0] & MemoryBlock::SizeBitmask;
}

int64_t MemoryBlock::getOffset() {
  return (this->Block[0] & MemoryBlock::OffsetMask) >>
      MemoryBlock::OffsetShift;
}

int64_t *MemoryBlock::get() {
  assert(this->Block);
  return &this->Block[1];
}

int64_t *MemoryBlock::getRaw() { return this->Block; }

MemoryBlock MemoryBlock::getPrev() {
  return MemoryBlock((int64_t *)this->Block[1]);
}

MemoryBlock MemoryBlock::getNext() {
  return MemoryBlock((int64_t *)this->Block[2]);
}

void MemoryBlock::setPrev(MemoryBlock N) {
  assert(N ? this->getWordSize() == N.getWordSize() : 1);
  this->Block[1] = (int64_t)N.Block;
}

void MemoryBlock::setNext(MemoryBlock N) {
  assert(N ? this->getWordSize() == N.getWordSize() : 1);
  this->Block[2] = (int64_t)N.Block;
}

bool MemoryBlock::isNulled() {
  assert(this->Block);

  int64_t Size = this->getWordSize();
  int64_t *Ptr = this->get();
  for (int64_t I = 0; I < Size; ++I)
    if (Ptr[I])
      return false;
  return true;
}

MemoryBlock::operator bool() { return this->Block; }

// Get index of free list in FreeLists when (word) size of block is Size.
// Note that Size is not including the first word of the block.
static int64_t freeListIndex(int64_t Size) {
  int64_t Ret;
  Size += sizeof(int64_t);
  asm("bsrq %1, %0\n\t" : "=r"(Ret) : "r"(Size));
  if (Size & (Size - 1))
    Ret += 1;
  return std::max(Ret - 5, (int64_t)0);
}

// Get head of free list at FreeListIndex.
```

```
static MemoryBlock headFreeList(int64_t FreeListIndex) {
  assert(FreeListIndex < FreeListCount);

  MemoryBlock Ret = FreeLists[FreeListIndex];

  assert(
      Ret ? freeListIndex(Ret.getWordSize() * sizeof(int64_t)) ==
              FreeListIndex
          : 1);

  return Ret;
}

namespace {
// Append to free list at FreeListIndex by making B the new head
   of the list.
void pushFreeList(int64_t FreeListIndex, MemoryBlock B) {
  assert(!!B);
  assert(!B.getPrev());
  assert(B.getWordSize() >= MemoryBlock::MinimumWordSize);
  assert(freeListIndex(B.getWordSize() * sizeof(int64_t)) ==
      FreeListIndex);

  MemoryBlock Next = FreeLists[FreeListIndex];
  B.setNext(Next);
  if (Next)
    Next.setPrev(B);
  FreeLists[FreeListIndex] = B;
}
} // End annonymous namespace.

namespace {
// Remove head of free list at FreeListIndex.
MemoryBlock popFreeList(int64_t FreeListIndex) {
  MemoryBlock B = headFreeList(FreeListIndex);
  assert(!!B);

  MemoryBlock Next = B.getNext();
  if (Next)
    Next.setPrev(MemoryBlock());
  B.setNext(MemoryBlock());
  FreeLists[FreeListIndex] = Next;

  assert(B.getWordSize() >= MemoryBlock::MinimumWordSize);
  return B;
}
} // End annonymous namespace.

namespace {
void removeFreeList(int64_t FreeListIndex, MemoryBlock B) {
  MemoryBlock Prev = B.getPrev();
  MemoryBlock Next = B.getNext();
  if (!Prev) {
    FreeLists[FreeListIndex] = Next;
  } else {
    Prev.setNext(Next);
    B.setPrev(MemoryBlock());
  }
  if (Next) {
    Next.setPrev(Prev);
    B.setNext(MemoryBlock());
  }
```

```
  assert(Prev ? Prev.getWordSize() >= MemoryBlock::MinimumWordSize : 1);
  assert(Next ? Next.getWordSize() >= MemoryBlock::MinimumWordSize : 1);
  assert(Next ? (Prev ? Next.getWordSize() == Prev.getWordSize() : 1) : 1);
}
} // End annonymous namespace.

namespace {
// Split head of free list at FreeListIndex. Insert one part
// in free list at FreeListIndex - 1 and return the other part.
MemoryBlock splitFreeList(int64_t FreeListIndex) {
  assert(FreeListIndex > 0);
  assert(FreeListIndex < FreeListCount);

  MemoryBlock B = popFreeList(FreeListIndex);

  auto P = B.split();
  pushFreeList(FreeListIndex - 1, P.second);

  assert(P.first.getWordSize() >= MemoryBlock::MinimumWordSize);
  assert(P.second.getWordSize() >= MemoryBlock::MinimumWordSize);
  assert(P.first.getWordSize() == P.second.getWordSize());
  return P.first;
}
} // End annonymous namespace.

struct BaseBlockNode {
  int64_t *BlockOffset;
  BaseBlockNode *Prev;
  BaseBlockNode *Next;
};
static BaseBlockNode *BaseBlockList;

struct BigBlockNode {
  BigBlockNode *Prev;
  BigBlockNode *Next;
  int64_t WordSize;
};
static BigBlockNode *BigBlockList;

static void triggerGC(FunctionFrame FirstFrame) {
// std::cout << " ************** TRIGGER GC ************** \n";
#ifdef SSML_GC_STATISTICS
  uint64_t NumberRoots = 0;
  Timer Tim;
  Tim.start();
#endif // SSML_GC_STATISTICS

  assert(NormalAllocSize >= NormalAllocUsage);

  // First mark list nodes used to hold base blocks.
  for (BaseBlockNode *N = BaseBlockList; N; N = N->Next)
    mark((int64_t *)N);

#ifdef SSML_SHADOW_STACK_GC
  for (StackEntry *R = llvm_gc_root_chain; R; R = R->Next) {
    assert(R->Map->NumMeta == 0);
    auto Roots = R->Roots;
    for (unsigned I = 0, E = R->Map->NumRoots; I != E; ++I) {
```

```
      markall((int64_t *)Roots[I]);
#ifdef SSML_GC_STATISTICS
      ++NumberRoots;
#endif // SSML_GC_STATISTICS
    }
  }
#else // !defined(SSML_SHADOW_STACK_GC)
  FunctionFrame Frame = FirstFrame;
  for (;;) {
    FrameDescr *Descr = getGCEntry(Frame.getFunction());
    int16_t *Offsets = Descr->RootOffsets;

    for (int64_t I = 0, Count = Descr->RootCount; I < Count; ++I) {
      markall(Frame.getRoot(Offsets[I]));
#ifdef SSML_GC_STATISTICS
      ++NumberRoots;
#endif // SSML_GC_STATISTICS
    }

    if (isEntryFunction(Frame.getFunction()))
      break;
    Frame = Frame.nextFrame(Descr);
  }
#endif // SSML_SHADOW_STACK_GC

  sweep();

  assert(NormalAllocSize >= NormalAllocUsage);
  if ((NormalAllocSize - NormalAllocUsage) * 4 < GCTriggerIncrease
      * 3)
    GCTriggerPoint = std::min(GCTriggerPoint + GCTriggerIncrease,
                              MAX_ALLOC_SIZE - BigAllocSize);

#ifdef SSML_GC_STATISTICS
  TotalGCRoots += NumberRoots;

  uint64_t Time = Tim.time();
  if (Time > WhorstGCTime)
    WhorstGCTime = Time;
  if (Time < BestGCTime)
    BestGCTime = Time;
  TotalGCTime += Time;
  ++TriggerGCCount;
#endif // SSML_GC_STATISTICS
}

static bool needGC(int64_t Size) {
  return NormalAllocSize + Size > GCTriggerPoint;
}

// Allocate zero initialized memory aligned with page size.
static int64_t *newBlock(uint64_t Size) {
#ifdef MMAP_ALLOC
  void *Mem = mmap(nullptr, Size, PROT_READ | PROT_WRITE,
                   MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
  if (Mem == MAP_FAILED) {
    perror("unable to allocate memory");
    error("out of memory");
  }
#else
  void *Mem = calloc(1, Size);
  if (!Mem) {
```

```
    perror("unable to allocate memory");
    error("out of memory");
  }
#endif
  return (int64_t *)Mem;
}

static int64_t *pushNewMemoryBlock() {
  int64_t *Block = newBlock(MemoryIncrease);
  assert(Block);

  NormalAllocSize += MemoryIncrease;

  /* Block with word size and offset 0. */
  MemoryBlock MemB(Block, (MemoryIncrease / sizeof(int64_t)) - 1,
      0);
  pushFreeList(FreeListMaxIndex, MemB);
  return Block;
}

static int64_t *addNewBlock(void *PrevFrame) {
  int64_t *Block;
  if (needGC(MemoryIncrease)) {
    int64_t *Prev = (int64_t *)PrevFrame;
#ifndef SSML_SHADOW_STACK_GC
    assert(isfunc(Prev[1]));
#endif
    FunctionFrame Frame = {&Prev[2]};
    triggerGC(Frame);
    Block = nullptr;
  } else {
    Block = pushNewMemoryBlock();
  }
  return Block;
}

static MemoryBlock multiSplitFreeList(int64_t HighIdx, int64_t
    LowIdx) {
  assert(HighIdx < FreeListCount);
  assert(LowIdx >= 0);
  assert(!!headFreeList(HighIdx));

  if (HighIdx == LowIdx)
    return popFreeList(HighIdx);

  MemoryBlock Ret;
  for (;;) {
    Ret = splitFreeList(HighIdx);
    --HighIdx;
    if (HighIdx == LowIdx)
      break;
    pushFreeList(HighIdx, Ret);
  }
  return Ret;
}

static int64_t *getFromFreeList(int64_t Idx) {
  if (headFreeList(Idx))
    return popFreeList(Idx).get();

  for (int64_t H = Idx + 1; H < FreeListCount; ++H) {
    if (headFreeList(H))
```

```
      return multiSplitFreeList(H, Idx).get();
  }

  return nullptr;
}

static int64_t *freeListAllocate(int64_t N, void *Frame) {
  assert(N <= MemoryIncrease);
  int64_t Idx = freeListIndex(N);

  int64_t *Ret = getFromFreeList(Idx);
  if (Ret)
    goto free_list_allocate_out;

  {
    int64_t *NewB = addNewBlock(Frame);
    if (!NewB) {
      /* We allocated as much memory as we are allowed to at
       * the moment. GC have been triggered, we might have a
       * block in the free list now. */
      Ret = getFromFreeList(Idx);
      if (Ret)
        goto free_list_allocate_out;

      /* We were not able to find a block.
       * Now we try to increase the amount of memory allocated before a
       * GC is triggered and allocate a new base block. */
      auto PrevGCPoint = GCTriggerPoint;
      assert(PrevGCPoint <= MAX_ALLOC_SIZE - BigAllocSize);
      GCTriggerPoint = std::min(GCTriggerPoint + GCTriggerIncrease,
                                MAX_ALLOC_SIZE - BigAllocSize);
      if (GCTriggerPoint - PrevGCPoint < (uint64_t)MemoryIncrease) {
        /* Not allowed to increase amount of memory before GC. */
        error("out of memory doing normal allocation");
      }
      NewB = pushNewMemoryBlock();
    }
    /* We have just allocated a new base block of memory.
     * Which we have put in the free list. Split this block to
     * get a block of correct size for the user. */
    Ret = multiSplitFreeList(FreeListMaxIndex, Idx).get();
    /* Alloc memory for MemoryBlockNode to put NewB in the allocated list. */
    BaseBlockNode *Node = (BaseBlockNode *)allocate(sizeof(BaseBlockNode));
    Node->BlockOffset = NewB;
    Node->Next = BaseBlockList;
    if (BaseBlockList)
      BaseBlockList->Prev = Node;
    BaseBlockList = Node;
  }

free_list_allocate_out:

  NormalAllocUsage += (Ret[-1] & MemoryBlock::SizeBitmask) * sizeof(int64_t);
  assert(NormalAllocUsage <= NormalAllocSize);
  return Ret;
}

```

```
static int64_t *bigAllocate(int64_t N, void *PrevFrame) {
  uint64_t Total = N + 3 * sizeof(int64_t);

  /*std::cout << "BIG ALLOCATE: " << Total << "\n";*/

  if (GCTriggerPoint + BigAllocSize + Total > MAX_ALLOC_SIZE) {
    /* There is not enough memory.
     * Trigger GC. Maybe we can find the necessary memory. */
    int64_t *Prev = (int64_t *)PrevFrame;
#ifndef SSML_SHADOW_STACK_GC
    assert(isfunc(Prev[1]));
#endif
    FunctionFrame Frame = {&Prev[2]};
    triggerGC(Frame);
    if (GCTriggerPoint + BigAllocSize + Total > MAX_ALLOC_SIZE)
      error("out of memory doing big allocation");
  }

  BigBlockNode *Block = (BigBlockNode *)newBlock(Total);
  BigAllocSize += Total;

  Block->WordSize = N / sizeof(int64_t);
  Block->Next = BigBlockList;
  if (BigBlockList)
    BigBlockList->Prev = Block;
  BigBlockList = Block;

  return &((int64_t *)Block)[3];
}

static void bigRelease(BigBlockNode *Block) {
  uint64_t Total = (Block->WordSize + 3) * sizeof(int64_t);

  // std::cout << "BIG RELEASE: " << Total << "\n";

  BigAllocSize -= Total;

#ifdef MMAP_ALLOC
  if (munmap(Block, Total)) {
    perror("unable to deallocate memory block");
    error("will not continue with unexpected deallocation error");
  }
#else
  free(Block);
#endif

  // std::cout << "BigAllocMemory is now: " << BigAllocSize <<
      "\n";
}

// Mark object starting at Obj[-1] as allocated.
static void markAllocated(int64_t *Obj) {
  assert((Obj[-1] & MemoryBlock::SizeBitmask) >=
      MemoryBlock::MinimumWordSize);
  Obj[-1] |= MemoryBlock::IsAllocatedBit;
}

// Return whether this block pointed to by Obj is marked as
    allocated.
bool isAllocated(int64_t *Obj) {
  assert((Obj[-1] & MemoryBlock::SizeBitmask) >=
      MemoryBlock::MinimumWordSize);
```

```
  return Obj[-1] & MemoryBlock::IsAllocatedBit;
}

// Mark ibject starting at Obj[-1] as free.
static void unmarkAllocated(int64_t *Obj) {
  assert((Obj[-1] & MemoryBlock::SizeBitmask) >=
      MemoryBlock::MinimumWordSize);
  Obj[-1] &= MemoryBlock::IsAllocatedMask;
}

extern "C" int64_t *allocateFrame(int64_t N, void *Frame) {
  int64_t *Ret;

  assert(NormalAllocSize <= GCTriggerPoint);
  assert(GCTriggerPoint + BigAllocSize <= MAX_ALLOC_SIZE);

  assert(N % sizeof(int64_t) == 0);
  if (!N)
    return 0;

  if (N < MemoryIncrease)
    Ret = freeListAllocate(N, Frame);
  else
    Ret = bigAllocate(N, Frame);

  markAllocated(Ret);
  assert((Ret[-1] & MemoryBlock::SizeBitmask) >=
      MemoryBlock::MinimumWordSize);

  assert(MemoryBlock(&Ret[-1]).isNulled());
  assert(NormalAllocSize <= GCTriggerPoint);
  assert(GCTriggerPoint + BigAllocSize <= MAX_ALLOC_SIZE);

  return Ret;
}

extern "C" int64_t *allocate(int64_t N) {
  return allocateFrame(N, __builtin_frame_address(0));
}

static MemoryBlock release(int64_t *Obj) {
  assert(isAllocated(Obj));
  assert(!ismarked(Obj));

  unmarkAllocated(Obj);

  MemoryBlock B(&Obj[-1]);
  // Zero the released memory block.
  bzero(B.get(), B.getWordSize() * sizeof(int64_t));

  assert(NormalAllocUsage >= B.getWordSize() * sizeof(int64_t));
  NormalAllocUsage -= B.getWordSize() * sizeof(int64_t);

  int64_t FreeListIndex;
  for (;;) {
    assert(B.getWordSize());

    int64_t TotalByteSize = (B.getWordSize() + 1) *
        sizeof(int64_t);
    assert(TotalByteSize % 32 == 0);

    FreeListIndex = freeListIndex(TotalByteSize - sizeof(int64_t));
```

```
    if (TotalByteSize == MemoryIncrease)
      break;

    uint64_t Address = (uint64_t)B.getRaw();
    uint64_t NextSize = TotalByteSize * 2;

    uint64_t Buddy;
    if (B.getOffset() % NextSize == 0)
      Buddy = Address + TotalByteSize;
    else
      Buddy = Address - TotalByteSize;

    if (isAllocated((int64_t *)Buddy + 1))
      break;

    MemoryBlock BuddyBlock((int64_t *)Buddy);

    assert(!ismarked((int64_t *)Buddy + 1));
    assert((BuddyBlock.getWordSize() + 1) * sizeof(int64_t) % 32
        == 0);
    assert(BuddyBlock.getWordSize() <= B.getWordSize());

    if (BuddyBlock.getWordSize() != B.getWordSize())
      break;

    removeFreeList(FreeListIndex, BuddyBlock);
    B = memoryBlockMergeNulled(B, BuddyBlock);
    assert((uint64_t)B.getWordSize() ==
           (TotalByteSize * 2) / sizeof(int64_t) - 1);
  }

  assert(B.getWordSize() >= MemoryBlock::MinimumWordSize);
  pushFreeList(FreeListIndex, B);

  return B;
}

static int64_t mark(int64_t *Obj) {
  int64_t Size = Obj[-1] & MemoryBlock::SizeBitmask;
  Obj[-1] |= MemoryBlock::ObjectMarkBit;
  return Size;
}

static void unmark(int64_t *Obj) { Obj[-1] &=
    MemoryBlock::ObjectMarkMask; }

static bool ismarked(int64_t *Obj) {
  return Obj[-1] & MemoryBlock::ObjectMarkBit;
}

#if 0
static void markall(int64_t *Root) {
  if (!istraced(Root) || ismarked(Root))
    return;
  assert(isAllocated(Root));

  int64_t Size = mark(Root);
  for (int64_t I = 0; I < Size; ++I)
    markall((int64_t *)Root[I]);
}
#else
// Function markall marks object Root and all objects
```

```
// reachable from Root. If Root is not dynamically allocated
// or if is not a pointer, then the function does nothing.
static void markall(int64_t *Root) {
  // If Root is not a dynamically allocated pointer or
  // if Root is already marked, then return:
  if (!istraced(Root) || ismarked(Root))
    return;

  // Linked list containing reachable unmarked objects:
  std::forward_list<int64_t *> Objects;
  // Initialliy Root is unmarked:
  Objects.push_front(Root);

  do {
    // Get Obj from front of list:
    int64_t *Obj = Objects.front();
    // Remove front of list:
    Objects.pop_front();

    assert(isAllocated(Obj));

    // Mark Obj, and return the word size of Obj:
    int64_t Size = mark(Obj);
    // For each word in Obj:
    for (int64_t I = 0; I < Size; ++I) {
      // Get Child from index I of Obj:
      int64_t *Child = (int64_t *)Obj[I];
      // If Child is a dynamically allocated object and
      // it has not been marked, then put it in list with
      // unmakred reachable objects:
      if (istraced(Child) && !ismarked(Child))
        Objects.push_front(Child);
    }
    // While there are reachable unmarked objects:
  } while (!Objects.empty());
}
#endif

static void sweep() {
  for (auto N = BaseBlockList; N; N = N->Next) {
    uint64_t Addr = (uint64_t)N->BlockOffset;
    uint64_t End = Addr + MemoryIncrease;
    uint64_t NextAddr = Addr;
    for (MemoryBlock B((int64_t *)Addr); NextAddr < End;) {
      auto Obj = B.get();
      if (ismarked(Obj))
        unmark(Obj);
      else if (isAllocated(Obj))
        B = release(Obj);

      uint64_t Inc = (B.getWordSize() + 1) * sizeof(int64_t);
      assert(Inc <= (uint64_t)MemoryIncrease);

      Addr = (uint64_t)B.getRaw();
      NextAddr = Addr + Inc;
      B = MemoryBlock((int64_t *)NextAddr);
    }
  }

  for (auto Block = BigBlockList; Block;) {
    BigBlockNode *B = Block;
    Block = Block->Next;
```

```
    int64_t *Obj = &((int64_t *)B)[3];
    if (ismarked(Obj)) {
      unmark(Obj);
    } else {
      if (B->Prev)
        B->Prev->Next = Block;
      else
        BigBlockList = Block;
      if (Block)
        Block->Prev = B->Prev;
      assert(isAllocated(Obj));
      bigRelease(B);
    }
  }
}

#endif // SSML_MARK_SWEEP_GC
```

Listing 8: runtime/Main.c

```
#include <stdlib.h>
#include <stdint.h>

#ifdef SSML_GENERATIONAL_GC
uint64_t FirstGenBegin;
uint64_t FirstGenEnd;

void setFirstGenBegin(uint64_t V) { FirstGenBegin = V; }
void setFirstGenEnd(uint64_t V) { FirstGenEnd = V; }

#endif /* SSML_GENERATIONAL_GC */

int64_t entry();

void initFramePrintVal();

void initStdLib();

void ssmlGCInit();

#ifdef SSML_GC_STATISTICS
void ssmlGCPrintStatistics();
#endif /* SSML_GC_STATISTICS */

/*void *gcmap;*/
int16_t gclookup_size;
int64_t *gclookup_map;

static void initGC() {
#if !defined(SSML_SHADOW_STACK_GC) && !defined(SSML_DISABLE_GC)
  /*asm("\tmovq $ssml_gcmap, %0\n" : "=r"(gcmap) :);*/
  asm("\tmovw ssml_gclookup_size, %0\n" : "=r"(gclookup_size) :);
  asm("\tmovq $ssml_gclookup_map, %0\n" : "=r"(gclookup_map) :);
#endif
  ssmlGCInit();
}

int main() {
  int ret;
  initFramePrintVal();
```

```
  initStdLib();
  initGC();
  ret = entry();
#ifdef SSML_GC_STATISTICS
  ssmlGCPrintStatistics();
#endif /* SSML_GC_STATISTICS */
  return ret;
}
```

Listing 9: runtime/GC.h

```
#ifndef LLVM_TOOLS_SSML_RUNTIME_GC_H
#define LLVM_TOOLS_SSML_RUNTIME_GC_H

#include <iostream>
#include <stdint.h>
#include <assert.h>

#pragma GCC diagnostic ignored "-pedantic"

#define MAX_ALLOC_SIZE ((uint64_t)1048576 * 2048) // 1MB * 2048.

extern "C" void error(const char *);

#ifdef SSML_SHADOW_STACK_GC
struct FrameMap {
  int32_t NumRoots;
  int32_t NumMeta;
  const void *Meta[0];
};

struct StackEntry {
  StackEntry *Next;
  const FrameMap *Map;
  void *Roots[0];
};

extern StackEntry *llvm_gc_root_chain;
#endif

extern "C" int16_t gclookup_size;
extern "C" int64_t *gclookup_map;

inline bool isstatic(int64_t *Obj) {
  extern char __executable_start;
  //extern char __etext;
  extern char end;
  return &__executable_start <= (char *)Obj && (char *)Obj <= &end;
}

inline bool istraced(int64_t *Obj) {
  if (!Obj || isstatic(Obj))
    return false;
  return !((int64_t)Obj & 1);
}

inline int64_t codeBegin() { return gclookup_map[0]; }

inline int64_t codeEnd() { return gclookup_map[gclookup_size * 2];
    }

```

```
inline bool isfunc(int64_t Obj) {
  return Obj >= codeBegin() && Obj < codeEnd();
}

struct FrameDescr {
  int16_t StackFrameSize;
  int16_t RootCount;
  int16_t RootOffsets[0];
};

namespace {
struct FunctionFrame {
  int64_t *Frame;

  int64_t getFunction() { return Frame[-1]; }

  struct FunctionFrame nextFrame(FrameDescr *D) {
    FunctionFrame Ret = {&Frame[D->StackFrameSize + 1]};
    assert(isfunc(Ret.getFunction()));
    return Ret;
  }

  int64_t *getRoot(int16_t Index) {
    return (int64_t *)this->Frame[Index];
  }

  void setRoot(int16_t Index, int64_t *Root) {
    this->Frame[Index] = (int64_t)Root;
  }
};
} // End annonymous namespace.

inline int64_t *gclookup(int64_t Idx) {
  assert(Idx < gclookup_size);
  return &gclookup_map[Idx * 2];
}

inline int64_t gclookup_compare(int64_t *LookupElem, int64_t Addr)
    {
  if (LookupElem[2] <= Addr)
    return 1;
  if (LookupElem[0] > Addr)
    return -1;
  return 0;
}

inline bool isEntryFunction(int64_t Addr) {
  return !gclookup_compare(gclookup(0), Addr);
}

inline FrameDescr *getGCEntry(int64_t Addr) {
  int64_t Min = 0;
  int64_t Max = gclookup_size - 1;

  while (Min <= Max) {
    int64_t Mid = Min + (Max - Min) / 2;
    int64_t *LookupElem = gclookup(Mid);
    int64_t Cmp = gclookup_compare(LookupElem, Addr);
    if (!Cmp)
      return (FrameDescr *)LookupElem[1];
    if (Cmp == 1)
      Min = Mid + 1;
```

```
    else
      Max = Mid - 1;
  }

  std::cerr << "unable to find function address: " << (void *)Addr << std::endl;
  error("GC function address was not found");
  return nullptr;
}

extern "C" void ssmlGCPrintStatistics();

extern "C" int64_t *allocateFrame(int64_t N, void *Frame);

extern "C" int64_t *allocate(int64_t N);

#endif // LLVM_TOOLS_SSML_RUNTIME_GC_H
```

Listing 10: runtime/MallocMarkSweepAlloc.cpp

```
#if 0
#ifdef SSML_MARK_SWEEP_GC

#include "GC.h"
#include "ssml/Common/Timer.h"

#include <stdlib.h>
#include <assert.h>
#include <stdint.h>
#include <errno.h>
#include <sys/mman.h>
#include <strings.h>

#include <algorithm>
#include <iostream>
#include <set>
#include <forward_list>

#pragma GCC diagnostic ignored "-pedantic"

using namespace ssml;

static const uint64_t TriggerPoint = pow(2, 26);
static uint64_t AllocatedSize;

extern "C" void ssmlGCInit() {}

extern "C" void ssmlGCPrintStatistics() {
  std::cout << "****** Malloc Mark Sweep GC *******\n";
  std::cout << "************************************" << std::endl;
}

static std::set<int64_t *> Objects;

static void markall(int64_t *Root);
static void sweep();

static void triggerGC(void *PrevFrame) {
  std::cout << " ***************** TRIGGER GC *************" << std::endl;
#ifndef SSML_SHADOW_STACK_GC
```

```
  int64_t *PrevF = (int64_t *)PrevFrame;
  assert(isfunc(PrevF[1]));
  FunctionFrame FirstFrame = {&PrevF[2]};
#endif

#ifdef SSML_SHADOW_STACK_GC
  for (StackEntry *R = llvm_gc_root_chain; R; R = R->Next) {
    assert(R->Map->NumMeta == 0);
    auto Roots = R->Roots;
    for (unsigned I = 0, E = R->Map->NumRoots; I != E; ++I) {
      markall((int64_t *)Roots[I]);
    }
  }

#else  // !defined(SSML_SHADOW_STACK_GC)
  FunctionFrame Frame = FirstFrame;
  for (;;) {
    FrameDescr *Descr = getGCEntry(Frame.getFunction());
    int16_t *Offsets = Descr->RootOffsets;

    for (int64_t I = 0, Count = Descr->RootCount; I < Count; ++I) {
      markall(Frame.getRoot(Offsets[I]));
    }

    if (isEntryFunction(Frame.getFunction()))
      break;
    Frame = Frame.nextFrame(Descr);
  }
#endif // SSML_SHADOW_STACK_GC

  sweep();
}

static int64_t ObjectNumber;

extern "C" int64_t *allocateFrame(int64_t N, void *Frame) {
  uint64_t Size = N + sizeof(int64_t) * 2;
  if (AllocatedSize + Size > TriggerPoint)
    triggerGC(Frame);
  if (AllocatedSize + Size > TriggerPoint)
    error("out of memory");

  AllocatedSize += Size;
  int64_t *Mem = (int64_t *)calloc(1, Size);
  Mem[0] = ++ObjectNumber;
  Mem[1] = N / sizeof(int64_t);

  Objects.insert(Mem + 2);

  return (int64_t *)Mem + 2;
}

extern "C" int64_t *allocate(int64_t N) {
  return allocateFrame(N, __builtin_frame_address(0));
}

static const int64_t ObjectMarkBit = (int64_t)1 << 63;
static const int64_t ObjectMarkMask = ~ObjectMarkBit;
static const int64_t SizeBitmask = ObjectMarkMask;

static int64_t getSize(int64_t *Obj) {
  return Obj[-1] & SizeBitmask;
```

```
}

static int64_t mark(int64_t *Obj) {
  int64_t Size = getSize(Obj);
  Obj[-1] |= ObjectMarkBit;
  return Size;
}

static void unmark(int64_t *Obj) { Obj[-1] &= ObjectMarkMask; }

static bool ismarked(int64_t *Obj) {
  return Obj[-1] & ObjectMarkBit;
}

static void markall(int64_t *Root) {
  if (!istraced(Root) || ismarked(Root))
    return;

  static int64_t *LastGood;
  if (!Objects.count(Root))
    std::cout << "Error object: " << Root << " PREV GOOD: " <<
        LastGood << std::endl;
  LastGood = Root;
  assert(Objects.count(Root));

  int64_t Size = mark(Root);
  for (int64_t I = 0; I < Size; ++I)
    markall((int64_t *)Root[I]);
}

static void sweep() {
  for(auto It = Objects.begin(); It != Objects.end(); ) {
    if(!ismarked(*It)) {
      AllocatedSize -= getSize(*It) * sizeof(int64_t);
      free(*It - 2);
      It = Objects.erase(It);
    } else {
      unmark(*It);
      ++It;
    }
  }
}

#endif // SSML_MARK_SWEEP_GC
#endif // 0
```

Listing 11: runtime/DisableGC.cpp

```
#ifdef SSML_DISABLE_GC

#include "GC.h"

#include <iostream>

#include <stdint.h>
#include <stdlib.h>

extern "C" void error(const char *);

extern "C" void ssmlGCInit() {}

```

```
static uint64_t MemoryUsage;

extern "C" int64_t *allocate(int64_t N) {
  size_t Total = N + sizeof(int64_t);
  MemoryUsage += Total;
  if (MemoryUsage > MAX_ALLOC_SIZE)
    error("out of memory");
  int64_t *Ret = (int64_t *)calloc(1, Total);
  if (!Ret)
    error("out of memory");
  return Ret;
}

extern "C" int64_t *allocateFrame(int64_t N, void *) {
  return allocate(N);
}

extern "C" void ssmlGCPrintStatistics() {
  uint64_t Avail = MAX_ALLOC_SIZE - MemoryUsage;
  std::cout << "*********** GC Disabled ***********\n";
  std::cout << "Memory Usage:       " << MemoryUsage << "B\n";
  std::cout << "Total Free Memory: " << Avail << "B\n";
  std::cout << "***********************************" << std::endl;
}

#endif // SSML_DISABLE_GC
```

Listing 12: tools/driver/Driver.cpp

```
#define DEBUG_TYPE "main"

#include "ssml/AST/Declaration.h"
#include "ssml/AST/DumpVisitor.h"
#include "ssml/AST/Fixup/Fixup.h"

#include "ssml/Typecheck/Typecheck.h"
#include "ssml/Codegen/Codegen.h"

#include "llvm/Support/CommandLine.h"
#include "llvm/Support/Debug.h"
#include "llvm/Support/raw_ostream.h"
#include "llvm/Support/ManagedStatic.h"

using namespace llvm;

ssml::ast::Root *getParseRoot_TEMPORARY();
int yyparse();
namespace ssml {
namespace parse {
void flexFinalize();
} // End namespace ssml.
} // End namespace parse.

static void versionPrinter() {
  errs() << "SSML:\n"
            "  SSML version 0.1\n";
}

static cl::opt<bool> DumpASTOption("dump-ast",
                                   cl::desc("Print AST after
                                       parsing"),
```

```
                                            cl::init(false));
static cl::opt<bool>
    DumpASTFixupOption("dump-ast-fixup",
                       cl::desc("Print AST after parsing and AST
                           fixup"),
                       cl::init(false));

static int start(int Argc, const char *const *Argv) {
  cl::SetVersionPrinter(versionPrinter);
  cl::ParseCommandLineOptions(Argc, Argv, "SSML program overview
      description");
  DEBUG(errs() << "TEST DEBUG\n");

  int ret = yyparse();
  if (ret)
    return ret;

  auto Root = getParseRoot_TEMPORARY();
  ssml::ast::DumpVisitor V;
  if (DumpASTOption) {
    errs() << "Initial AST:\n";
    Root->accept(&V);
  }

  ssml::parse::flexFinalize();

  ssml::ast::fixup(Root);
  if (DumpASTFixupOption) {
    if (DumpASTOption)
      errs() << "\n";
    errs() << "Fixup AST:\n";
    Root->accept(&V);
  }

  ssml::typecheck::typecheck(Root);
  ssml::codegen::codegen(Root);

  delete Root;
  return 0;
}

int main(int Argc, const char *const *Argv) {
  llvm_shutdown_obj ShutdownObj;
  (void)ShutdownObj;
  return start(Argc, Argv);
}
```

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

```
#ifndef SSML_CODEGEN_CODEGEN_H
#define SSML_CODEGEN_CODEGEN_H

namespace ssml {
namespace ast {
class Root;
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace codegen {
void codegen(ssml::ast::Root *);
```

```
} // End namespace codegen.
} // End namespace ssml.

#endif // SSML_CODEGEN_CODEGEN_H
```

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

```
#ifndef SSML_COMMON_MEMORY_H
#define SSML_COMMON_MEMORY_H

#include <memory>

namespace ssml {
template <typename T>
std::shared_ptr<T> toSharedPtr(T *P) { return
    std::shared_ptr<T>(P); }
}

#endif // SSML_COMMON_MEMORY_H
```

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

```
#ifndef SSML_COMMON_STRINGMANIPULATION_H
#define SSML_COMMON_STRINGMANIPULATION_H

namespace ssml {
char *duplicate(const char *str);
} // End namespace ssml.

#endif // SSML_COMMON_STRINGMANIPULATION_H
```

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

```
#ifndef SSML_COMMON_TIMER_H
#define SSML_COMMON_TIMER_H

#include <cstdint>
#include <sys/time.h>

namespace ssml {
class Timer {
  timeval Start;
  timeval Stop;

public:
  void start() {
    gettimeofday(&this->Start, nullptr);
  }

  uint64_t time() {
    timeval Diff;
    gettimeofday(&this->Stop, nullptr);
    timersub(&Stop, &Start, &Diff);
    return (uint64_t)Diff.tv_sec * 1000000 +
        (uint64_t)Diff.tv_usec;
  }
};
} // End namespace ssml.

```

```
#endif // SSML_COMMON_TIMER_H
```

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

```
#ifndef SSML_COMMON_LIMITS_H
#define SSML_COMMON_LIMITS_H

#include <cstdlib>

namespace ssml {
namespace limits {
namespace int64 {
constexpr int64_t min() { return -9223372036854775807; }
constexpr int64_t max() { return 9223372036854775807; }
} // End namespace int64.
} // End namespace limits.
} // End namespace ssml.

#endif // SSML_COMMON_LIMITS_H
```

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

```
#ifndef SSML_COMMON_FATALEXIT_H
#define SSML_COMMON_FATALEXIT_H

#include "llvm/ADT/Twine.h"

namespace ssml {
void fatalExit(llvm::Twine Msg);
} // End namespace ssml.

namespace ssml {
void fatalExitOutOfMem();
} // End namespace ssml.

#endif // SSML_COMMON_FATALEXIT_H
```

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

```
#ifndef SSML_COMMON_COMPARE_H
#define SSML_COMMON_COMPARE_H

#include <memory>

namespace ssml {
template <typename T> struct SharedPtrLess {
  bool operator()(std::shared_ptr<const T> L,
      std::shared_ptr<const T> R) const {
    return *L < *R;
  }
};
} // End namespace ssml.

namespace ssml {
extern template struct SharedPtrLess<std::string>;
} // End namespace ssml.

#endif // SSML_COMMON_COMPARE_H
```

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

```
#ifndef SSML_SOURCELOCATION_H
#define SSML_SOURCELOCATION_H

#include <string>

namespace llvm {
class raw_ostream;
} // End namespace llvm.

namespace ssml {
class SourceLocation {
  unsigned Line;
  unsigned Column;

public:
  explicit SourceLocation() = default;
  constexpr SourceLocation(unsigned Line, unsigned Col)
      : Line(Line), Column(Col) {}

  constexpr unsigned getLine() const { return this->Line; }
  constexpr unsigned getColumn() const { return this->Column; }

  std::string toString() const;
};
} // End namespace ssml

namespace llvm {
raw_ostream &operator<<(raw_ostream &, ssml::SourceLocation);
} // End namespace llvm.

#endif // SSML_SOURCELOCATION_H
```

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

```
#ifndef SSML_COMMON_ERRORMESSAGES_H
#define SSML_COMMON_ERRORMESSAGES_H

#include "ssml/Common/SourceLocation.h"

namespace llvm {
class StringRef;
} // End namespace llvm.

namespace ssml {
void errorExit(llvm::StringRef FileName, SourceLocation Loc,
               llvm::StringRef Message);
} // End namespace ssml.

namespace ssml {
void errorExit(SourceLocation Loc, llvm::StringRef Message);
} // End namespace ssml.

#endif // SSML_COMMON_ERRORMESSAGES_H
```

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

```
#ifndef SSML_COMMON_COMMANDLINE_H
#define SSML_COMMON_COMMANDLINE_H

```

```
#include <memory>
#include <string>

namespace ssml {
class CommandlineImpl {
public:
  const std::shared_ptr<std::string> getFilename() const;
};
} // End namespace ssml.

namespace ssml {
extern CommandlineImpl Commandline;
}

#endif // SSML_COMMON_COMMANDLINE_H
```

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

```
#ifndef SSML_AST_ALL_H
#define SSML_AST_ALL_H

#include "ssml/AST/Node.h"
#include "ssml/AST/Declaration.h"
#include "ssml/AST/Identifier.h"
#include "ssml/AST/Literal.h"
#include "ssml/AST/Match.h"
#include "ssml/AST/Pattern.h"
#include "ssml/AST/Expression.h"
#include "ssml/AST/Type.h"
#include "ssml/AST/Definition.h"

#endif // SSML_AST_ALL_H
```

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

```
#ifndef SSML_AST_DEFINITION_H
#define SSML_AST_DEFINITION_H

#include "Node.h"

#include <memory>

namespace ssml {
namespace ast {
class LongIdentifier;

class SeqDeclaration;
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class Definition : public Node {
protected:
  explicit Definition(SourceLocation);
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
```

```
class LongIdentifierDefinition : public Definition {
private:
  std::shared_ptr<LongIdentifier> ID;

public:
  LongIdentifierDefinition(SourceLocation, std::shared_ptr<LongIdentifier> ID);
  void accept(Visitor *) override;

  std::shared_ptr<LongIdentifier> getID() { return this->ID; }
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class StructDefinition : public Definition {
private:
  std::shared_ptr<SeqDeclaration> Decls;

public:
  StructDefinition(SourceLocation,
                   std::shared_ptr<SeqDeclaration> Declarations);
  void accept(Visitor *) override;

  std::shared_ptr<SeqDeclaration> getDecls() { return this->Decls; }
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class AnnotationDefinition : public Definition {
private:
  std::shared_ptr<Definition> StructDef;
  std::shared_ptr<Definition> SigDef;
  bool Transparent;

public:
  AnnotationDefinition(SourceLocation, std::shared_ptr<Definition> StructDef,
                       std::shared_ptr<Definition> SigDef, bool isTransparent);
  void accept(Visitor *) override;

  std::shared_ptr<Definition> getStructDef() { return this->StructDef; }
  std::shared_ptr<Definition> getSigDef() { return this->SigDef; }
  bool isTransparent() const { return this->Transparent; }
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class ShortFunctorDefinition : public Definition {
private:
  std::shared_ptr<LongIdentifier> FunctorID;
  std::shared_ptr<Definition> StructDef;

public:
```

```
  ShortFunctorDefinition(SourceLocation,
                         std::shared_ptr<LongIdentifier> FunctorID,
                         std::shared_ptr<Definition> StructDef);
  void accept(Visitor *) override;

  std::shared_ptr<LongIdentifier> getFunctorID() { return this->FunctorID; }
  std::shared_ptr<Definition> getStructDef() { return this->StructDef; }
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class LongFunctorDefinition : public Definition {
private:
  std::shared_ptr<LongIdentifier> FunctorID;
  std::shared_ptr<SeqDeclaration> Decls;

public:
  LongFunctorDefinition(SourceLocation,
                        std::shared_ptr<LongIdentifier> FunctorID,
                        std::shared_ptr<SeqDeclaration> Declarations);
  void accept(Visitor *) override;

  std::shared_ptr<LongIdentifier> getFunctorID() { return this->FunctorID; }
  std::shared_ptr<SeqDeclaration> getDecls() { return this->Decls; }
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class SigDefinition : public Definition {
private:
  std::shared_ptr<SeqDeclaration> Decls;

public:
  SigDefinition(SourceLocation, std::shared_ptr<SeqDeclaration> Declarations);
  void accept(Visitor *) override;

  std::shared_ptr<SeqDeclaration> getDecls() { return this->Decls; }
};
} // End namespace ast.
} // End namespace ssml.

#endif // SSML_AST_DEFINITION_H
```

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

```
#ifndef SSML_AST_LITERAL_H
#define SSML_AST_LITERAL_H

#include "Node.h"

```

```
#include <string>
#include <memory>

namespace ssml {
namespace ast {
class Literal : public Node {
  std::shared_ptr<std::string> Value;

protected:
  explicit Literal(SourceLocation, std::shared_ptr<std::string>
      Lit);

public:
  const std::shared_ptr<std::string> getValueString() { return
      this->Value; }
  virtual int64_t toInt() = 0;
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class IntLiteral : public Literal {
public:
  using IntType = std::int64_t;

private:
  IntType IntValue;

public:
  explicit IntLiteral(SourceLocation, std::shared_ptr<std::string>
      Value);
  void accept(Visitor *) override;
  int64_t getIntValue() const;
  int64_t toInt() override;
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class RealLiteral : public Literal {
public:
  explicit RealLiteral(SourceLocation,
      std::shared_ptr<std::string> Value);
  void accept(Visitor *) override;
  int64_t toInt() override;
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class CharLiteral : public Literal {
public:
  explicit CharLiteral(SourceLocation,
      std::shared_ptr<std::string> Value);
  void accept(Visitor *) override;
  int64_t toInt() override;
};
} // End namespace ast.
} // End namespace ssml.
```

```

namespace ssml {
namespace ast {
class StringLiteral : public Literal {
public:
  explicit StringLiteral(SourceLocation,
      std::shared_ptr<std::string> Value);
  void accept(Visitor *) override;
  int64_t toInt() override;
};
} // End namespace ast.
} // End namespace ssml.

#endif // SSML_AST_LITERAL_H
```

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

```
#ifndef SSML_AST_PATTERN_H
#define SSML_AST_PATTERN_H

#include "Node.h"

#include <vector>
#include <memory>

namespace ssml {
namespace ast {
class Literal;
class Identifier;
class ShortIdentifier;
class LongIdentifier;
class Type;
class TypePattern;
class LongIdentifierPattern;
class ApplyPattern;
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class Pattern : public Node {
protected:
  explicit Pattern(SourceLocation);
  bool HasSimpleType = false;
public:
  virtual TypePattern *asTypePattern();
  virtual LongIdentifierPattern *asLongIdentifierPattern();
  virtual ApplyPattern *asApplyPattern();

  bool hasSimpleType() { return this->HasSimpleType; }
  void setHasSimpleType(bool B = true) { this->HasSimpleType = B; }
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class SeqPattern : public Pattern,
                   public std::vector<std::shared_ptr<Pattern>> {
public:
  explicit SeqPattern(SourceLocation);
```

```
  explicit SeqPattern(SourceLocation, std::shared_ptr<Pattern>
      First);
  void accept(Visitor *V) override;
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class LiteralPattern : public Pattern {
  std::shared_ptr<Literal> Value;

public:
  explicit LiteralPattern(SourceLocation,
      std::shared_ptr<Literal>);
  void accept(Visitor *) override;
  std::shared_ptr<Literal> getValue() { return this->Value; }
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class WildcardPattern : public Pattern {
public:
  explicit WildcardPattern(SourceLocation);
  void accept(Visitor *) override;
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class LongIdentifierPattern : public Pattern {
private:
  std::shared_ptr<LongIdentifier> ID;
  bool opKeyPrefixed;

public:
  explicit LongIdentifierPattern(SourceLocation,
                                 std::shared_ptr<LongIdentifier>
                                     ID,
                                 bool isPrefixedWithOpKey = false);
  void accept(Visitor *) override;
  bool isPrefixedWithOpKey() const { return this->opKeyPrefixed; }
  std::shared_ptr<LongIdentifier> getIDs() { return this->ID; }
  LongIdentifierPattern *asLongIdentifierPattern() override;
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class TypePattern : public Pattern {
private:
  std::shared_ptr<Pattern> Pat;
  std::shared_ptr<Type> Typ;

public:
  explicit TypePattern(SourceLocation, std::shared_ptr<Pattern> P,
                       std::shared_ptr<Type> T);
  void accept(Visitor *) override;
```

```
  std::shared_ptr<Pattern> getPattern() { return this->Pat; }
  std::shared_ptr<Type> getType() { return this->Typ; }
  TypePattern *asTypePattern() override;
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class AsPattern : public Pattern {
private:
  std::shared_ptr<Pattern> LeftPattern;
  std::shared_ptr<Pattern> RightPattern;

public:
  explicit AsPattern(SourceLocation, std::shared_ptr<Pattern> LHS,
                     std::shared_ptr<Pattern> RHS);
  void accept(Visitor *) override;
  std::shared_ptr<Pattern> getLeftPattern() { return this->LeftPattern; }
  std::shared_ptr<Pattern> getRightPattern() { return this->RightPattern; }
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class ApplyPattern : public Pattern,
                     public std::vector<std::shared_ptr<Pattern>> {
public:
  explicit ApplyPattern(SourceLocation, std::shared_ptr<Pattern> First);
  void accept(Visitor *V) override;
  ApplyPattern *asApplyPattern() override;
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class IdentifierEqualsPattern : public Pattern {
private:
  std::shared_ptr<Identifier> Iden;
  std::shared_ptr<Pattern> Pat;

public:
  explicit IdentifierEqualsPattern(SourceLocation,
                                   std::shared_ptr<Identifier> LHS,
                                   std::shared_ptr<Pattern> RHS);
  void accept(Visitor *V) override;
  std::shared_ptr<Identifier> getID() { return this->Iden; }
  std::shared_ptr<Pattern> getPattern() { return this->Pat; }
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class ShortIdentifierPattern : public Pattern {
private:
  std::shared_ptr<ShortIdentifier> ID;
```

```
public:
  explicit ShortIdentifierPattern(SourceLocation,
                                  std::shared_ptr<ShortIdentifier>
                                      ID);
  void accept(Visitor *) override;
  std::shared_ptr<ShortIdentifier> getID() { return this->ID; }
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class RecordPattern : public Pattern,
                      public std::vector<std::shared_ptr<Pattern>>
                          {
private:
  bool EllipsisTerminated = false;

public:
  explicit RecordPattern(SourceLocation, std::shared_ptr<Pattern>
      First);
  void accept(Visitor *V) override;
  bool isEllipsisTerminated() const { return
      this->EllipsisTerminated; }
  void makeEllipsisTerminated() { EllipsisTerminated = true; }
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class ListPattern : public Pattern {
private:
  std::shared_ptr<SeqPattern> Patterns;

public:
  explicit ListPattern(SourceLocation, std::shared_ptr<SeqPattern>
      Patterns);
  void accept(Visitor *V) override;
  std::shared_ptr<SeqPattern> getPatterns() { return
      this->Patterns; }
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class TuplePattern : public Pattern {
private:
  std::shared_ptr<SeqPattern> Patterns;

public:
  explicit TuplePattern(SourceLocation,
      std::shared_ptr<SeqPattern> Patterns);
  void accept(Visitor *V) override;
  std::shared_ptr<SeqPattern> getPatterns() { return
      this->Patterns; }
};
} // End namespace ast.
} // End namespace ssml.
```

```
#endif // SSML_AST_PATTERN_H
```

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

```
#ifndef SSML_AST_MATCH_H
#define SSML_AST_MATCH_H

#include "Node.h"

#include <vector>
#include <memory>

namespace ssml {
namespace ast {
class Pattern;
class Expression;
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class Match : public Node {
private:
  std::shared_ptr<Pattern> Pat;
  std::shared_ptr<Expression> Expr;

public:
  Match(SourceLocation, std::shared_ptr<Pattern> P,
        std::shared_ptr<Expression> E);
  void accept(Visitor *V) override;
  std::shared_ptr<Pattern> getPattern() { return this->Pat; }
  std::shared_ptr<Expression> getExpr() { return this->Expr; }
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class SeqMatch : public Node, public
    std::vector<std::shared_ptr<Match>> {
public:
  SeqMatch(SourceLocation, std::shared_ptr<Match> First);
  void accept(Visitor *V) override;
};
} // End namespace ast.
} // End namespace ssml.

#endif // SSML_AST_MATCH_H
```

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

```
#ifndef SSML_AST_PASS_FIXUP_H
#define SSML_AST_PASS_FIXUP_H

namespace ssml {
namespace ast {
class Root;
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
```

```
namespace ast {
void fixup(Root *);
} // End namespace ast.
} // End namespace ssml.

#endif // SSML_AST_PASS_FIXUP_H
```

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

```
#ifndef SSML_AST_FUNCTIONVALS_H
#define SSML_AST_FUNCTIONVALS_H

#include <vector>

namespace ssml {
namespace ast {
class FunctionVals {
private:
  std::vector<bool> Table;

public:
  void addVal(bool IsPointer) {
    this->Table.push_back(IsPointer);
  }
  std::vector<bool> &getFlags() { return this->Table; }
};
} // End namespace ast.
} // End namespace ssml.

#endif // SSML_AST_FUNCTIONVALS_H
```

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

```
#ifndef SSML_AST_EXPRESSION_H
#define SSML_AST_EXPRESSION_H

#include "Literal.h"
#include "FunctionVals.h"

#include <vector>

namespace ssml {
namespace ast {
class SeqMatch;
class Type;
class SeqDeclaration;
class Identifier;
class LongIdentifier;
class LongIdentifierExpression;
class ApplyExpression;
class TupleExpression;
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class Expression : public Node {
protected:
  explicit Expression(SourceLocation);
public:
  virtual TupleExpression *asTupleExpression();
```

```
  virtual LongIdentifierExpression *asLongIdentifierExpression();
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class SeqExpression : public Expression,
                      public
                          std::vector<std::shared_ptr<Expression>>
                          {
public:
  explicit SeqExpression(SourceLocation);
  explicit SeqExpression(SourceLocation,
      std::shared_ptr<Expression> First);
  void accept(Visitor *V) override;
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class LiteralExpression : public Expression {
  std::shared_ptr<Literal> Value;

public:
  explicit LiteralExpression(SourceLocation,
      std::shared_ptr<Literal> Lit);
  void accept(Visitor *) override;
  std::shared_ptr<Literal> getValue() { return this->Value; }
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class LongIdentifierExpression : public Expression {
private:
  std::shared_ptr<LongIdentifier> ID;
  bool opKeyPrefixed;

public:
  explicit LongIdentifierExpression(SourceLocation,
                                    std::shared_ptr<LongIdentifier>
                                        ID,
                                    bool isPrefixedWithOpKey =
                                        false);
  void accept(Visitor *) override;
  bool isPrefixedWithOpKey() const { return this->opKeyPrefixed; }
  std::shared_ptr<LongIdentifier> getIDs() { return this->ID; }
  LongIdentifierExpression *asLongIdentifierExpression() override;
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class TupleExpression : public Expression {
  std::shared_ptr<SeqExpression> Expressions;

public:
  explicit TupleExpression(SourceLocation,
```

```
                              std::shared_ptr<SeqExpression> Exprs);
  void accept(Visitor *) override;
  std::shared_ptr<SeqExpression> getExprs() { return
      this->Expressions; }
  TupleExpression *asTupleExpression() override;
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class ListExpression : public Expression {
  std::shared_ptr<SeqExpression> Expressions;

public:
  explicit ListExpression(SourceLocation,
      std::shared_ptr<SeqExpression> Exprs);
  void accept(Visitor *) override;
  std::shared_ptr<SeqExpression> getExprs() { return
      this->Expressions; }
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class IdentifierEqualsExpression : public Expression {
private:
  std::shared_ptr<Identifier> Iden;
  std::shared_ptr<Expression> Expr;

public:
  explicit IdentifierEqualsExpression(SourceLocation,
                                      std::shared_ptr<Identifier>
                                          LHS,
                                      std::shared_ptr<Expression>
                                          RHS);
  void accept(Visitor *V) override;
  std::shared_ptr<Identifier> getID() { return this->Iden; }
  std::shared_ptr<Expression> getExpr() { return this->Expr; }
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class RecordExpression
    : public Expression,
      public
          std::vector<std::shared_ptr<IdentifierEqualsExpression>>
          {
public:
  explicit RecordExpression(SourceLocation,
                            std::shared_ptr<IdentifierEqualsExpression>
                                First);
  void accept(Visitor *V) override;
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
```

```
class SelectorExpression : public Expression {
private:
  std::shared_ptr<Identifier> ID;

public:
  explicit SelectorExpression(SourceLocation,
      std::shared_ptr<Identifier> ID);
  void accept(Visitor *V) override;
  std::shared_ptr<Identifier> getID() { return this->ID; }
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class ApplyExpression : public Expression,
                        public
                            std::vector<std::shared_ptr<Expression>>
                            {
public:
  explicit ApplyExpression(SourceLocation,
      std::shared_ptr<Expression> First);
  void accept(Visitor *V) override;
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class TypeExpression : public Expression {
private:
  std::shared_ptr<Expression> Expr;
  std::shared_ptr<Type> Typ;

public:
  explicit TypeExpression(SourceLocation,
      std::shared_ptr<Expression> E,
                          std::shared_ptr<Type> T);
  void accept(Visitor *V) override;
  std::shared_ptr<Expression> getExpr() { return this->Expr; }
  std::shared_ptr<Type> getType() { return this->Typ; }
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class OrElseExpression : public Expression {
private:
  std::shared_ptr<Expression> Left;
  std::shared_ptr<Expression> Right;

public:
  explicit OrElseExpression(SourceLocation,
      std::shared_ptr<Expression> LHS,
                            std::shared_ptr<Expression> RHS);
  void accept(Visitor *V) override;
  std::shared_ptr<Expression> getLeftExpr() { return this->Left; }
  std::shared_ptr<Expression> getRightExpr() { return this->Right;
      }
};
} // End namespace ast.
```

```
} // End namespace ssml.

namespace ssml {
namespace ast {
class AndAlsoExpression : public Expression {
private:
  std::shared_ptr<Expression> Left;
  std::shared_ptr<Expression> Right;

public:
  explicit AndAlsoExpression(SourceLocation,
      std::shared_ptr<Expression> LHS,
                             std::shared_ptr<Expression> RHS);
  void accept(Visitor *V) override;
  std::shared_ptr<Expression> getLeftExpr() { return this->Left; }
  std::shared_ptr<Expression> getRightExpr() { return this->Right;
      }
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class LetExpression : public Expression {
private:
  std::shared_ptr<SeqDeclaration> Declarations;
  std::shared_ptr<SeqExpression> Expressions;

public:
  explicit LetExpression(SourceLocation,
      std::shared_ptr<SeqDeclaration> LHS,
                         std::shared_ptr<SeqExpression> RHS);
  void accept(Visitor *V) override;

  std::shared_ptr<SeqDeclaration> getDecls() {
    return this->Declarations;
  }

  std::shared_ptr<SeqExpression> getExprs() { return
      this->Expressions; }
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class IfExpression : public Expression {
private:
  std::shared_ptr<Expression> CondExpr;
  std::shared_ptr<Expression> ThenExpr;
  std::shared_ptr<Expression> ElseExpr;

public:
  explicit IfExpression(SourceLocation,
      std::shared_ptr<Expression> Cond,
                        std::shared_ptr<Expression> Then,
                        std::shared_ptr<Expression> Else);
  void accept(Visitor *V) override;
  std::shared_ptr<Expression> getCondExpr() { return
      this->CondExpr; }
  std::shared_ptr<Expression> getThenExpr() { return
      this->ThenExpr; }
```

```
  std::shared_ptr<Expression> getElseExpr() { return this->ElseExpr; }
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class WhileExpression : public Expression {
private:
  std::shared_ptr<Expression> CondExpr;
  std::shared_ptr<Expression> BodyExpr;

public:
  explicit WhileExpression(SourceLocation, std::shared_ptr<Expression> Cond,
                           std::shared_ptr<Expression> Body);
  void accept(Visitor *V) override;
  std::shared_ptr<Expression> getCondExpr() { return this->CondExpr; }
  std::shared_ptr<Expression> getBodyExpr() { return this->BodyExpr; }
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class CaseExpression : public Expression {
private:
  std::shared_ptr<Expression> Expr;
  std::shared_ptr<SeqMatch> Cases;

public:
  explicit CaseExpression(SourceLocation, std::shared_ptr<Expression> Expr,
                          std::shared_ptr<SeqMatch> Cases);
  void accept(Visitor *V) override;
  std::shared_ptr<Expression> getExpr() { return this->Expr; }
  std::shared_ptr<SeqMatch> getCases() { return this->Cases; }
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class LambdaExpression : public Expression {
private:
  std::shared_ptr<SeqMatch> Cases;
  std::shared_ptr<FunctionVals> FunVals;

public:
  explicit LambdaExpression(SourceLocation, std::shared_ptr<SeqMatch> Cases);
  void accept(Visitor *V) override;
  std::shared_ptr<SeqMatch> getCases() { return this->Cases; }

  void setFunctionVals(std::shared_ptr<FunctionVals> V) { this->FunVals = V; }
  std::shared_ptr<FunctionVals> getFunctionVals() { return this->FunVals; }
};
```

```
} // End namespace ast.
} // End namespace ssml.

#endif // SSML_AST_EXPRESSION_H
```

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

```
#ifndef SSML_AST_TYPE_H
#define SSML_AST_TYPE_H

#include "Node.h"

#include <vector>
#include <memory>

namespace ssml {
namespace ast {
class ApplyType;
class LongIdentifier;
class ShortIdentifier;
class Identifier;
class TupleType;
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class Type : public Node {
protected:
  explicit Type(SourceLocation);

public:
  virtual TupleType *asTupleType();
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class LongIdentifierType : public Type {
private:
  std::shared_ptr<LongIdentifier> ID;

public:
  explicit LongIdentifierType(SourceLocation,
                              std::shared_ptr<LongIdentifier> ID);
  void accept(Visitor *) override;
  std::shared_ptr<LongIdentifier> getIDs() { return this->ID; }
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class VariableType : public Type {
private:
  std::shared_ptr<ShortIdentifier> ID;

public:
  explicit VariableType(SourceLocation,
      std::shared_ptr<ShortIdentifier> ID);
```

```
  void accept(Visitor *) override;
  std::shared_ptr<ShortIdentifier> getID() { return this->ID; }
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class SeqType : public Type, public std::vector<std::shared_ptr<Type>> {
public:
  explicit SeqType(SourceLocation, std::shared_ptr<Type> First);
  void accept(Visitor *) override;
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class SeqVariableType : public Type,
                        public std::vector<std::shared_ptr<VariableType>> {
public:
  explicit SeqVariableType(SourceLocation, std::shared_ptr<VariableType> First);
  void accept(Visitor *) override;
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class TupleType : public Type {
private:
  std::shared_ptr<SeqType> Types;

public:
  explicit TupleType(SourceLocation, std::shared_ptr<SeqType> Types);
  void accept(Visitor *) override;
  std::shared_ptr<SeqType> getTypes() { return this->Types; }
  TupleType *asTupleType() override;
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class IdentifierColonType : public Type {
private:
  std::shared_ptr<Identifier> Iden;
  std::shared_ptr<Type> Typ;

public:
  explicit IdentifierColonType(SourceLocation, std::shared_ptr<Identifier> ID,
                               std::shared_ptr<Type> T);
  void accept(Visitor *) override;
  std::shared_ptr<Identifier> getIDs() { return this->Iden; }
  std::shared_ptr<Type> getType() { return this->Typ; }
};
```

```
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class RecordType : public Type,
                   public std::vector<std::shared_ptr<IdentifierColonType>> {
public:
  explicit RecordType(SourceLocation,
                      std::shared_ptr<IdentifierColonType> First);
  void accept(Visitor *) override;
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class ApplyType : public Type, public std::vector<std::shared_ptr<Type>> {
public:
  explicit ApplyType(SourceLocation, std::shared_ptr<Type> First);
  void accept(Visitor *) override;
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class ProductType : public Type, public std::vector<std::shared_ptr<Type>> {
public:
  explicit ProductType(SourceLocation, std::shared_ptr<Type> First);
  void accept(Visitor *) override;
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class FunctionType : public Type, public std::vector<std::shared_ptr<Type>> {
public:
  explicit FunctionType(SourceLocation, std::shared_ptr<Type> First);
  void accept(Visitor *) override;
};
} // End namespace ast.
} // End namespace ssml.

#endif // SSML_AST_TYPE_H
```

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

```
#ifndef SSML_AST_DECLARATION_H
#define SSML_AST_DECLARATION_H

#include "Node.h"
#include "FunctionVals.h"
```

```

#include <vector>
#include <memory>

namespace ssml {
namespace ast {
class IntLiteral;
class ShortIdentifier;
class LongIdentifier;
class SeqLongIdentifier;
class SeqShortIdentifier;
class Pattern;
class Expression;
class Definition;
class Type;
class SeqVariableType;
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class Declaration : public Node {
protected:
  explicit Declaration(SourceLocation);
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class SeqDeclaration : public Declaration,
                       public
                           std::vector<std::shared_ptr<Declaration>>
                           {
public:
  explicit SeqDeclaration(SourceLocation);
  explicit SeqDeclaration(SourceLocation,
      std::shared_ptr<Declaration> First);
  void accept(Visitor *) override;
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class Root : public SeqDeclaration {
private:
  std::shared_ptr<FunctionVals> MainVals;

public:
  explicit Root(SourceLocation);
  explicit Root(SourceLocation, std::shared_ptr<Declaration>
      First);
  void accept(Visitor *) override;

  void setFunctionVals(std::shared_ptr<FunctionVals> V) {
      this->MainVals = V; }
  std::shared_ptr<FunctionVals> getFunctionVals() { return
      this->MainVals; }
};
} // End namespace ast.
} // End namespace ssml.
```

```cpp

namespace ssml {
namespace ast {
class ValDeclaration : public Declaration {
  std::shared_ptr<Pattern> Dest;
  std::shared_ptr<Expression> Source;

public:
  explicit ValDeclaration(SourceLocation L, std::shared_ptr<Pattern> Dest,
                          std::shared_ptr<Expression> Src);
  void accept(Visitor *) override;
  std::shared_ptr<Pattern> getDest() { return Dest; }
  std::shared_ptr<Expression> getSource() { return Source; }
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class OpenDeclaration : public Declaration {
private:
  std::shared_ptr<SeqLongIdentifier> IDs;

public:
  explicit OpenDeclaration(SourceLocation L,
                           std::shared_ptr<SeqLongIdentifier> IDs);
  void accept(Visitor *) override;
  std::shared_ptr<SeqLongIdentifier> getIDs() { return this->IDs; }
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class NonfixDeclaration : public Declaration {
private:
  std::shared_ptr<SeqShortIdentifier> IDs;

public:
  explicit NonfixDeclaration(SourceLocation L,
                             std::shared_ptr<SeqShortIdentifier> IDs);
  void accept(Visitor *) override;
  std::shared_ptr<SeqShortIdentifier> getIDs() { return this->IDs; }
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class InfixDeclaration : public Declaration {
private:
  std::shared_ptr<IntLiteral> Precedence;
  std::shared_ptr<SeqShortIdentifier> Idens;

public:
  explicit InfixDeclaration(SourceLocation L,
                            std::shared_ptr<IntLiteral> Precedence,
                            std::shared_ptr<SeqShortIdentifier> IDs);
```

```
  void accept(Visitor *) override;
  std::shared_ptr<IntLiteral> getPrecedence() { return
      this->Precedence; }
  std::shared_ptr<SeqShortIdentifier> getIDs() { return
      this->Idens; }
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class InfixRDeclaration : public Declaration {
private:
  std::shared_ptr<IntLiteral> Precedence;
  std::shared_ptr<SeqShortIdentifier> Idens;

public:
  explicit InfixRDeclaration(SourceLocation L,
                             std::shared_ptr<IntLiteral>
                                 Precedence,
                             std::shared_ptr<SeqShortIdentifier>
                                 IDs);
  void accept(Visitor *) override;
  std::shared_ptr<IntLiteral> getPrecedence() { return
      this->Precedence; }
  std::shared_ptr<SeqShortIdentifier> getIDs() { return
      this->Idens; }
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class RestrictingSigDefinition {
private:
  std::shared_ptr<Definition> Def;
  bool IsTransparent;

public:
  explicit RestrictingSigDefinition(std::shared_ptr<Definition>
      SigDef,
                                    bool isTransparent);
  std::shared_ptr<Definition> getSigDef() { return this->Def; }
  bool isTransparent() const { return this->IsTransparent; }
  bool isOpaque() const { return !this->isTransparent(); }
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class BareStructDeclaration : public Declaration {
private:
  std::shared_ptr<ShortIdentifier> ID;
  std::shared_ptr<Definition> StructDef;

public:
  explicit BareStructDeclaration(SourceLocation,
                                 std::shared_ptr<ShortIdentifier>
                                     ID,
                                 std::shared_ptr<Definition>
                                     StructDef);
```

```
  void accept(Visitor *) override;
  std::shared_ptr<ShortIdentifier> getID() { return this->ID; }
  std::shared_ptr<Definition> getStructDef() { return this->StructDef; }
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class RestrictedStructDeclaration : public BareStructDeclaration {
private:
  RestrictingSigDefinition SigDef;

public:
  explicit RestrictedStructDeclaration(SourceLocation,
                                       std::shared_ptr<ShortIdentifier> ID,
                                       RestrictingSigDefinition SigDef,
                                       std::shared_ptr<Definition> StructDef);
  void accept(Visitor *) override;

  std::shared_ptr<Definition> getSigDef() { return this->SigDef.getSigDef(); }
  bool isSigTransparent() const { return this->SigDef.isTransparent(); }
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class AbstractValDeclaration : public Declaration {
private:
  std::shared_ptr<ShortIdentifier> ID;
  std::shared_ptr<Type> Typ;

public:
  explicit AbstractValDeclaration(SourceLocation,
                                  std::shared_ptr<ShortIdentifier> ID,
                                  std::shared_ptr<Type> T);
  void accept(Visitor *) override;
  std::shared_ptr<ShortIdentifier> getID() { return this->ID; }
  std::shared_ptr<Type> getType() { return this->Typ; }
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class AbstractTypeDeclaration : public Declaration {
private:
  std::shared_ptr<ShortIdentifier> ID;

public:
  explicit AbstractTypeDeclaration(SourceLocation,
                                   std::shared_ptr<ShortIdentifier> ID);
  void accept(Visitor *) override;
```

```
  std::shared_ptr<ShortIdentifier> getID() { return this->ID; }
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class AbstractStructDeclaration : public Declaration {
private:
  std::shared_ptr<ShortIdentifier> ID;
  std::shared_ptr<Definition> SigDef;

public:
  explicit AbstractStructDeclaration(SourceLocation,
                                     std::shared_ptr<ShortIdentifier>
                                         ID,
                                     std::shared_ptr<Definition>
                                         SigDef);
  void accept(Visitor *) override;
  std::shared_ptr<ShortIdentifier> getID() { return this->ID; }
  std::shared_ptr<Definition> getSigDef() { return this->SigDef; }
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class SharingTypeDeclaration
    : public Declaration,
      public std::vector<std::shared_ptr<LongIdentifier>> {
public:
  explicit SharingTypeDeclaration(SourceLocation,
                                  std::shared_ptr<LongIdentifier>
                                      First);
  void accept(Visitor *) override;
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class SigDeclaration : public Declaration {
private:
  std::shared_ptr<ShortIdentifier> ID;
  std::shared_ptr<Definition> Def;

public:
  explicit SigDeclaration(SourceLocation,
      std::shared_ptr<ShortIdentifier> ID,
                          std::shared_ptr<Definition> Def);
  void accept(Visitor *) override;
  std::shared_ptr<ShortIdentifier> getID() { return this->ID; }
  std::shared_ptr<Definition> getSigDef() { return this->Def; }
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class FunPatternDeclaration : public Declaration {
private:
  std::shared_ptr<Pattern> IDParams;
```

```
  std::shared_ptr<Expression> Def;

public:
  explicit FunPatternDeclaration(SourceLocation,
                                 std::shared_ptr<Pattern>
                                     IDParamaters,
                                 std::shared_ptr<Expression>
                                     FunDef);
  void accept(Visitor *) override;
  std::shared_ptr<Pattern> getIDParams() { return this->IDParams; }
  std::shared_ptr<Expression> getDef() { return this->Def; }
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class FunDeclaration
    : public Declaration,
      public std::vector<std::shared_ptr<FunPatternDeclaration>> {
private:
  std::shared_ptr<FunctionVals> FunVals;

public:
  explicit FunDeclaration(SourceLocation,
                          std::shared_ptr<FunPatternDeclaration>
                              First);
  void accept(Visitor *) override;

  void setFunctionVals(std::shared_ptr<FunctionVals> V) {
      this->FunVals = V; }
  std::shared_ptr<FunctionVals> getFunctionVals() { return
      this->FunVals; }
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class TypeDeclaration : public Declaration {
private:
  std::shared_ptr<ShortIdentifier> ID;
  std::shared_ptr<Type> Typ;

public:
  explicit TypeDeclaration(SourceLocation,
      std::shared_ptr<ShortIdentifier> ID,
                          std::shared_ptr<Type> Typ);
  void accept(Visitor *) override;
  std::shared_ptr<ShortIdentifier> getID() { return this->ID; }
  std::shared_ptr<Type> getType() { return this->Typ; }
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class DatatypeBareInstanceDeclaration : public Declaration {
private:
  std::shared_ptr<ShortIdentifier> ID;

public:
```

```
  explicit DatatypeBareInstanceDeclaration(SourceLocation,
                                           std::shared_ptr<ShortIdentifier> ID);
  void accept(Visitor *) override;
  std::shared_ptr<ShortIdentifier> getID() { return this->ID; }
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class DatatypeTypedInstanceDeclaration
    : public DatatypeBareInstanceDeclaration {
private:
  std::shared_ptr<Type> Typ;

public:
  explicit DatatypeTypedInstanceDeclaration(SourceLocation,
                                            std::shared_ptr<ShortIdentifier> ID,
                                            std::shared_ptr<Type> T);
  void accept(Visitor *) override;
  std::shared_ptr<Type> getType() { return this->Typ; }
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class BareDatatypeDeclaration : public Declaration {
private:
  std::shared_ptr<ShortIdentifier> ID;
  std::shared_ptr<SeqDeclaration> Def;

public:
  BareDatatypeDeclaration(SourceLocation,
      std::shared_ptr<ShortIdentifier> ID,
                          std::shared_ptr<SeqDeclaration> Def);
  void accept(Visitor *) override;
  std::shared_ptr<ShortIdentifier> getID() { return this->ID; }
  std::shared_ptr<SeqDeclaration> getDef() { return this->Def; }
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class TypedDatatypeDeclaration : public BareDatatypeDeclaration {
private:
  std::shared_ptr<SeqVariableType> TypeParams;

public:
  TypedDatatypeDeclaration(SourceLocation,
                           std::shared_ptr<SeqVariableType> TypeParams,
                           std::shared_ptr<ShortIdentifier> ID,
                           std::shared_ptr<SeqDeclaration> Def);
  void accept(Visitor *) override;
  std::shared_ptr<SeqVariableType> getTypeParams() { return this->TypeParams; }
};
```

```
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class ShortBareFunctorDeclaration : public Declaration {
private:
  std::shared_ptr<ShortIdentifier> FunctorID;
  std::shared_ptr<ShortIdentifier> ParamID;
  std::shared_ptr<Definition> ParamSigDef;
  std::shared_ptr<Definition> FunctorStructDef;

public:
  ShortBareFunctorDeclaration(SourceLocation,
                              std::shared_ptr<ShortIdentifier> FunctorID,
                              std::shared_ptr<ShortIdentifier> ParamID,
                              std::shared_ptr<Definition> ParamSig,
                              std::shared_ptr<Definition> StructDef);
  void accept(Visitor *) override;

  std::shared_ptr<ShortIdentifier> getFunctorID() { return this->FunctorID; }

  std::shared_ptr<ShortIdentifier> getParamID() { return this->ParamID; }

  std::shared_ptr<Definition> getParamSigDef() { return this->ParamSigDef; }

  std::shared_ptr<Definition> getFunctorStructDef() {
    return this->FunctorStructDef;
  }
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class ShortRestrictedFunctorDeclaration : public ShortBareFunctorDeclaration {
private:
  RestrictingSigDefinition FunctorSigDef;

public:
  ShortRestrictedFunctorDeclaration(SourceLocation,
                                    std::shared_ptr<ShortIdentifier> FunctorID,
                                    std::shared_ptr<ShortIdentifier> ParamID,
                                    std::shared_ptr<Definition> ParamSig,
                                    RestrictingSigDefinition FunctorSigDef,
                                    std::shared_ptr<Definition> StructDef);
  void accept(Visitor *) override;

  std::shared_ptr<Definition> getFunctorSigDef() {
    return this->FunctorSigDef.getSigDef();
```

```
  }

  bool isFunctorSigDefTransparent() {
    return this->FunctorSigDef.isTransparent();
  }
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class LongBareFunctorDeclaration : public Declaration {
private:
  std::shared_ptr<ShortIdentifier> FunctorID;
  std::shared_ptr<SeqDeclaration> Params;
  std::shared_ptr<Definition> FunctorStructDef;

public:
  LongBareFunctorDeclaration(SourceLocation,
                             std::shared_ptr<ShortIdentifier>
                                 FunctorID,
                             std::shared_ptr<SeqDeclaration>
                                 Params,
                             std::shared_ptr<Definition>
                                 StructDef);
  void accept(Visitor *) override;

  std::shared_ptr<ShortIdentifier> getFunctorID() { return
      this->FunctorID; }

  std::shared_ptr<SeqDeclaration> getParams() { return
      this->Params; }

  std::shared_ptr<Definition> getFunctorStructDef() {
    return this->FunctorStructDef;
  }
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class LongRestrictedFunctorDeclaration : public
    LongBareFunctorDeclaration {
private:
  RestrictingSigDefinition FunctorSigDef;

public:
  LongRestrictedFunctorDeclaration(SourceLocation,
                                   std::shared_ptr<ShortIdentifier>
                                       FunctorID,
                                   std::shared_ptr<SeqDeclaration>
                                       Params,
                                   RestrictingSigDefinition
                                       FunctorSigDef,
                                   std::shared_ptr<Definition>
                                       StructDef);
  void accept(Visitor *) override;

  std::shared_ptr<Definition> getFunctorSigDef() {
    return this->FunctorSigDef.getSigDef();
  }
```

```cpp

  bool isFunctorSigDefTransparent() {
    return this->FunctorSigDef.isTransparent();
  }
};
} // End namespace ast.
} // End namespace ssml.

#endif // SSML_AST_DECLARATION_H
```

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

```cpp
#ifndef SSML_AST_NODE_H
#define SSML_AST_NODE_H

#include "ssml/Common/SourceLocation.h"

namespace ssml {
namespace ast {
class Visitor;
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class Node {
  SourceLocation Location;

protected:
  explicit Node(SourceLocation Loc);

public:
  Node(const Node &) = delete;
  Node &operator=(const Node &) = delete;
  virtual ~Node() = 0;

  virtual void accept(Visitor *) = 0;

  SourceLocation getLocation() const { return Location; }
  void setLocation(SourceLocation L) { this->Location = L; }
};
} // End namespace ast.
} // End namespace ssml.

#endif // SSML_AST_NODE_H
```

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

```cpp
#ifndef SSML_AST_IDENTIFIER_H
#define SSML_AST_IDENTIFIER_H

#include "Node.h"

#include <vector>
#include <string>
#include <memory>

namespace ssml {
namespace ast {
class IntLiteral;
} // End namespace ast.
```

```cpp
} // End namespace ssml.

namespace ssml {
namespace ast {
class Identifier : public Node {
protected:
  Identifier(SourceLocation);

public:
  virtual std::string toString() = 0;
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class ShortIdentifier : public Identifier {
private:
  std::shared_ptr<std::string> ID;

public:
  ShortIdentifier(SourceLocation, std::shared_ptr<std::string> ID);
  void accept(Visitor *) override;
  std::string toString() override;
  std::shared_ptr<const std::string> getID() const { return this->ID; }
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class LongIdentifier : public Identifier,
                       public std::vector<std::shared_ptr<ShortIdentifier>> {
public:
  LongIdentifier(SourceLocation, std::shared_ptr<ShortIdentifier> First);
  void accept(Visitor *) override;
  std::string toString() override;
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class IntIdentifier : public Identifier {
private:
  std::shared_ptr<IntLiteral> Literal;

public:
  IntIdentifier(SourceLocation, std::shared_ptr<IntLiteral> Literal);
  void accept(Visitor *) override;
  std::string toString() override;
  std::shared_ptr<IntLiteral> getLiteral() { return this->Literal; }
};
} // End namespace ast.
} // End namespace ssml.
```

```
namespace ssml {
namespace ast {
class SeqShortIdentifier
    : public Identifier,
      public std::vector<std::shared_ptr<ShortIdentifier>> {
public:
  SeqShortIdentifier(SourceLocation,
      std::shared_ptr<ShortIdentifier> First);
  void accept(Visitor *) override;
  std::string toString() override;
};
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class SeqLongIdentifier
    : public Identifier,
      public std::vector<std::shared_ptr<LongIdentifier>> {
public:
  SeqLongIdentifier(SourceLocation,
      std::shared_ptr<LongIdentifier> First);
  void accept(Visitor *) override;
  std::string toString() override;
};
} // End namespace ast.
} // End namespace ssml.

#endif // SSML_AST_IDENTIFIER_H
```

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

```
#ifndef SSML_AST_DUMPVISITOR_H
#define SSML_AST_DUMPVISITOR_H

#include "Visitor.h"

#include "ssml/Common/SourceLocation.h"

#include <memory>

namespace ssml {
namespace ast {
class Node;
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class DumpVisitor : public Visitor {
  unsigned Indent = 0;

private:
  void printIndent();
  void incrementIndent();
  void decrementIndent();
  void printAttribute(const char *Name, const std::string &Value);
  void printAttribute(const char *Name,
                      std::shared_ptr<const std::string> Value);
  void openBeginTag(const char *Tag, SourceLocation L);
  void closeBeginTag();
```

```
  void openCloseEndTag(const char *Tag);

public:
  void visit(IntLiteral *) override;
  void visit(RealLiteral *) override;
  void visit(CharLiteral *) override;
  void visit(StringLiteral *) override;

  void visit(ShortIdentifier *) override;
  void visit(LongIdentifier *) override;
  void visit(IntIdentifier *) override;
  void visit(SeqShortIdentifier *) override;
  void visit(SeqLongIdentifier *) override;

  void visit(Match *) override;
  void visit(SeqMatch *) override;

  void visit(SeqExpression *) override;
  void visit(LiteralExpression *) override;
  void visit(LongIdentifierExpression *) override;
  void visit(TupleExpression *) override;
  void visit(ListExpression *) override;
  void visit(IdentifierEqualsExpression *) override;
  void visit(RecordExpression *) override;
  void visit(SelectorExpression *) override;
  void visit(ApplyExpression *) override;
  void visit(TypeExpression *) override;
  void visit(OrElseExpression *) override;
  void visit(AndAlsoExpression *) override;
  void visit(LetExpression *) override;
  void visit(IfExpression *) override;
  void visit(WhileExpression *) override;
  void visit(CaseExpression *) override;
  void visit(LambdaExpression *) override;

  void visit(SeqPattern *) override;
  void visit(LiteralPattern *) override;
  void visit(WildcardPattern *) override;
  void visit(ShortIdentifierPattern *) override;
  void visit(LongIdentifierPattern *) override;
  void visit(TypePattern *) override;
  void visit(AsPattern *) override;
  void visit(ApplyPattern *) override;
  void visit(IdentifierEqualsPattern *) override;
  void visit(RecordPattern *) override;
  void visit(ListPattern *) override;
  void visit(TuplePattern *) override;

  void visit(SeqDeclaration *) override;
  void visit(Root *) override;
  void visit(ValDeclaration *) override;
  void visit(OpenDeclaration *) override;
  void visit(NonfixDeclaration *) override;
  void visit(InfixDeclaration *) override;
  void visit(InfixRDeclaration *) override;
  void visit(RestrictedStructDeclaration *) override;
  void visit(BareStructDeclaration *) override;
  void visit(AbstractValDeclaration *) override;
  void visit(AbstractTypeDeclaration *) override;
  void visit(AbstractStructDeclaration *) override;
  void visit(SharingTypeDeclaration *) override;
  void visit(SigDeclaration *) override;
```

```
  void visit(FunPatternDeclaration *) override;
  void visit(FunDeclaration *) override;
  void visit(TypeDeclaration *) override;
  void visit(DatatypeBareInstanceDeclaration *) override;
  void visit(DatatypeTypedInstanceDeclaration *) override;
  void visit(BareDatatypeDeclaration *) override;
  void visit(TypedDatatypeDeclaration *) override;
  void visit(ShortBareFunctorDeclaration *) override;
  void visit(ShortRestrictedFunctorDeclaration *) override;
  void visit(LongBareFunctorDeclaration *) override;
  void visit(LongRestrictedFunctorDeclaration *) override;

  void visit(LongIdentifierDefinition *) override;
  void visit(StructDefinition *) override;
  void visit(AnnotationDefinition *) override;
  void visit(ShortFunctorDefinition *) override;
  void visit(LongFunctorDefinition *) override;
  void visit(SigDefinition *) override;

  void visit(LongIdentifierType *) override;
  void visit(VariableType *) override;
  void visit(SeqType *) override;
  void visit(SeqVariableType *) override;
  void visit(TupleType *) override;
  void visit(IdentifierColonType *) override;
  void visit(RecordType *) override;
  void visit(ApplyType *) override;
  void visit(ProductType *) override;
  void visit(FunctionType *) override;
};
} // End namespace ast.
} // End namespace ssml.

#endif // SSML_AST_DUMPVISITOR_H
```

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

```
#ifndef SSML_AST_VISITOR_H
#define SSML_AST_VISITOR_H

namespace ssml {
namespace ast {
class IntLiteral;
class RealLiteral;
class CharLiteral;
class StringLiteral;

class ShortIdentifier;
class LongIdentifier;
class IntIdentifier;
class SeqShortIdentifier;
class SeqLongIdentifier;

class Match;
class SeqMatch;

class SeqExpression;
class LiteralExpression;
class LongIdentifierExpression;
class TupleExpression;
class ListExpression;
```

```
class IdentifierEqualsExpression;
class RecordExpression;
class SelectorExpression;
class ApplyExpression;
class TypeExpression;
class OrElseExpression;
class AndAlsoExpression;
class LetExpression;
class IfExpression;
class WhileExpression;
class CaseExpression;
class LambdaExpression;

class SeqPattern;
class LiteralPattern;
class WildcardPattern;
class ShortIdentifierPattern;
class LongIdentifierPattern;
class TypePattern;
class AsPattern;
class ApplyPattern;
class IdentifierEqualsPattern;
class RecordPattern;
class ListPattern;
class TuplePattern;

class SeqDeclaration;
class Root;
class ValDeclaration;
class OpenDeclaration;
class NonfixDeclaration;
class InfixDeclaration;
class InfixRDeclaration;
class RestrictedStructDeclaration;
class BareStructDeclaration;
class AbstractValDeclaration;
class AbstractTypeDeclaration;
class AbstractStructDeclaration;
class SharingTypeDeclaration;
class SigDeclaration;
class FunPatternDeclaration;
class FunDeclaration;
class TypeDeclaration;
class DatatypeBareInstanceDeclaration;
class DatatypeTypedInstanceDeclaration;
class BareDatatypeDeclaration;
class TypedDatatypeDeclaration;
class ShortBareFunctorDeclaration;
class ShortRestrictedFunctorDeclaration;
class LongBareFunctorDeclaration;
class LongRestrictedFunctorDeclaration;

class LongIdentifierDefinition;
class StructDefinition;
class AnnotationDefinition;
class ShortFunctorDefinition;
class LongFunctorDefinition;
class SigDefinition;

class LongIdentifierType;
class VariableType;
class SeqType;
```

```
class SeqVariableType;
class TupleType;
class IdentifierColonType;
class RecordType;
class ApplyType;
class ProductType;
class FunctionType;
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class Visitor {
public:
  virtual ~Visitor() = 0;

  virtual void visit(IntLiteral *) = 0;
  virtual void visit(RealLiteral *);
  virtual void visit(CharLiteral *);
  virtual void visit(StringLiteral *);

  virtual void visit(ShortIdentifier *) = 0;
  virtual void visit(LongIdentifier *) = 0;
  virtual void visit(IntIdentifier *);
  virtual void visit(SeqShortIdentifier *) = 0;
  virtual void visit(SeqLongIdentifier *) = 0;

  virtual void visit(Match *) = 0;
  virtual void visit(SeqMatch *) = 0;

  virtual void visit(SeqExpression *) = 0;
  virtual void visit(LiteralExpression *) = 0;
  virtual void visit(LongIdentifierExpression *) = 0;
  virtual void visit(TupleExpression *) = 0;
  virtual void visit(ListExpression *) = 0;
  virtual void visit(IdentifierEqualsExpression *);
  virtual void visit(RecordExpression *);
  virtual void visit(SelectorExpression *);
  virtual void visit(ApplyExpression *) = 0;
  virtual void visit(TypeExpression *);
  virtual void visit(OrElseExpression *) = 0;
  virtual void visit(AndAlsoExpression *) = 0;
  virtual void visit(IfExpression *) = 0;
  virtual void visit(LetExpression *) = 0;
  virtual void visit(WhileExpression *) = 0;
  virtual void visit(CaseExpression *);
  virtual void visit(LambdaExpression *) = 0;

  virtual void visit(SeqPattern *) = 0;
  virtual void visit(LiteralPattern *) = 0;
  virtual void visit(WildcardPattern *) = 0;
  virtual void visit(ShortIdentifierPattern *);
  virtual void visit(LongIdentifierPattern *) = 0;
  virtual void visit(TypePattern *) = 0;
  virtual void visit(AsPattern *);
  virtual void visit(ApplyPattern *) = 0;
  virtual void visit(IdentifierEqualsPattern *);
  virtual void visit(RecordPattern *);
  virtual void visit(ListPattern *) = 0;
  virtual void visit(TuplePattern *) = 0;

  virtual void visit(SeqDeclaration *) = 0;
```

```
  virtual void visit(Root *) = 0;
  virtual void visit(ValDeclaration *) = 0;
  virtual void visit(OpenDeclaration *);
  virtual void visit(NonfixDeclaration *) = 0;
  virtual void visit(InfixDeclaration *) = 0;
  virtual void visit(InfixRDeclaration *) = 0;
  virtual void visit(RestrictedStructDeclaration *);
  virtual void visit(BareStructDeclaration *);
  virtual void visit(AbstractValDeclaration *);
  virtual void visit(AbstractTypeDeclaration *);
  virtual void visit(AbstractStructDeclaration *);
  virtual void visit(SharingTypeDeclaration *);
  virtual void visit(SigDeclaration *);
  virtual void visit(FunPatternDeclaration *) = 0;
  virtual void visit(FunDeclaration *) = 0;
  virtual void visit(TypeDeclaration *);
  virtual void visit(DatatypeBareInstanceDeclaration *) = 0;
  virtual void visit(DatatypeTypedInstanceDeclaration *) = 0;
  virtual void visit(BareDatatypeDeclaration *) = 0;
  virtual void visit(TypedDatatypeDeclaration *) = 0;
  virtual void visit(ShortBareFunctorDeclaration *);
  virtual void visit(ShortRestrictedFunctorDeclaration *);
  virtual void visit(LongBareFunctorDeclaration *);
  virtual void visit(LongRestrictedFunctorDeclaration *);

  virtual void visit(LongIdentifierDefinition *);
  virtual void visit(StructDefinition *);
  virtual void visit(AnnotationDefinition *);
  virtual void visit(ShortFunctorDefinition *);
  virtual void visit(LongFunctorDefinition *);
  virtual void visit(SigDefinition *);

  virtual void visit(LongIdentifierType *) = 0;
  virtual void visit(VariableType *) = 0;
  virtual void visit(SeqType *) = 0;
  virtual void visit(SeqVariableType *) = 0;
  virtual void visit(TupleType *) = 0;
  virtual void visit(IdentifierColonType *);
  virtual void visit(RecordType *);
  virtual void visit(ApplyType *) = 0;
  virtual void visit(ProductType *) = 0;
  virtual void visit(FunctionType *) = 0;
};
} // End namespace ast.
} // End namespace ssml.

#endif // SSML_AST_VISITOR_H
```

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

```
#ifndef SSML_TYPECHECK_TYPECHECK_H
#define SSML_TYPECHECK_TYPECHECK_H

namespace ssml {
namespace ast {
class Root;
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace typecheck {
```

```
void typecheck(ssml::ast::Root *);
} // End namespace typecheck.
} // End namespace ssml.

#endif // SSML_TYPECHECK_TYPECHECK_H
```

Listing 38: lib/Codegen/TypeMap.cpp

```
#include "TypeMap.h"

#include "ssml/Common/FatalExit.h"

#include <cassert>

using namespace ssml::codegen;

void TypeInstanceMap::insert(const std::string &In, int64_t V,
    bool B) {
  Maps.back()[In] = {V, B};
}

std::pair<int64_t, bool> TypeInstanceMap::get(const std::string
    &In, bool ANo) {
  for (auto It = Maps.rbegin(), End = Maps.rend(); It != End; ++It)
    if (It->count(In))
      return It->find(In)->second;
  if (!ANo)
    fatalExit("value " + In + " not in type map");
  return {-1, false};
}

void TypeInstanceMap::enterScope() { Maps.push_back(MapType()); }

void TypeInstanceMap::leaveScope() { Maps.pop_back(); }
```

Listing 39: lib/Codegen/NameMap.h

```
#ifndef LLVM_TOOLS_SSML_LIB_CODEGEN_NAMEMAP_H
#define LLVM_TOOLS_SSML_LIB_CODEGEN_NAMEMAP_H

#include <map>
#include <vector>
#include <string>

namespace llvm {
class Value;
} // End namespace llvm.

namespace ssml {
namespace codegen {
class NameMap {
public:
  struct ScopeValue {
    int32_t Scope;          // Which function scope. Starting from
        0.
    int32_t Index;          // Index to function variables.

    bool isTypeInstance() const { return this->Scope >= 0 &&
        this->Index < 0; }
    bool isExisting() const { return this->Scope >= 0; }
  };
```

```

  using MapType = std::map<std::string, ScopeValue>;

private:
  std::vector<MapType> Maps;

public:
  void insert(const std::string &K, ScopeValue V);
  ScopeValue get(const std::string &K);
  void enterScope();
  void leaveScope();
};
} // End namespace codegen.
} // End namespace ssml.

#endif // LLVM_TOOLS_SSML_LIB_CODEGEN_NAMEMAP_H
```

Listing 40: lib/Codegen/CodegenVisitor.h

```
#ifndef LLVM_TOOLS_SSML_LIB_CODEGEN_CODEGENVISITOR_H
#define LLVM_TOOLS_SSML_LIB_CODEGEN_CODEGENVISITOR_H

#include "TypeMap.h"
#include "NameMap.h"

#include "ssml/AST/FunctionVals.h"
#include "ssml/AST/Visitor.h"

#include "llvm/Support/raw_ostream.h"
#include "llvm/IR/Instructions.h"
#include "llvm/IR/InstrTypes.h"

#include <cstdlib>
#include <vector>
#include <map>
#include <string>
#include <memory>

#if !defined(SSML_DISABLE_GC) && !defined(SSML_MARK_SWEEP_GC)
#define SSML_VOLATILE_MEMORY
#endif

namespace llvm {
class Value;
class Module;
class Function;
class BasicBlock;
class AllocaInst;
class ConstantInt;
} // End namespace llvm.

namespace ssml {
namespace ast {
class Node;
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace codegen {
class FuncContext {
private:
```

```
  llvm::Function *Fun;
  FuncContext *Prev;
  std::vector<std::map<std::string, int32_t>> LocalOffsets;
  std::vector<std::map<std::string, int32_t>> LocalOffsetsRemembered;
  llvm::AllocaInst *LocalPtr = nullptr;
  std::vector<std::vector<size_t>> ScopeLocalIndicess;
  std::vector<std::vector<llvm::Value *>> ScopeTempss;
  int32_t FramePtrIndex;
  int32_t LocalMemSize;
  int32_t MaxLocalMemSize;
  int32_t LocalMemSizeRemembered;
  std::shared_ptr<ssml::ast::FunctionVals> FunVals;
  std::pair<std::string, llvm::Function *> CurrFuncPair;
  bool IsStepFunction = false;
  bool IsInsideLetDecl = false;
  llvm::Value *FirstArg = nullptr;
  llvm::Value *SecondArg = nullptr;

private:
  int32_t getNest(int32_t Depth);
  int32_t getTrueNest(int32_t Depth);

public:
  explicit FuncContext(llvm::Function *F,
                       std::shared_ptr<ssml::ast::FunctionVals> Vs,
                       const std::string &Name = "",
                       llvm::Function *CurrFunc = nullptr)
      : Fun(F), Prev(0), LocalOffsets{std::map<std::string, int32_t>()},
        LocalPtr(0), LocalMemSize(0), MaxLocalMemSize(0),
        LocalMemSizeRemembered(0), FunVals(Vs), CurrFuncPair{Name, CurrFunc} {
    this->ScopeLocalIndicess.emplace_back();
    this->ScopeTempss.emplace_back();
    this->FramePtrIndex = 0;
    this->LocalMemSize += this->FramePtrIndex * sizeof(int64_t);
    this->MaxLocalMemSize =
        (this->FramePtrIndex + 1 + this->FunVals->getFlags().size()) *
        sizeof(int64_t);
  }

  void setIsStepFunction() { this->IsStepFunction = true; }

  bool getIsStepFunction() { return this->IsStepFunction; }

  const std::string &currFuncName() { return this->CurrFuncPair.first; }

  llvm::Function *currFunc() { return this->CurrFuncPair.second; }

  void setInsideLetDecl(bool B) { this->IsInsideLetDecl = B; }
  bool getInsideLetDecl() { return this->IsInsideLetDecl; }

  int32_t getFramePtrIndex() { return this->FramePtrIndex; }

  void appendLocalIndex(size_t I) {
    assert(this->ScopeLocalIndicess.size() > 0);
    this->ScopeLocalIndicess.back().push_back(I);
  }
  void appendTemp(llvm::Value *Temp, size_t Idx = 0) {
```

```
    if (this->ScopeTempss.size() == 1 && Idx == 1) {
      this->ScopeTempss.back().push_back(Temp);
    } else {
      assert(this->ScopeTempss.size() > Idx);
      this->ScopeTempss[this->ScopeTempss.size() - 1 -
          Idx].push_back(Temp);
    }
  }
  std::vector<size_t> &getScopeLocalIndices() {
    return this->ScopeLocalIndicess.back();
  }
  std::vector<llvm::Value *> &getScopeTemps() {
    return this->ScopeTempss.back();
  }

  void func(llvm::Function *F) { this->Fun = F; }

  llvm::Function *func() { return this->Fun; }

  void setPrev(FuncContext *F) { this->Prev = F; }

  FuncContext *getPrev() { return this->Prev; }

  size_t getLocalMemSize() { return this->LocalMemSize; }

  void enterScope();

  void leaveScope();

  void addLocal(const std::string &Name);

  int32_t getLocalIndex(const std::string &Name);

  void setLocalPtr(llvm::AllocaInst *In) { this->LocalPtr = In; }
  llvm::Value *loadLocalPtr(llvm::BasicBlock *B) {
#ifdef SSML_VOLATILE_MEMORY
    return new llvm::LoadInst(this->LocalPtr, "frame", true, B);
#else
    return new llvm::LoadInst(this->LocalPtr, "frame", false, B);
#endif
  }

  llvm::AllocaInst *getPostTempsInst() { return this->LocalPtr; }

  int32_t getNest() { return this->getNest(0); }

  int32_t getTrueNest() { return this->getTrueNest(0); }

  int32_t getMaxLocalMemSize() { return this->MaxLocalMemSize; }

  void setFirstArg(llvm::Value *FirstArg) { this->FirstArg =
      FirstArg; }

  void setSecondArg(llvm::Value *SecondArg) { this->SecondArg =
      SecondArg; }

  llvm::Value *getFirstArg() { return this->FirstArg; }

  llvm::Value *getSecondArg() { return this->SecondArg; }

  void rememberContext();

```

```
  void restoreRememberedContext();
};
} // End namespace codegen.
} // End namespace ssml.

namespace ssml {
namespace codegen {
class CodegenVisitor : public ssml::ast::Visitor {
private:
  FuncContext *FunContext = nullptr;
  llvm::Module *Module;
  NameMap Names;
  TypeInstanceMap Instances;
  std::shared_ptr<const std::string> DatatypeTypename;
  llvm::BasicBlock *BasicBlock = nullptr;
  llvm::BasicBlock *TrueBlock = nullptr;
  llvm::BasicBlock *FalseBlock = nullptr;
  llvm::Value *PatternExpr = nullptr;
  llvm::Value *Temp = nullptr;
  int64_t DatatypeInstanceValue = -1;
  bool ExprWasTypeInstance = false;
  bool InsideLetDecls = false;
  bool LastExpr = false;

private:
  void pushFunc(llvm::Function *F, std::shared_ptr<ssml::ast::FunctionVals>,
                const std::string &Name, llvm::Function *CurrFunc);
  void pushFunc(llvm::Function *F, std::shared_ptr<ssml::ast::FunctionVals> V) {
    this->pushFunc(F, V, "", nullptr);
  }
  void changeFunc(llvm::Function *F);
  FuncContext *peekFunc();
  void popFunc();

  llvm::Module *mod() { return this->Module; }

  llvm::Value *loadLocalPtr() {
    return this->peekFunc()->loadLocalPtr(this->BasicBlock);
  }

  void beginMain(std::shared_ptr<ssml::ast::FunctionVals>);
  void endMain();

  void addFuncEntry(FuncContext *F, bool IsMain, bool PrevFrameIsLocals);
  void fixupFuncEntry(FuncContext *);
  llvm::CallInst *addAllocateCall(llvm::BasicBlock *B, llvm::Value *Size);
  void addGCStore(llvm::Value *Source, llvm::Value *Dest, llvm::Value *Base);
  llvm::Value *addGCLoad(llvm::Value *Source, llvm::Value *Base);
  llvm::Value *addLocalLookup(llvm::BasicBlock *B, NameMap::ScopeValue SV);
  void addLocalFuncVal(const std::string &Identifier, llvm::Value *Val);
  void insertTypeInstances();

  void enterScope();
  void leaveScope(bool ZeroUnusedVals);

```

```
  void doVisit(std::shared_ptr<ssml::ast::Node>);

  bool tryBuiltinApplyExpr(ssml::ast::ApplyExpression *ApplyExpr);

  void nextTemp();
  void storeNextTemp(llvm::Value *);
  llvm::Value *loadTemp(llvm::Value *Temp = nullptr);
  void setExprResult(llvm::Value *V) { this->Temp = V; }
  llvm::Value *getExprResult() { return this->Temp; }

public:
  explicit CodegenVisitor();
  ~CodegenVisitor();

  void visit(ssml::ast::IntLiteral *) override;

  void visit(ssml::ast::ShortIdentifier *) override;
  void visit(ssml::ast::LongIdentifier *) override;
  void visit(ssml::ast::SeqShortIdentifier *) override;
  void visit(ssml::ast::SeqLongIdentifier *) override;

  void visit(ssml::ast::Match *) override;
  void visit(ssml::ast::SeqMatch *) override;

  void visit(ssml::ast::SeqExpression *) override;
  void visit(ssml::ast::LiteralExpression *) override;
  void visit(ssml::ast::LongIdentifierExpression *) override;
  void visit(ssml::ast::TupleExpression *) override;
  void visit(ssml::ast::ListExpression *) override;
  void visit(ssml::ast::ApplyExpression *) override;
  void visit(ssml::ast::OrElseExpression *) override;
  void visit(ssml::ast::AndAlsoExpression *) override;
  void visit(ssml::ast::LetExpression *) override;
  void visit(ssml::ast::IfExpression *) override;
  void visit(ssml::ast::WhileExpression *) override;
  void visit(ssml::ast::LambdaExpression *) override;

  void visit(ssml::ast::SeqPattern *) override;
  void visit(ssml::ast::LiteralPattern *) override;
  void visit(ssml::ast::WildcardPattern *) override;
  void visit(ssml::ast::LongIdentifierPattern *) override;
  void visit(ssml::ast::TypePattern *) override;
  void visit(ssml::ast::ApplyPattern *) override;
  void visit(ssml::ast::ListPattern *) override;
  void visit(ssml::ast::TuplePattern *) override;

  void visit(ssml::ast::SeqDeclaration *) override;
  void visit(ssml::ast::Root *) override;
  void visit(ssml::ast::ValDeclaration *) override;
  void visit(ssml::ast::NonfixDeclaration *) override;
  void visit(ssml::ast::InfixDeclaration *) override;
  void visit(ssml::ast::InfixRDeclaration *) override;
  void visit(ssml::ast::FunPatternDeclaration *) override;
  void visit(ssml::ast::FunDeclaration *) override;
  void visit(ssml::ast::DatatypeBareInstanceDeclaration *) override;
  void visit(ssml::ast::DatatypeTypedInstanceDeclaration *) override;
  void visit(ssml::ast::BareDatatypeDeclaration *) override;
  void visit(ssml::ast::TypedDatatypeDeclaration *) override;

  void visit(ssml::ast::LongIdentifierType *) override;
```

```
  void visit(ssml::ast::VariableType *) override;
  void visit(ssml::ast::SeqType *) override;
  void visit(ssml::ast::SeqVariableType *) override;
  void visit(ssml::ast::TupleType *) override;
  void visit(ssml::ast::ApplyType *) override;
  void visit(ssml::ast::ProductType *) override;
  void visit(ssml::ast::FunctionType *) override;
};
} // End namespace codegen.
} // End namespace ssml.

#endif // LLVM_TOOLS_SSML_LIB_CODEGEN_CODEGENVISITOR_H
```

Listing 41: lib/Codegen/TypeMap.h

```
#ifndef LLVM_TOOLS_SSML_LIB_CODEGEN_TYPEMAP_H
#define LLVM_TOOLS_SSML_LIB_CODEGEN_TYPEMAP_H

#include <vector>
#include <map>
#include <string>

namespace llvm {
class StructType;
} // End namespace llvm.

namespace ssml {
namespace codegen {
class TypeInstanceMap {
public:
  using MapType = std::map<std::string, std::pair<int64_t, bool>>;

private:
  std::vector<MapType> Maps;

public:
  void insert(const std::string &Instance, int64_t Value, bool
      IsTyped);
  std::pair<int64_t, bool> get(const std::string &Instance,
                               bool AllowNotInMap = false);
  void enterScope();
  void leaveScope();

  int64_t getValue(const std::string &Instance) {
    return this->get(Instance).first;
  }
  int64_t getIsTyped(const std::string &Instance) {
    return this->get(Instance).second;
  }
};
} // End namespace codegen.
} // End namespace ssml.

#endif // LLVM_TOOLS_SSML_LIB_CODEGEN_TYPEMAP_H
```

Listing 42: lib/Codegen/Codegen.cpp

```
#include "ssml/Codegen/Codegen.h"
#include "CodegenVisitor.h"

#include "ssml/AST/Declaration.h"
```

```
void ssml::codegen::codegen(ssml::ast::Root *R) {
  CodegenVisitor V;
  R->accept(&V);
}
```

Listing 43: lib/Codegen/CodegenVisitor.cpp

```
#define DEBUG_TYPE "codegen"

#include "CodegenVisitor.h"

#include "ssml/AST/All.h"
#include "ssml/Common/FatalExit.h"

#include "llvm/IR/Verifier.h"
#include "llvm/IR/Function.h"
#include "llvm/IR/BasicBlock.h"
#include "llvm/IR/Instructions.h"
#include "llvm/IR/CallingConv.h"
#include "llvm/IR/Module.h"
#include "llvm/IR/LLVMContext.h"
#include "llvm/IR/Constants.h"
#include "llvm/Support/raw_ostream.h"
#include "llvm/Support/ToolOutputFile.h"
#include "llvm/Support/FileSystem.h"
#include "llvm/Bitcode/ReaderWriter.h"

#include "llvm/Support/raw_ostream.h"
#include "llvm/Support/Debug.h"

#include <system_error>

#define CODEGEN_DLOG(msg) DEBUG(llvm::errs() << DEBUG_TYPE ":: "
    << msg << '\n')

using namespace ssml;
using namespace ssml::codegen;
using namespace ssml::ast;

#if defined(SSML_SHADOW_STACK_GC)
static const char gcstrategy[] = "shadow-stack";
#elif !defined(SSML_DISABLE_GC)
static const char gcstrategy[] = "ssml";
#else
static const char gcstrategy[] = "";
#endif

#if !defined(SSML_DISABLE_GC)
static void setGC(llvm::Function *F) { F->setGC(gcstrategy); }
#else  // defined(SSML_DISABLE_GC)
static void setGC(llvm::Function *F) {}
#endif // !defined(SSML_DISABLE_GC)

static const size_t alignment = sizeof(int64_t);

static llvm::IntegerType *getInteger(llvm::Module *M, int32_t I) {
  return llvm::IntegerType::get(M->getContext(), I);
}

static llvm::IntegerType *getInteger64(llvm::Module *M) {
```

```
  return llvm::IntegerType::get(M->getContext(), 64);
}

static llvm::PointerType *getPointerTo(llvm::Type *V)
    __attribute__((unused));
static llvm::PointerType *getPointerTo(llvm::Type *V) {
  return llvm::PointerType::getUnqual(V);
}

static llvm::PointerType *getInteger64Pointer(llvm::Module *M) {
  return llvm::PointerType::getUnqual(getInteger64(M));
}

static llvm::ConstantInt *getConst(llvm::Module *M, int64_t val,
    int32_t Bits) {
  return llvm::ConstantInt::getSigned(getInteger(M, Bits), val);
}

static llvm::ConstantInt *getConst(llvm::Module *M, int64_t val) {
  return getConst(M, val, 64);
}

static llvm::ConstantInt *getZeroConst(llvm::Module *M) {
  return getConst(M, 0);
}

static int64_t markValue(int64_t Val) { return Val * 2 | 1; }

static llvm::Value *markValue(llvm::Module *M, llvm::Value *V,
                              llvm::BasicBlock *B) {
  auto Sh = llvm::BinaryOperator::Create(llvm::Instruction::Shl, V,
                                         getConst(M, 1), "", B);
  return llvm::BinaryOperator::CreateOr(Sh, getConst(M, 1), "", B);
}
static llvm::Value *unmarkValue(llvm::Module *M, llvm::Value *V,
                                llvm::BasicBlock *B) {
  return llvm::BinaryOperator::Create(llvm::Instruction::AShr, V,
                                      getConst(M, 1), "", B);
}

static llvm::Value *getSimpleBuiltinConst(llvm::Module *M,
                                          const std::string &ID,
                                              bool Marked) {
  if (ID == "false")
    return Marked ? getConst(M, markValue(0)) : getConst(M, 0);
  else if (ID == "true")
    return Marked ? getConst(M, markValue(1)) : getConst(M, 1);
  return nullptr;
}

static llvm::Value *getTernaryBuintinFunc(llvm::Module *M,
                                          const std::string &ID) {
  llvm::Value *Ret = nullptr;
  if (ID == "Array.update") {
    Ret = M->getGlobalVariable("frameArrayupdate");
    assert(!!Ret);
  }
  return Ret;
}

static llvm::Value *getBinaryBuintinFunc(llvm::Module *M,
                                         const std::string &ID) {
```

```
  llvm::Value *Ret = nullptr;
  if (ID == "+") {
    Ret = M->getGlobalVariable("framePlus");
    assert(!!Ret);
  } else if (ID == "-") {
    Ret = M->getGlobalVariable("frameMinus");
    assert(!!Ret);
  } else if (ID == "*") {
    Ret = M->getGlobalVariable("frameMultiply");
    assert(!!Ret);
  } else if (ID == "div") {
    Ret = M->getGlobalVariable("frameDivision");
    assert(!!Ret);
  } else if (ID == "mod") {
    Ret = M->getGlobalVariable("frameModulo");
    assert(!!Ret);
  } else if (ID == ":=") {
    Ret = M->getGlobalVariable("frameRefassign");
    assert(!!Ret);
  } else if (ID == "Array.array") {
    Ret = M->getGlobalVariable("frameArray");
    assert(!!Ret);
  } else if (ID == "Array.get") {
    Ret = M->getGlobalVariable("frameArrayget");
    assert(!!Ret);
  } else if (ID == "=") {
    Ret = M->getGlobalVariable("frameEquals");
    assert(!!Ret);
  } else if (ID == "<>") {
    Ret = M->getGlobalVariable("frameNotEquals");
    assert(!!Ret);
  } else if (ID == "<") {
    Ret = M->getGlobalVariable("frameLess");
    assert(!!Ret);
  } else if (ID == ">") {
    Ret = M->getGlobalVariable("frameGreater");
    assert(!!Ret);
  } else if (ID == "<=") {
    Ret = M->getGlobalVariable("frameLessEquals");
    assert(!!Ret);
  } else if (ID == ">=") {
    Ret = M->getGlobalVariable("frameGreaterEquals");
    assert(!!Ret);
  }
  return Ret;
}

static llvm::Value *getUnaryBuintinFunc(llvm::Module *M,
                                        const std::string &ID) {
  llvm::Value *Ret = nullptr;
  if (ID == "print") {
    Ret = M->getGlobalVariable("framePrintVal");
    assert(!!Ret);
  } else if (ID == "~") {
    Ret = M->getGlobalVariable("frameNegate");
    assert(!!Ret);
  } else if (ID == "not") {
    Ret = M->getGlobalVariable("frameNot");
    assert(!!Ret);
  } else if (ID == "ref") {
    Ret = M->getGlobalVariable("frameRef");
    assert(!!Ret);
```

```
  } else if (ID == "Array.length") {
    Ret = M->getGlobalVariable("frameArraylength");
    assert(!!Ret);
  } else if (ID == "!") {
    Ret = M->getGlobalVariable("frameDeref");
    assert(!!Ret);
  }
  return Ret;
}

static llvm::Value *getBuiltinValue(llvm::Module *M, const std::string &ID) {
  llvm::Value *Ret = nullptr;
  if (ID == "Array.empty") {
    Ret = M->getGlobalVariable("arrayempty");
    assert(!!Ret);
  }
  return Ret;
}

static llvm::Value *getBuiltin(llvm::Module *M, const std::string &ID) {
  llvm::Value *Ret = getTernaryBuintinFunc(M, ID);
  if (!Ret)
    Ret = getBinaryBuintinFunc(M, ID);
  if (!Ret)
    Ret = getUnaryBuintinFunc(M, ID);
  if (!Ret)
    Ret = getBuiltinValue(M, ID);
  return Ret;
}

static llvm::FunctionType *getStandardFuncType(llvm::Module *M) {
  auto RetType = getInteger64(M);
  llvm::SmallVector<llvm::Type *, 2> Arg;
  Arg.push_back(getInteger64(M));
  Arg.push_back(getInteger64(M));
  auto FunType = llvm::FunctionType::get(RetType, Arg, false);
  return FunType;
}

int32_t FuncContext::getNest(int32_t Depth) {
  if (this->Prev)
    return this->Prev->getNest(Depth + 1);
  return Depth;
}

int32_t FuncContext::getTrueNest(int32_t Depth) {
  if (this->Prev) {
    if (this->currFunc())
      return this->Prev->Prev->getNest(Depth + 1);
    else
      return this->Prev->getNest(Depth + 1);
  }
  return Depth;
}

void FuncContext::enterScope() {
  this->ScopeLocalIndicess.emplace_back();
  this->ScopeTempss.emplace_back();
  this->LocalOffsets.emplace_back();
}
```

```
void FuncContext::leaveScope() {
  this->LocalOffsets.pop_back();
  this->ScopeLocalIndicess.pop_back();
  this->ScopeTempss.pop_back();
}

void FuncContext::addLocal(const std::string &Name) {
  this->LocalOffsets.back()[Name] = this->LocalMemSize;
  this->LocalMemSize += alignment;
  if (this->LocalMemSize > this->MaxLocalMemSize)
    fatalExit("too many local variables in function");
}

int32_t FuncContext::getLocalIndex(const std::string &Name) {
  for (auto It = LocalOffsets.rbegin(), End = LocalOffsets.rend();
      It != End;
       ++It) {
    if (It->count(Name))
      return It->find(Name)->second / alignment;
  }
  assert(0 && "local val name not found");
  return 0;
}

void FuncContext::rememberContext() {
  this->LocalMemSizeRemembered = this->LocalMemSize;
  this->LocalOffsetsRemembered = this->LocalOffsets;
}

void FuncContext::restoreRememberedContext() {
  this->LocalMemSize = this->LocalMemSizeRemembered;
  this->LocalOffsets = this->LocalOffsetsRemembered;
}

static void addGCRootDecl(llvm::Module *M) {
  auto FunRet = llvm::Type::getVoidTy(llvm::getGlobalContext());
  llvm::SmallVector<llvm::Type *, 2> Arg;
  Arg.push_back(llvm::PointerType::getUnqual(llvm::PointerType::getUnqual(
      llvm::IntegerType::get(M->getContext(), 8))));
  Arg.push_back(
      llvm::PointerType::getUnqual(llvm::IntegerType::get(M->getContext(),
          8)));
  auto FunType = llvm::FunctionType::get(FunRet, Arg, false);
  auto F = llvm::Function::Create(FunType,
      llvm::GlobalValue::ExternalLinkage,
                                  "llvm.gcroot", M);
  F->setCallingConv(llvm::CallingConv::C);
  setGC(F);
}

static void addPrintDecl(llvm::Module *M) {
  auto FunRet = getInteger64Pointer(M);
  llvm::SmallVector<llvm::Type *, 1> Arg;
  Arg.push_back(getInteger64(M));
  auto FunType = llvm::FunctionType::get(FunRet, Arg, false);
  auto F = llvm::Function::Create(FunType,
      llvm::GlobalValue::ExternalLinkage,
                                  "print", M);
  F->setCallingConv(llvm::CallingConv::C);
  setGC(F);
}
```

```
#if 0
static void addPatternCheckDecl(llvm::Module *M) {
  auto FunRet = llvm::Type::getVoidTy(llvm::getGlobalContext());
  llvm::SmallVector<llvm::Type *, 1> Arg;
  Arg.push_back(getInteger64(M));
  auto FunType = llvm::FunctionType::get(FunRet, Arg, false);
  auto F = llvm::Function::Create(FunType,
      llvm::GlobalValue::ExternalLinkage,
                                  "patternCheck", M);
  F->setCallingConv(llvm::CallingConv::C);
}

static void callPatternCheck(llvm::Module *M, llvm::BasicBlock *B,
                             llvm::Value *A) {
  llvm::SmallVector<llvm::Value *, 1> Arg;
  Arg.push_back(A);
  auto F = M->getFunction("patternCheck");
  assert(F);
  llvm::CallInst::Create(F, Arg, "", B);
}
#endif

static void addFirstGenBeginDecl(llvm::Module *M) {
#ifdef SSML_GENERATIONAL_GC
  auto Type = getInteger64(M);
  new llvm::GlobalVariable(*M, Type, true,
      llvm::GlobalValue::ExternalLinkage,
                           nullptr, "FirstGenBegin", nullptr,
                           llvm::GlobalValue::NotThreadLocal, 0,
                               true);
#endif
}

static void addFirstGenEndDecl(llvm::Module *M) {
#ifdef SSML_GENERATIONAL_GC
  auto Type = getInteger64(M);
  new llvm::GlobalVariable(*M, Type, true,
      llvm::GlobalValue::ExternalLinkage,
                           nullptr, "FirstGenEnd", nullptr,
                           llvm::GlobalValue::NotThreadLocal, 0,
                               true);
#endif
}

static void addMemoryBarrierDecl(llvm::Module *M) {
#ifdef SSML_GENERATIONAL_GC
  auto FunRet = llvm::Type::getVoidTy(llvm::getGlobalContext());
  llvm::SmallVector<llvm::Type *, 1> Arg;
  Arg.push_back(getInteger64Pointer(M));
  // Arg.push_back(getInteger64(M));
  auto FunType = llvm::FunctionType::get(FunRet, Arg, false);
  auto F = llvm::Function::Create(FunType,
      llvm::GlobalValue::ExternalLinkage,
                                  "rememberset_insert", M);
  F->setCallingConv(llvm::CallingConv::C);
#endif
}

static void addMemoryBarrierDef(llvm::Module *M) {
#ifdef SSML_GENERATIONAL_GC
  auto FunRet = llvm::Type::getVoidTy(llvm::getGlobalContext());
```

```
  llvm::SmallVector<llvm::Type *, 2> Arg;
  Arg.push_back(getInteger64Pointer(M));
  Arg.push_back(getInteger64(M));
  auto FunType = llvm::FunctionType::get(FunRet, Arg, false);
  auto F = llvm::Function::Create(FunType,
      llvm::GlobalValue::InternalLinkage,
                                   "ssmlMemorybarrier", M);
  F->setCallingConv(llvm::CallingConv::C);

  auto Entry = llvm::BasicBlock::Create(M->getContext(),
      "entrylabel", F, 0);

  auto ArgIt = F->arg_begin();
  llvm::Value *Base = ArgIt;
  llvm::Value *Source = ++ArgIt;

  auto TB1 = llvm::BasicBlock::Create(llvm::getGlobalContext());
  auto TB2 = llvm::BasicBlock::Create(llvm::getGlobalContext());
  auto TB3 = llvm::BasicBlock::Create(llvm::getGlobalContext());
  auto FB = llvm::BasicBlock::Create(llvm::getGlobalContext());
  auto And = llvm::BinaryOperator::CreateAnd(Source, getConst(M,
      1), "", Entry);
  auto Cond =
      new llvm::ICmpInst(*Entry, llvm::CmpInst::ICMP_NE, And,
          getConst(M, 1));
  llvm::BranchInst::Create(TB1, FB, Cond, Entry);
  TB1->insertInto(F);

  auto BaseInt = new llvm::PtrToIntInst(Base, getInteger64(M), "",
      TB1);

  llvm::Value *Fen = M->getGlobalVariable("FirstGenEnd");
  assert(Fen);
  Fen = new llvm::LoadInst(Fen, "", TB1);

  Cond = new llvm::ICmpInst(*TB1, llvm::CmpInst::ICMP_UGE,
      BaseInt, Fen);
  llvm::BranchInst::Create(TB3, TB2, Cond, TB1);
  TB2->insertInto(F);

  llvm::Value *Fbe = M->getGlobalVariable("FirstGenBegin");
  assert(Fbe);
  Fbe = new llvm::LoadInst(Fbe, "", TB2);

  Cond = new llvm::ICmpInst(*TB2, llvm::CmpInst::ICMP_ULT,
      BaseInt, Fbe);
  llvm::BranchInst::Create(TB3, FB, Cond, TB2);
  TB3->insertInto(F);

  llvm::SmallVector<llvm::Value *, 1> Args;
  Args.push_back(Base);

  auto Barrier = M->getFunction("rememberset_insert");
  auto Call = llvm::CallInst::Create(Barrier, Args, "", TB3);
  Call->setTailCall();
  llvm::BranchInst::Create(FB, TB3);

  FB->insertInto(F);
  llvm::ReturnInst::Create(M->getContext(), nullptr, FB);
#endif
}
```

```
static void addGCWriteDecl(llvm::Module *M) {
  auto FunRet = llvm::Type::getVoidTy(llvm::getGlobalContext());
  llvm::SmallVector<llvm::Type *, 3> Arg;
  Arg.push_back(
      llvm::PointerType::getUnqual(llvm::IntegerType::get(M->getContext(),
          8)));
  Arg.push_back(
      llvm::PointerType::getUnqual(llvm::IntegerType::get(M->getContext(),
          8)));
  Arg.push_back(llvm::PointerType::getUnqual(llvm::PointerType::getUnqual(
      llvm::IntegerType::get(M->getContext(), 8))));
  auto FunType = llvm::FunctionType::get(FunRet, Arg, false);
  auto F = llvm::Function::Create(FunType,
      llvm::GlobalValue::ExternalLinkage,
                                    "llvm.gcwrite", M);
  F->setCallingConv(llvm::CallingConv::C);
  setGC(F);
}

static void addGCReadDecl(llvm::Module *M) {
  auto FunRet =
      llvm::PointerType::getUnqual(llvm::IntegerType::get(M->getContext(),
          8));
  llvm::SmallVector<llvm::Type *, 2> Arg;
  Arg.push_back(
      llvm::PointerType::getUnqual(llvm::IntegerType::get(M->getContext(),
          8)));
  Arg.push_back(llvm::PointerType::getUnqual(llvm::PointerType::getUnqual(
      llvm::IntegerType::get(M->getContext(), 8))));
  auto FunType = llvm::FunctionType::get(FunRet, Arg, false);
  auto F = llvm::Function::Create(FunType,
      llvm::GlobalValue::ExternalLinkage,
                                    "llvm.gcread", M);
  F->setCallingConv(llvm::CallingConv::C);
  setGC(F);
}

static void addAllocateDecl(llvm::Module *M) {
  auto FunRet = getInteger64Pointer(M);
  llvm::SmallVector<llvm::Type *, 1> Arg;
  Arg.push_back(getInteger64(M));
  auto FunType = llvm::FunctionType::get(FunRet, Arg, false);
  auto F = llvm::Function::Create(FunType,
      llvm::GlobalValue::ExternalLinkage,
                                    "allocate", M);
  F->setCallingConv(llvm::CallingConv::C);
  setGC(F);
}

static void addDoarrayDecl(llvm::Module *M) {
  auto FunRet = getInteger64(M);
  llvm::SmallVector<llvm::Type *, 2> Args;
  Args.push_back(getInteger64(M));
  Args.push_back(getInteger64(M));
  auto FunType = llvm::FunctionType::get(FunRet, Args, false);
  auto F = llvm::Function::Create(FunType,
      llvm::GlobalValue::ExternalLinkage,
                                    "doarray", M);
  F->setCallingConv(llvm::CallingConv::C);
  setGC(F);
}

```

```
static void addDoarraygetDecl(llvm::Module *M) {
  auto FunRet = getInteger64(M);
  llvm::SmallVector<llvm::Type *, 2> Args;
  Args.push_back(getInteger64(M));
  Args.push_back(getInteger64(M));
  auto FunType = llvm::FunctionType::get(FunRet, Args, false);
  auto F = llvm::Function::Create(FunType,
      llvm::GlobalValue::ExternalLinkage,
                                  "doarrayget", M);
  F->setCallingConv(llvm::CallingConv::C);
  setGC(F);
}

static void addDoarrayupdateDecl(llvm::Module *M) {
  auto FunRet = getInteger64(M);
  llvm::SmallVector<llvm::Type *, 3> Args;
  Args.push_back(getInteger64(M));
  Args.push_back(getInteger64(M));
  Args.push_back(getInteger64(M));
  auto FunType = llvm::FunctionType::get(FunRet, Args, false);
  auto F = llvm::Function::Create(FunType,
      llvm::GlobalValue::ExternalLinkage,
                                  "doarrayupdate", M);
  F->setCallingConv(llvm::CallingConv::C);
  setGC(F);
}

static void addArrayemptyDecl(llvm::Module *M) {
  auto Type = getInteger64(M);
  new llvm::GlobalVariable(*M, Type, true,
      llvm::GlobalValue::ExternalLinkage,
                           nullptr, "arrayempty", nullptr,
                           llvm::GlobalValue::NotThreadLocal, 0,
                               true);
}

static void addFrameArrayupdateDecl(llvm::Module *M) {
  auto Type = getInteger64Pointer(M);
  new llvm::GlobalVariable(*M, Type, true,
      llvm::GlobalValue::ExternalLinkage,
                           nullptr, "frameArrayupdate", nullptr,
                           llvm::GlobalValue::NotThreadLocal, 0,
                               true);
}

static void addFramePrintDecl(llvm::Module *M) {
  auto Type = getInteger64Pointer(M);
  new llvm::GlobalVariable(*M, Type, true,
      llvm::GlobalValue::ExternalLinkage,
                           nullptr, "framePrintVal", nullptr,
                           llvm::GlobalValue::NotThreadLocal, 0,
                               true);
}

static void addFramePlusDecl(llvm::Module *M) {
  auto Type = getInteger64Pointer(M);
  new llvm::GlobalVariable(*M, Type, true,
      llvm::GlobalValue::ExternalLinkage,
                           nullptr, "framePlus", nullptr,
                           llvm::GlobalValue::NotThreadLocal, 0,
                               true);
}
```

```

static void addFrameMinusDecl(llvm::Module *M) {
  auto Type = getInteger64Pointer(M);
  new llvm::GlobalVariable(*M, Type, true,
      llvm::GlobalValue::ExternalLinkage,
                           nullptr, "frameMinus", nullptr,
                           llvm::GlobalValue::NotThreadLocal, 0,
                               true);
}

static void addFrameMultiplyDecl(llvm::Module *M) {
  auto Type = getInteger64Pointer(M);
  new llvm::GlobalVariable(*M, Type, true,
      llvm::GlobalValue::ExternalLinkage,
                           nullptr, "frameMultiply", nullptr,
                           llvm::GlobalValue::NotThreadLocal, 0,
                               true);
}

static void addFrameDivisionDecl(llvm::Module *M) {
  auto Type = getInteger64Pointer(M);
  new llvm::GlobalVariable(*M, Type, true,
      llvm::GlobalValue::ExternalLinkage,
                           nullptr, "frameDivision", nullptr,
                           llvm::GlobalValue::NotThreadLocal, 0,
                               true);
}

static void addFrameModuloDecl(llvm::Module *M) {
  auto Type = getInteger64Pointer(M);
  new llvm::GlobalVariable(*M, Type, true,
      llvm::GlobalValue::ExternalLinkage,
                           nullptr, "frameModulo", nullptr,
                           llvm::GlobalValue::NotThreadLocal, 0,
                               true);
}

static void addFrameRefassignDecl(llvm::Module *M) {
  auto Type = getInteger64Pointer(M);
  new llvm::GlobalVariable(*M, Type, true,
      llvm::GlobalValue::ExternalLinkage,
                           nullptr, "frameRefassign", nullptr,
                           llvm::GlobalValue::NotThreadLocal, 0,
                               true);
}

static void addFrameArrayDecl(llvm::Module *M) {
  auto Type = getInteger64Pointer(M);
  new llvm::GlobalVariable(*M, Type, true,
      llvm::GlobalValue::ExternalLinkage,
                           nullptr, "frameArray", nullptr,
                           llvm::GlobalValue::NotThreadLocal, 0,
                               true);
}

static void addFrameArraygetDecl(llvm::Module *M) {
  auto Type = getInteger64Pointer(M);
  new llvm::GlobalVariable(*M, Type, true,
      llvm::GlobalValue::ExternalLinkage,
                           nullptr, "frameArrayget", nullptr,
                           llvm::GlobalValue::NotThreadLocal, 0,
                               true);
```

```
}

static void addFrameNegateDecl(llvm::Module *M) {
  auto Type = getInteger64Pointer(M);
  new llvm::GlobalVariable(*M, Type, true,
      llvm::GlobalValue::ExternalLinkage,
                           nullptr, "frameNegate", nullptr,
                           llvm::GlobalValue::NotThreadLocal, 0,
                               true);
}

static void addFrameNotDecl(llvm::Module *M) {
  auto Type = getInteger64Pointer(M);
  new llvm::GlobalVariable(*M, Type, true,
      llvm::GlobalValue::ExternalLinkage,
                           nullptr, "frameNot", nullptr,
                           llvm::GlobalValue::NotThreadLocal, 0,
                               true);
}

static void addFrameRefDecl(llvm::Module *M) {
  auto Type = getInteger64Pointer(M);
  new llvm::GlobalVariable(*M, Type, true,
      llvm::GlobalValue::ExternalLinkage,
                           nullptr, "frameRef", nullptr,
                           llvm::GlobalValue::NotThreadLocal, 0,
                               true);
}

static void addFrameDerefDecl(llvm::Module *M) {
  auto Type = getInteger64Pointer(M);
  new llvm::GlobalVariable(*M, Type, true,
      llvm::GlobalValue::ExternalLinkage,
                           nullptr, "frameDeref", nullptr,
                           llvm::GlobalValue::NotThreadLocal, 0,
                               true);
}

static void addFrameArraylengthDecl(llvm::Module *M) {
  auto Type = getInteger64Pointer(M);
  new llvm::GlobalVariable(*M, Type, true,
      llvm::GlobalValue::ExternalLinkage,
                           nullptr, "frameArraylength", nullptr,
                           llvm::GlobalValue::NotThreadLocal, 0,
                               true);
}

static void addFrameEqualsDecl(llvm::Module *M) {
  auto Type = getInteger64Pointer(M);
  new llvm::GlobalVariable(*M, Type, true,
      llvm::GlobalValue::ExternalLinkage,
                           nullptr, "frameEquals", nullptr,
                           llvm::GlobalValue::NotThreadLocal, 0,
                               true);
}

static void addFrameNotEqualsDecl(llvm::Module *M) {
  auto Type = getInteger64Pointer(M);
  new llvm::GlobalVariable(*M, Type, true,
      llvm::GlobalValue::ExternalLinkage,
                           nullptr, "frameNotEquals", nullptr,
                           llvm::GlobalValue::NotThreadLocal, 0,
```

```
                                true);
}

static void addFrameLessDecl(llvm::Module *M) {
  auto Type = getInteger64Pointer(M);
  new llvm::GlobalVariable(*M, Type, true,
      llvm::GlobalValue::ExternalLinkage,
                           nullptr, "frameLess", nullptr,
                           llvm::GlobalValue::NotThreadLocal, 0,
                               true);
}

static void addFrameGreaterDecl(llvm::Module *M) {
  auto Type = getInteger64Pointer(M);
  new llvm::GlobalVariable(*M, Type, true,
      llvm::GlobalValue::ExternalLinkage,
                           nullptr, "frameGreater", nullptr,
                           llvm::GlobalValue::NotThreadLocal, 0,
                               true);
}

static void addFrameLessEqualsDecl(llvm::Module *M) {
  auto Type = getInteger64Pointer(M);
  new llvm::GlobalVariable(*M, Type, true,
      llvm::GlobalValue::ExternalLinkage,
                           nullptr, "frameLessEquals", nullptr,
                           llvm::GlobalValue::NotThreadLocal, 0,
                               true);
}

static void addFrameGreaterEqualsDecl(llvm::Module *M) {
  auto Type = getInteger64Pointer(M);
  new llvm::GlobalVariable(*M, Type, true,
      llvm::GlobalValue::ExternalLinkage,
                           nullptr, "frameGreaterEquals", nullptr,
                           llvm::GlobalValue::NotThreadLocal, 0,
                               true);
}

static void addMatchErrorDecl(llvm::Module *M) {
  auto FunRet = llvm::Type::getVoidTy(llvm::getGlobalContext());
  llvm::SmallVector<llvm::Type *, 0> Arg;
  auto FunType = llvm::FunctionType::get(FunRet, Arg, false);
  auto F = llvm::Function::Create(FunType,
      llvm::GlobalValue::ExternalLinkage,
                                  "matchError", M);
  F->setCallingConv(llvm::CallingConv::C);
  llvm::AttrBuilder B;
  B.addAttribute(llvm::Attribute::NoReturn);
  F->setAttributes(llvm::AttributeSet::get(
      llvm::getGlobalContext(), llvm::AttributeSet::FunctionIndex,
          B));
}

static llvm::BasicBlock *
getMatchErrorBasicBlock(llvm::Module *M, const char *matchstr =
    "match_error") {
  auto Ret = llvm::BasicBlock::Create(llvm::getGlobalContext(),
      matchstr);
  auto ErrorFunc = M->getFunction("matchError");
  assert(ErrorFunc);
  llvm::SmallVector<llvm::Value *, 0> Arg;
```

```
  auto Call = llvm::CallInst::Create(ErrorFunc, Arg, "", Ret);
  Call->setTailCall();
  llvm::ReturnInst::Create(M->getContext(), getZeroConst(M), Ret);
  return Ret;
}

static llvm::LoadInst *addVolatileLoad(llvm::Value *Ptr,
                                       const llvm::Twine &NameStr,
                                       llvm::BasicBlock *B) {
#ifdef SSML_VOLATILE_MEMORY
  return new llvm::LoadInst(Ptr, NameStr, true, B);
#else
  return new llvm::LoadInst(Ptr, NameStr, false, B);
#endif
}

static llvm::StoreInst *addVolatileStore(llvm::Value *Src, llvm::Value *Dst,
                                         llvm::BasicBlock *B) {
#ifdef SSML_VOLATILE_MEMORY
  return new llvm::StoreInst(Src, Dst, true, B);
#else
  return new llvm::StoreInst(Src, Dst, false, B);
#endif
}

static llvm::Module *makeModule() {
  auto Ret = new llvm::Module("llvm-ir.ll", llvm::getGlobalContext());
  Ret->setDataLayout("e-m:e-i64:64-f80:128-n8:16:32:64-S128");
  Ret->setTargetTriple("x86_64-unknown-linux-gnu");

  // Add runtime forward declarations.
  addGCRootDecl(Ret);
  addGCWriteDecl(Ret);
  addGCReadDecl(Ret);

  addAllocateDecl(Ret);
  addPrintDecl(Ret);
  addDoarrayDecl(Ret);
  addDoarraygetDecl(Ret);
  addDoarrayupdateDecl(Ret);
  addArrayemptyDecl(Ret);
  addMatchErrorDecl(Ret);

  addFrameArrayupdateDecl(Ret);

  // addPatternCheckDecl(Ret);
  addFirstGenBeginDecl(Ret);
  addFirstGenEndDecl(Ret);
  addMemoryBarrierDecl(Ret);
  addMemoryBarrierDef(Ret);

  addFramePrintDecl(Ret);
  addFramePlusDecl(Ret);
  addFrameMinusDecl(Ret);
  addFrameMultiplyDecl(Ret);
  addFrameDivisionDecl(Ret);
  addFrameModuloDecl(Ret);
  addFrameRefassignDecl(Ret);
  addFrameArrayDecl(Ret);
  addFrameArraygetDecl(Ret);
```

```
  addFrameNegateDecl(Ret);
  addFrameNotDecl(Ret);
  addFrameRefDecl(Ret);
  addFrameDerefDecl(Ret);
  addFrameArraylengthDecl(Ret);

  addFrameEqualsDecl(Ret);
  addFrameNotEqualsDecl(Ret);
  addFrameLessDecl(Ret);
  addFrameGreaterDecl(Ret);
  addFrameLessEqualsDecl(Ret);
  addFrameGreaterEqualsDecl(Ret);

  return Ret;
}

CodegenVisitor::CodegenVisitor() : Module(makeModule()) {}

CodegenVisitor::~CodegenVisitor() { delete this->mod(); }

void CodegenVisitor::doVisit(std::shared_ptr<ssml::ast::Node> N) {
  N->accept(this);
}

void CodegenVisitor::enterScope() {
  this->Names.enterScope();
  this->Instances.enterScope();
  this->peekFunc()->enterScope();
}

void CodegenVisitor::leaveScope(bool ZeroUnused) {
  if (ZeroUnused) {
    auto &Loc = this->peekFunc()->getScopeLocalIndices();

#if 0
    auto FramePtr = this->peekFunc()->getLocalPtr();
    auto Frame = new llvm::LoadInst(FramePtr, "",
        this->BasicBlock);
#else
    auto Frame = this->loadLocalPtr();
#endif
    for (auto I : Loc) {
      llvm::SmallVector<llvm::Value *, 1> Indices;
      Indices.push_back(getConst(this->mod(), I));
      auto Next = llvm::GetElementPtrInst::Create(
          getInteger64(this->mod()), Frame, Indices, "",
              this->BasicBlock);
#if 0
      auto S = new llvm::StoreInst(getZeroConst(this->mod()),
          Next, false,
                                          this->BasicBlock);
      S->setAlignment(alignment);
#else
      this->addGCStore(getZeroConst(this->mod()), Next, Frame);
#endif
    }

    auto Temps = this->peekFunc()->getScopeTemps();
    for (auto T : Temps) {
      auto S = addVolatileStore(getZeroConst(this->mod()), T,
          this->BasicBlock);
```

```
      S->setAlignment(alignment);
    }
  }

  this->Names.leaveScope();
  this->Instances.leaveScope();
  this->peekFunc()->leaveScope();
}

void CodegenVisitor::nextTemp() {
  auto F = this->peekFunc();

  auto TAlloc = new
      llvm::AllocaInst(getInteger64Pointer(this->mod()),
                                     "tempptr",
                                         F->getPostTempsInst());
  TAlloc->setAlignment(alignment);
#if 0
  auto Zstore = new llvm::StoreInst(
      llvm::ConstantPointerNull::get(getInteger64Pointer(this->mod())),
          TAlloc,
      false, F->getPostTempsInst());
  Zstore->setAlignment(alignment);
#endif

#if !defined(SSML_DISABLE_GC)
  auto GC = this->mod()->getFunction("llvm.gcroot");
  assert(GC);
  llvm::SmallVector<llvm::Value *, 2> Args;
  Args.push_back(new llvm::BitCastInst(
      TAlloc, getPointerTo(getPointerTo(getInteger(this->mod(),
          8))), "",
      F->getPostTempsInst()));
  Args.push_back(
      llvm::ConstantPointerNull::get(getPointerTo(getInteger(this->mod(),
          8))));
  auto Call = llvm::CallInst::Create(GC, Args, "",
      F->getPostTempsInst());
  Call->setTailCall();
#endif

  this->Temp = new llvm::BitCastInst(TAlloc,
      getInteger64Pointer(this->mod()),
                                     "temp",
                                         F->getPostTempsInst());

  if (this->InsideLetDecls)
    this->peekFunc()->appendTemp(this->Temp);
  else
    this->peekFunc()->appendTemp(this->Temp, 1);
}

void CodegenVisitor::storeNextTemp(llvm::Value *V) {
  if (!this->LastExpr) {
    this->nextTemp();
    auto S = addVolatileStore(V, this->Temp, this->BasicBlock);
    S->setAlignment(alignment);
  } else {
    this->Temp = V;
  }
}

```

```
llvm::Value *CodegenVisitor::loadTemp(llvm::Value *Temp) {
  if (!Temp)
    Temp = this->Temp;
  return addVolatileLoad(Temp, "", this->BasicBlock);
}

llvm::CallInst *CodegenVisitor::addAllocateCall(llvm::BasicBlock *B,
                                                 llvm::Value *Size) {
  auto AllocFunc = this->mod()->getFunction("allocate");
  assert(AllocFunc);
  llvm::SmallVector<llvm::Value *, 1> Arg;
  Arg.push_back(Size);
  auto Ret = llvm::CallInst::Create(AllocFunc, Arg, "", B);
  Ret->setTailCall();
  return Ret;
}

llvm::Value *CodegenVisitor::addGCLoad(llvm::Value *Source, llvm::Value *Base) {
  if (Source->getType()->isPointerTy())
    Source = new llvm::PtrToIntInst(Source, getInteger64(this->mod()), "",
                                    this->BasicBlock);

  auto L = addVolatileLoad(
      new llvm::IntToPtrInst(Source, getInteger64Pointer(this->mod()), "",
                             this->BasicBlock),
      "", this->BasicBlock);
  return L;

#if 0
  if (Base->getType()->isPointerTy())
    Base = new llvm::PtrToIntInst(Base, getInteger64(this->mod()), "",
                                  this->BasicBlock);
  auto F = this->mod()->getFunction("llvm.gcread");
  llvm::SmallVector<llvm::Value *, 2> Args;
  Args.push_back(new llvm::IntToPtrInst(
      Base, getPointerTo(getInteger(this->mod(), 8)), "", this->BasicBlock));
  Args.push_back(new llvm::IntToPtrInst(
      Source, getPointerTo(getPointerTo(getInteger(this->mod(), 8))), "",
      this->BasicBlock));
  auto Call = llvm::CallInst::Create(F, Args, "", this->BasicBlock);
  Call->setTailCall();
  return new llvm::PtrToIntInst(Call, getInteger64(this->mod()), "gcload",
                                this->BasicBlock);
#endif
}

void CodegenVisitor::addGCStore(llvm::Value *Source, llvm::Value *Dest,
                                llvm::Value *Base) {
  if (Dest->getType()->isPointerTy())
    Dest = new llvm::PtrToIntInst(Dest, getInteger64(this->mod()), "",
```

```
                                  this->BasicBlock);

#ifdef SSML_GENERATIONAL_GC
  if (!Base->getType()->isPointerTy())
    Base = new llvm::IntToPtrInst(Base,
        getInteger64Pointer(this->mod()), "",
                                  this->BasicBlock);
  llvm::SmallVector<llvm::Value *, 1> Args;
  Args.push_back(Base);
  Args.push_back(Source);
  auto F = this->mod()->getFunction("ssmlMemorybarrier");
  auto Call = llvm::CallInst::Create(F, Args, "",
      this->BasicBlock);
  Call->setTailCall();
#endif

  auto D = new llvm::IntToPtrInst(Dest,
      getInteger64Pointer(this->mod()), "",
                                  this->BasicBlock);
  auto S = addVolatileStore(Source, D, this->BasicBlock);
  S->setAlignment(alignment);
}

void CodegenVisitor::addFuncEntry(FuncContext *F, bool IsMain,
                                  bool PrevFrameIsLocals) {
  auto FrameSize = F->getMaxLocalMemSize();
  auto &EntryBB = F->func()->getEntryBlock();

  auto Alloc = new
      llvm::AllocaInst(getInteger64Pointer(this->mod()), "locals",
                                    &EntryBB);
  Alloc->setAlignment(alignment);
  F->setLocalPtr(Alloc);

  if (!IsMain) {
#ifdef SSML_GENERATIONAL_GC
    this->nextTemp();
#endif
    auto ArgIt = F->func()->arg_begin();
    this->storeNextTemp(ArgIt);
    F->setFirstArg(this->getExprResult());
    this->storeNextTemp(++ArgIt);
    F->setSecondArg(this->getExprResult());
  } else {
#ifdef SSML_GENERATIONAL_GC
    // Mark main as last stack frame to search for roots.
    this->storeNextTemp(getConst(this->mod(), 1));
#endif
  }

#if !defined(SSML_DISABLE_GC)
  auto GC = this->mod()->getFunction("llvm.gcroot");
  assert(GC);
  llvm::SmallVector<llvm::Value *, 2> Args;

  Args.push_back(new llvm::BitCastInst(
      Alloc, getPointerTo(getPointerTo(getInteger(this->mod(),
          8))), "",
      this->BasicBlock));
  Args.push_back(
      llvm::ConstantPointerNull::get(getPointerTo(getInteger(this->mod(),
          8))));
```

```
  auto Call = llvm::CallInst::Create(GC, Args, "", this->BasicBlock);
  Call->setTailCall();
#endif

  llvm::Value *LocalPtr;
  if (!PrevFrameIsLocals) {
    LocalPtr =
        this->addAllocateCall(&EntryBB, getConst(this->mod(), FrameSize));
    auto Store = addVolatileStore(LocalPtr, Alloc, &EntryBB);
    Store->setAlignment(alignment);
  } else {
    llvm::Value *PrevFrame = this->loadTemp(F->getFirstArg());
    LocalPtr = PrevFrame = new llvm::IntToPtrInst(
        PrevFrame, getInteger64Pointer(this->mod()), "", &EntryBB);
    auto Store = addVolatileStore(PrevFrame, Alloc, &EntryBB);
    Store->setAlignment(alignment);
  }

#if 0
  {
    llvm::SmallVector<llvm::Value *, 1> Arg;
    Arg.push_back(new llvm::PtrToIntInst(Temps, getInteger64(this->mod()), "",
                                         this->BasicBlock));
    auto Fun = this->mod()->getFunction("print");
    assert(Fun);
    llvm::CallInst::Create(Fun, Arg, "", this->BasicBlock);
  }
  {
    llvm::SmallVector<llvm::Value *, 1> Arg;
    Arg.push_back(new llvm::PtrToIntInst(LocalPtr, getInteger64(this->mod()),
                                         "", this->BasicBlock));
    auto Fun = this->mod()->getFunction("print");
    assert(Fun);
    llvm::CallInst::Create(Fun, Arg, "", this->BasicBlock);
  }
#endif
}

void CodegenVisitor::pushFunc(llvm::Function *F,
                              std::shared_ptr<FunctionVals> Vs,
                              const std::string &Name,
                              llvm::Function *CurrFunc) {
  auto Next = new FuncContext(F, Vs, Name, CurrFunc);
  Next->setPrev(this->FunContext);
  this->FunContext = Next;
  Next->setInsideLetDecl(this->InsideLetDecls);
  this->InsideLetDecls = false;
}

void CodegenVisitor::changeFunc(llvm::Function *F) {
  this->FunContext->func(F);
}

FuncContext *CodegenVisitor::peekFunc() { return this->FunContext; }

void CodegenVisitor::popFunc() {
  this->InsideLetDecls = this->peekFunc()->getInsideLetDecl();
```

```
  auto Prev = this->peekFunc()->getPrev();
  delete this->peekFunc();
  this->FunContext = Prev;
}

void CodegenVisitor::beginMain(std::shared_ptr<FunctionVals> Vs) {
  llvm::Module *M = this->mod();
  auto FunRet = getInteger64(M);
  auto T = llvm::FunctionType::get(FunRet, false);
  auto F =
      llvm::Function::Create(T,
          llvm::GlobalValue::ExternalLinkage, "entry", M);
  F->setCallingConv(llvm::CallingConv::C);
  setGC(F);
  this->BasicBlock =
      llvm::BasicBlock::Create(M->getContext(), "entrylabel", F,
          0);
  this->pushFunc(F, Vs);
  this->addFuncEntry(this->peekFunc(), true, false);
}

void CodegenVisitor::endMain() {
  auto M = this->mod();
  auto Main = this->peekFunc()->func();
  assert(!!(Main->getName() == "entry"));
  (void)Main;
  auto Zero = getZeroConst(M);
  llvm::ReturnInst::Create(M->getContext(), Zero,
      this->BasicBlock);
  this->popFunc();
}

llvm::Value *CodegenVisitor::addLocalLookup(llvm::BasicBlock *B,
                                            NameMap::ScopeValue
                                                SV) {
  assert(!SV.isTypeInstance());
  auto CurrScope = this->peekFunc()->getNest();
  auto LookupCount = CurrScope - SV.Scope;
#if 0
  llvm::Value *Frame = this->peekFunc()->getLocalPtr();
  Frame = new llvm::LoadInst(Frame, "", B);
#endif
  auto Frame = this->loadLocalPtr();

  auto CurrFunc = this->peekFunc();
  for (int32_t I = 0; I < LookupCount; ++I) {
    llvm::SmallVector<llvm::Value *, 1> Indices1;
    Indices1.push_back(getConst(this->mod(),
        CurrFunc->getFramePtrIndex()));
    llvm::Value *Tmp = llvm::GetElementPtrInst::Create(
        getInteger64(this->mod()), Frame, Indices1, "", B);
#if 0
    Tmp = new llvm::LoadInst(Tmp, "", B);
#else
    Tmp = this->addGCLoad(Tmp, Frame);
#endif
    Frame =
        new llvm::IntToPtrInst(Tmp,
            getInteger64Pointer(this->mod()), "", B);
    CurrFunc = CurrFunc->getPrev();
  }
  // Should store Frame in temp storage, but a GC cannot be
```

```
      triggered here.
  llvm::SmallVector<llvm::Value *, 1> Indices;
  Indices.push_back(getConst(this->mod(), SV.Index));
  auto Ptr =
      llvm::GetElementPtrInst::Create(getInteger64(this->mod()),
      Frame,
                                                      Indices, "", B);
#if 0
  return new llvm::LoadInst(Ptr, "", B);
#else
  return this->addGCLoad(Ptr, Frame);
#endif
}

void CodegenVisitor::visit(IntLiteral *N) {
  fatalExit("CodegenVisitor visit IntLiteral");
}

void CodegenVisitor::visit(ShortIdentifier *N) {
  fatalExit("CodegenVisitor visit ShortIdentifier");
}

void CodegenVisitor::visit(LongIdentifier *N) {
  fatalExit("CodegenVisitor visit LongIdentifier");
}

void CodegenVisitor::visit(SeqShortIdentifier *N) {
  fatalExit("CodegenVisitor visit SeqShortIdentifier");
}

void CodegenVisitor::visit(SeqLongIdentifier *N) {
  fatalExit("CodegenVisitor visit SeqLongIdentifier");
}

void CodegenVisitor::visit(SeqExpression *N) {
  bool IsLast = this->LastExpr;
  this->LastExpr = false;

  auto TB = this->TrueBlock;
  auto FB = this->FalseBlock;
  auto Size = N->size();
  for (size_t I = 0; I < Size; ++I) {
    if (I < Size - 1) {
      this->TrueBlock = nullptr;
      this->FalseBlock = nullptr;
    } else {
      this->LastExpr = IsLast;
      this->TrueBlock = TB;
      this->FalseBlock = FB;
    }
    this->doVisit((*N)[I]);
  }
  this->ExprWasTypeInstance = false;
}

void CodegenVisitor::visit(LiteralExpression *N) {
  auto Val = N->getValue()->toInt();
  auto ExprResultTmp = getConst(this->mod(), markValue(Val));
  this->storeNextTemp(ExprResultTmp);
  this->ExprWasTypeInstance = false;
}

```

```
void CodegenVisitor::visit(LongIdentifierExpression *N) {
  NameMap::ScopeValue SV =
      this->Names.get(N->getIDs()->toString());
  if (SV.isTypeInstance()) {
    auto P = this->Instances.get(N->getIDs()->toString());
    llvm::Value *ExprResultTmp;
    if (P.second)
      ExprResultTmp = this->addAllocateCall(
          this->BasicBlock, getConst(this->mod(), alignment * 2));
    else
      ExprResultTmp = this->addAllocateCall(this->BasicBlock,
                                            getConst(this->mod(),
                                                alignment));
    ExprResultTmp = new llvm::PtrToIntInst(
        ExprResultTmp, getInteger64(this->mod()), "",
            this->BasicBlock);
    this->storeNextTemp(ExprResultTmp);
#if 0
    auto S = new llvm::StoreInst(getConst(this->mod(),
        markValue(P.first)),
                                 ExprResultTmp, false,
                                     this->BasicBlock);
    S->setAlignment(alignment);
#else
    this->addGCStore(getConst(this->mod(), markValue(P.first)),
        ExprResultTmp,
                     ExprResultTmp);
#endif
    this->ExprWasTypeInstance = true;
  } else if (SV.isExisting()) {
    auto ExprResultTmp = this->addLocalLookup(this->BasicBlock,
        SV);
    this->storeNextTemp(ExprResultTmp);
    this->ExprWasTypeInstance = false;
  } else {
    auto ID = N->getIDs()->toString();
    llvm::Value *Val = getSimpleBuiltinConst(this->mod(), ID,
        true);
    if (Val) {
      this->storeNextTemp(Val);
      goto out;
    }
    Val = getBuiltin(this->mod(), ID);
    if (Val) {
      auto ExprResultTmp = new llvm::PtrToIntInst(
          Val, getInteger64(this->mod()), "", this->BasicBlock);
      this->storeNextTemp(ExprResultTmp);
      goto out;
    }

    NameMap::ScopeValue ScopeVal = {this->peekFunc()->getNest() +
        1, -1};
    auto Context = this->peekFunc();
    llvm::Value *CurrFunc = Context->currFunc();
    do {
      if (Context->currFuncName() == ID) {
        if (Context->getPrev() &&
            Context->getPrev()->getIsStepFunction()) {
          --ScopeVal.Scope;
          ScopeVal.Index = Context->getFramePtrIndex();
          Context = Context->getPrev();
        }
```

```
        break;
      }
      --ScopeVal.Scope;
      ScopeVal.Index = Context->getFramePtrIndex();
      Context = Context->getPrev();
      CurrFunc = Context->currFunc();
    } while (Context);

    if (Context) {
      auto Result = this->addAllocateCall(this->BasicBlock,
                                          getConst(this->mod(), 2
                                              * alignment));
      auto ExprResultTmp = new llvm::PtrToIntInst(
          Result, getInteger64(this->mod()), "", this->BasicBlock);
      this->storeNextTemp(ExprResultTmp);

      llvm::Value *Frame;
      if (ScopeVal.Index != -1) {
        Frame = this->addLocalLookup(this->BasicBlock, ScopeVal);
        Frame = new llvm::IntToPtrInst(Frame,
            getInteger64Pointer(this->mod()),
                                      "", this->BasicBlock);
      } else {
#if 0
        Frame = new
            llvm::LoadInst(this->peekFunc()->getLocalPtr(), "",
                                   this->BasicBlock);
#else
        Frame = this->loadLocalPtr();
#endif
      }

      llvm::SmallVector<llvm::Value *, 1> Indices;
      Indices.push_back(getConst(this->mod(),
          Context->getFramePtrIndex()));
      auto PrevFrameOffset = llvm::GetElementPtrInst::Create(
          getInteger64(this->mod()), Frame, Indices, "",
              this->BasicBlock);
// Should put frame in temp storage, but GC cannot ge triggered
    here.
#if 0
      auto PrevFrameInt =
          new llvm::LoadInst(PrevFrameOffset, "",
              this->BasicBlock);
#else
      auto PrevFrameInt = this->addGCLoad(PrevFrameOffset, Frame);
#endif
      auto Fun = new llvm::PtrToIntInst(CurrFunc,
          getInteger64(this->mod()), "",
                                       this->BasicBlock);
#if 0
      auto Store = new llvm::StoreInst(Fun, Result, false,
          this->BasicBlock);
      Store->setAlignment(alignment);
#else
      this->addGCStore(Fun, Result, Result);
#endif
      Indices.clear();
      Indices.push_back(getConst(this->mod(), 1));
      auto Snd = llvm::GetElementPtrInst::Create(
          getInteger64(this->mod()), Result, Indices, "",
              this->BasicBlock);
```

```
#if 0
      Store = new llvm::StoreInst(PrevFrameInt, Snd, false,
          this->BasicBlock);
      Store->setAlignment(alignment);
#else
      this->addGCStore(PrevFrameInt, Snd, Result);
#endif
      goto out;
    }
    fatalExit("unexpected LongIdentifierExpression: " + ID);
  }
out:;
}

void CodegenVisitor::visit(TupleExpression *N) {
  auto IsLast = this->LastExpr;
  this->LastExpr = false;

  auto FB = this->FalseBlock;
  auto TB = this->TrueBlock;

  auto Seq = N->getExprs();

  llvm::Value *ExprResultTmp;
  llvm::Value *Ptr = nullptr;
  if (Seq->size()) {
    Ptr = addAllocateCall(this->BasicBlock,
                          getConst(this->mod(), Seq->size() *
                              alignment));
    ExprResultTmp = new llvm::PtrToIntInst(Ptr,
        getInteger64(this->mod()), "",
                                          this->BasicBlock);
  } else {
    ExprResultTmp = getZeroConst(this->mod());
  }
  this->storeNextTemp(ExprResultTmp);
  llvm::Value *Result = this->getExprResult();

  size_t Idx = 0;
  for (auto E : *Seq) {
    this->TrueBlock = nullptr;
    this->FalseBlock = nullptr;
    this->doVisit(E);

    llvm::SmallVector<llvm::Value *, 1> Indices;
    Indices.push_back(getConst(this->mod(), Idx));
    llvm::Value *Dst = this->loadTemp(Result);
    Dst = new llvm::IntToPtrInst(Dst,
        getInteger64Pointer(this->mod()), "",
                                this->BasicBlock);
    auto Loc =
        llvm::GetElementPtrInst::Create(getInteger64(this->mod()),
        Dst,
                                        Indices, "",
                                            this->BasicBlock);
    this->addGCStore(this->loadTemp(), Loc, Dst);

    ++Idx;
  }

  this->LastExpr = IsLast;
  if (!IsLast)
```

```
    this->setExprResult(Result);
  else
    this->setExprResult(this->loadTemp(Result));

  this->ExprWasTypeInstance = false;
  this->TrueBlock = TB;
  this->FalseBlock = FB;
}

void CodegenVisitor::visit(ListExpression *N) {
  bool IsLast = this->LastExpr;
  this->LastExpr = false;

  auto FB = this->FalseBlock;
  auto TB = this->TrueBlock;

  llvm::SmallVector<llvm::Value *, 1> Indices;
  Indices.push_back(getConst(this->mod(), 1));

  auto Seq = N->getExprs();
  llvm::Value *Result = nullptr;
  llvm::Value *First = nullptr;
  llvm::Value *Prev = nullptr;
  size_t Idx = 0;
  if (Seq->size()) {
    for (auto E : *Seq) {
      this->TrueBlock = nullptr;
      this->FalseBlock = nullptr;
      this->doVisit(E);
      auto Expr = this->loadTemp();

      auto List = addAllocateCall(this->BasicBlock,
                                  getConst(this->mod(), 2 *
                                      alignment));
      auto ListInt = new llvm::PtrToIntInst(List,
          getInteger64(this->mod()), "",
                                             this->BasicBlock);
      this->storeNextTemp(ListInt);
      if (!First) {
        First = ListInt;
        Result = this->getExprResult();
      }
#if 0
      auto Store = new llvm::StoreInst(getConst(this->mod(),
          markValue(0)),
                                        List, false,
                                            this->BasicBlock);
      Store->setAlignment(alignment);
#else
      this->addGCStore(getConst(this->mod(), markValue(0)),
          ListInt, ListInt);
#endif
      auto Tup = addAllocateCall(this->BasicBlock,
                                 getConst(this->mod(), 2 *
                                     alignment));
      auto TupInt = new llvm::PtrToIntInst(Tup,
          getInteger64(this->mod()), "",
                                            this->BasicBlock);
      this->storeNextTemp(TupInt);
      auto NextPrev = this->getExprResult();

      auto Snd = llvm::GetElementPtrInst::Create(
```

```
          getInteger64(this->mod()), List, Indices, "",
              this->BasicBlock);

#if 0
      Store = new llvm::StoreInst(TupInt, Snd, false,
          this->BasicBlock);
      Store->setAlignment(alignment);
#else
      this->addGCStore(TupInt, Snd, ListInt);
#endif

#if 0
      Store =
          new llvm::StoreInst(this->loadTemp(), Tup, false,
              this->BasicBlock);
      Store->setAlignment(alignment);
#else
      this->addGCStore(Expr, TupInt, TupInt);
#endif

      if (Prev) {
        llvm::Value *Ptr = this->loadTemp(Prev);
        Ptr = new llvm::IntToPtrInst(Ptr,
            getInteger64Pointer(this->mod()), "",
                                         this->BasicBlock);
        Snd =
            llvm::GetElementPtrInst::Create(getInteger64(this->mod()),
            Ptr,
                                                     Indices, "",
                                                         this->BasicBlock);
#if 0
        Store = new llvm::StoreInst(ListInt, Snd, false,
            this->BasicBlock);
        Store->setAlignment(alignment);
#else
        this->addGCStore(ListInt, Snd, Ptr);
#endif
      }

      if (Idx == Seq->size() - 1) {
        auto End =
            addAllocateCall(this->BasicBlock,
                getConst(this->mod(), alignment));
        auto EndInt = new llvm::PtrToIntInst(End,
            getInteger64(this->mod()), "",
                                                    this->BasicBlock);
        this->storeNextTemp(EndInt);
#if 0
        Store = new llvm::StoreInst(getConst(this->mod(),
            markValue(1)), End,
                                        false, this->BasicBlock);
        Store->setAlignment(alignment);
#else
        this->addGCStore(getConst(this->mod(), markValue(1)),
            EndInt, EndInt);
#endif

        Snd =
            llvm::GetElementPtrInst::Create(getInteger64(this->mod()),
            Tup,
                                                     Indices, "",
                                                         this->BasicBlock);
```

```
#if 0
        Store = new llvm::StoreInst(EndInt, Snd, false,
            this->BasicBlock);
        Store->setAlignment(alignment);
#else
        this->addGCStore(EndInt, Snd, TupInt);
#endif
      }

      Prev = NextPrev;
      ++Idx;
    }
  } else {
    auto End =
        addAllocateCall(this->BasicBlock, getConst(this->mod(),
            alignment));
    First = new llvm::PtrToIntInst(End, getInteger64(this->mod()),
        "",
                                   this->BasicBlock);
    this->storeNextTemp(First);
    Result = this->getExprResult();
#if 0
    auto Store = new llvm::StoreInst(getConst(this->mod(),
        markValue(1)), End,
                                     false, this->BasicBlock);
    Store->setAlignment(alignment);
#else
    this->addGCStore(getConst(this->mod(), markValue(1)), First,
        First);
#endif
  }

  if (!IsLast)
    this->setExprResult(Result);
  else
    this->setExprResult(this->loadTemp(Result));

  this->ExprWasTypeInstance = false;
  this->TrueBlock = TB;
  this->FalseBlock = FB;
}

bool CodegenVisitor::tryBuiltinApplyExpr(ApplyExpression
    *ApplyExpr) {

  if (ApplyExpr->size() == 1) {
    auto IDExpr = (*ApplyExpr)[0]->asLongIdentifierExpression();
    if (!IDExpr)
      return false;
    auto ID = IDExpr->getIDs()->toString();
    if (getBuiltinValue(this->mod(), ID)) {
      bool IsLast = this->LastExpr;
      this->LastExpr = false;
      if (ID == "Array.empty") {
        llvm::Value *ExprResultTmp =
            this->mod()->getGlobalVariable("arrayempty");
        ExprResultTmp = new llvm::PtrToIntInst(
            ExprResultTmp, getInteger64(this->mod()), "",
                this->BasicBlock);
        this->LastExpr = IsLast;
        this->storeNextTemp(ExprResultTmp);
      } else {
```

```
        fatalExit("unexpected buintin value: " + ID);
      }
      return true;
    }
    return false;
  }

  if (ApplyExpr->size() != 2)
    return false;

  auto Left = (*ApplyExpr)[0];
  auto Right = (*ApplyExpr)[1];
  auto IDExpr = Left->asLongIdentifierExpression();
  if (!IDExpr)
    return false;

  auto ID = IDExpr->getIDs()->toString();

  auto SV = this->Names.get(ID);
  if (SV.isExisting())
    return false;

  if (getTernaryBuintinFunc(this->mod(), ID)) {
    bool IsLast = this->LastExpr;
    this->LastExpr = false;

    auto Tup = Right->asTupleExpression();
    assert(Tup);
    auto Seq = Tup->getExprs();
    assert(Seq->size() == 3);

    this->doVisit((*Seq)[0]);
    auto LeftRes = this->getExprResult();
    this->doVisit((*Seq)[1]);
    auto MidRes = this->getExprResult();
    this->doVisit((*Seq)[2]);
    auto RightRes = this->getExprResult();
    if (ID == "Array.update") {
      auto Fun = this->mod()->getFunction("doarrayupdate");
      assert(Fun);
      llvm::SmallVector<llvm::Value *, 3> Args;
      Args.push_back(this->loadTemp(LeftRes));
      Args.push_back(this->loadTemp(MidRes));
      Args.push_back(this->loadTemp(RightRes));
      auto ExprResultTmp =
          llvm::CallInst::Create(Fun, Args, "", this->BasicBlock);
      ExprResultTmp->setTailCall();
      this->LastExpr = IsLast;
      this->storeNextTemp(ExprResultTmp);
    } else {
      fatalExit("unexpected ternary buintin function: " + ID);
    }
  } else if (getBinaryBuintinFunc(this->mod(), ID)) {
    bool IsLast = this->LastExpr;
    this->LastExpr = false;

    auto Tup = Right->asTupleExpression();
    assert(Tup);
    auto Seq = Tup->getExprs();
    assert(Seq->size() == 2);

    this->doVisit((*Seq)[0]);
```

```
    auto LeftRes = this->getExprResult();
    this->doVisit((*Seq)[1]);
    auto RightRes = this->getExprResult();
    if (ID == "+") {
      LeftRes =
          unmarkValue(this->mod(), this->loadTemp(LeftRes),
              this->BasicBlock);
      RightRes =
          unmarkValue(this->mod(), this->loadTemp(RightRes),
              this->BasicBlock);
      llvm::Value *ExprResultTmp = llvm::BinaryOperator::CreateAdd(
          LeftRes, RightRes, "", this->BasicBlock);
      ExprResultTmp = markValue(this->mod(), ExprResultTmp,
          this->BasicBlock);
      this->LastExpr = IsLast;
      this->storeNextTemp(ExprResultTmp);
    } else if (ID == "-") {
      LeftRes =
          unmarkValue(this->mod(), this->loadTemp(LeftRes),
              this->BasicBlock);
      RightRes =
          unmarkValue(this->mod(), this->loadTemp(RightRes),
              this->BasicBlock);
      llvm::Value *ExprResultTmp = llvm::BinaryOperator::CreateSub(
          LeftRes, RightRes, "", this->BasicBlock);
      ExprResultTmp = markValue(this->mod(), ExprResultTmp,
          this->BasicBlock);
      this->LastExpr = IsLast;
      this->storeNextTemp(ExprResultTmp);
    } else if (ID == "*") {
      LeftRes =
          unmarkValue(this->mod(), this->loadTemp(LeftRes),
              this->BasicBlock);
      RightRes =
          unmarkValue(this->mod(), this->loadTemp(RightRes),
              this->BasicBlock);
      llvm::Value *ExprResultTmp = llvm::BinaryOperator::CreateMul(
          LeftRes, RightRes, "", this->BasicBlock);
      ExprResultTmp = markValue(this->mod(), ExprResultTmp,
          this->BasicBlock);
      this->LastExpr = IsLast;
      this->storeNextTemp(ExprResultTmp);
    } else if (ID == "div") {
      LeftRes =
          unmarkValue(this->mod(), this->loadTemp(LeftRes),
              this->BasicBlock);
      RightRes =
          unmarkValue(this->mod(), this->loadTemp(RightRes),
              this->BasicBlock);
      llvm::Value *ExprResultTmp =
          llvm::BinaryOperator::CreateSDiv(
          LeftRes, RightRes, "", this->BasicBlock);
      ExprResultTmp = markValue(this->mod(), ExprResultTmp,
          this->BasicBlock);
      this->LastExpr = IsLast;
      this->storeNextTemp(ExprResultTmp);
    } else if (ID == "mod") {
      LeftRes =
          unmarkValue(this->mod(), this->loadTemp(LeftRes),
              this->BasicBlock);
      RightRes =
          unmarkValue(this->mod(), this->loadTemp(RightRes),
```

```
                this->BasicBlock);
        llvm::Value *ExprResultTmp =
            llvm::BinaryOperator::CreateSRem(
            LeftRes, RightRes, "", this->BasicBlock);
        ExprResultTmp = markValue(this->mod(), ExprResultTmp,
            this->BasicBlock);
        this->LastExpr = IsLast;
        this->storeNextTemp(ExprResultTmp);
    } else if (ID == "=") {
        auto B =
            new llvm::ICmpInst(*this->BasicBlock,
                llvm::CmpInst::ICMP_EQ,
                               this->loadTemp(LeftRes),
                                   this->loadTemp(RightRes));
        llvm::Value *ExprResultTmp =
            llvm::CastInst::CreateIntegerCast(
            B, getInteger64(this->mod()), false, "",
                this->BasicBlock);
        ExprResultTmp = markValue(this->mod(), ExprResultTmp,
            this->BasicBlock);
        this->LastExpr = IsLast;
        this->storeNextTemp(ExprResultTmp);
    } else if (ID == "<>") {
        auto B =
            new llvm::ICmpInst(*this->BasicBlock,
                llvm::CmpInst::ICMP_NE,
                               this->loadTemp(LeftRes),
                                   this->loadTemp(RightRes));
        llvm::Value *ExprResultTmp =
            llvm::CastInst::CreateIntegerCast(
            B, getInteger64(this->mod()), false, "",
                this->BasicBlock);
        ExprResultTmp = markValue(this->mod(), ExprResultTmp,
            this->BasicBlock);
        this->LastExpr = IsLast;
        this->storeNextTemp(ExprResultTmp);
    } else if (ID == "<") {
        auto B =
            new llvm::ICmpInst(*this->BasicBlock,
                llvm::CmpInst::ICMP_SLT,
                               this->loadTemp(LeftRes),
                                   this->loadTemp(RightRes));
        llvm::Value *ExprResultTmp =
            llvm::CastInst::CreateIntegerCast(
            B, getInteger64(this->mod()), false, "",
                this->BasicBlock);
        ExprResultTmp = markValue(this->mod(), ExprResultTmp,
            this->BasicBlock);
        this->LastExpr = IsLast;
        this->storeNextTemp(ExprResultTmp);
    } else if (ID == ">") {
        auto B =
            new llvm::ICmpInst(*this->BasicBlock,
                llvm::CmpInst::ICMP_SGT,
                               this->loadTemp(LeftRes),
                                   this->loadTemp(RightRes));
        llvm::Value *ExprResultTmp =
            llvm::CastInst::CreateIntegerCast(
            B, getInteger64(this->mod()), false, "",
                this->BasicBlock);
        ExprResultTmp = markValue(this->mod(), ExprResultTmp,
            this->BasicBlock);
```

```
      this->LastExpr = IsLast;
      this->storeNextTemp(ExprResultTmp);
    } else if (ID == "<=") {
      auto B =
          new llvm::ICmpInst(*this->BasicBlock,
              llvm::CmpInst::ICMP_SLE,
                                 this->loadTemp(LeftRes),
                                     this->loadTemp(RightRes));
      llvm::Value *ExprResultTmp =
          llvm::CastInst::CreateIntegerCast(
          B, getInteger64(this->mod()), false, "",
              this->BasicBlock);
      ExprResultTmp = markValue(this->mod(), ExprResultTmp,
          this->BasicBlock);
      this->LastExpr = IsLast;
      this->storeNextTemp(ExprResultTmp);
    } else if (ID == ">=") {
      auto B =
          new llvm::ICmpInst(*this->BasicBlock,
              llvm::CmpInst::ICMP_SGE,
                                 this->loadTemp(LeftRes),
                                     this->loadTemp(RightRes));
      llvm::Value *ExprResultTmp =
          llvm::CastInst::CreateIntegerCast(
          B, getInteger64(this->mod()), false, "",
              this->BasicBlock);
      ExprResultTmp = markValue(this->mod(), ExprResultTmp,
          this->BasicBlock);
      this->LastExpr = IsLast;
      this->storeNextTemp(ExprResultTmp);
    } else if (ID == ":=") {
      auto Ref = new llvm::IntToPtrInst(this->loadTemp(LeftRes),
                                        getInteger64Pointer(this->mod()),
                                            "",
                                        this->BasicBlock);
#if 0
      auto S = new llvm::StoreInst(this->loadTemp(RightRes), Ref,
          false,
                                   this->BasicBlock);
      S->setAlignment(alignment);
#else
      this->addGCStore(this->loadTemp(RightRes), Ref, Ref);
#endif
      llvm::Value *ExprResultTmp = getZeroConst(this->mod());
      this->LastExpr = IsLast;
      this->storeNextTemp(ExprResultTmp);
    } else if (ID == "Array.array") {
      auto Fun = this->mod()->getFunction("doarray");
      assert(Fun);
      llvm::SmallVector<llvm::Value *, 2> Args;
      Args.push_back(this->loadTemp(LeftRes));
      Args.push_back(this->loadTemp(RightRes));
      auto ExprResultTmp =
          llvm::CallInst::Create(Fun, Args, "", this->BasicBlock);
      ExprResultTmp->setTailCall();
      this->LastExpr = IsLast;
      this->storeNextTemp(ExprResultTmp);
    } else if (ID == "Array.get") {
      auto Fun = this->mod()->getFunction("doarrayget");
      assert(Fun);
      llvm::SmallVector<llvm::Value *, 2> Args;
      Args.push_back(this->loadTemp(LeftRes));
```

```
      Args.push_back(this->loadTemp(RightRes));
      auto ExprResultTmp =
          llvm::CallInst::Create(Fun, Args, "", this->BasicBlock);
      ExprResultTmp->setTailCall();
      this->LastExpr = IsLast;
      this->storeNextTemp(ExprResultTmp);
    } else {
      fatalExit("unexpected binary buintin function: " + ID);
    }
    return true;
  } else if (getUnaryBuintinFunc(this->mod(), ID)) {
    bool IsLast = this->LastExpr;
    this->LastExpr = false;

    this->doVisit(Right);
    auto Res = this->getExprResult();
    if (ID == "print") {
      llvm::SmallVector<llvm::Value *, 1> Args;
      Args.push_back(this->loadTemp(Res));

      auto Fun = this->mod()->getFunction("print");
      assert(Fun);
      auto Call = llvm::CallInst::Create(Fun, Args, "",
          this->BasicBlock);
      Call->setTailCall();
      auto ExprResultTmp = new llvm::PtrToIntInst(
          Call, getInteger64(this->mod()), "", this->BasicBlock);
      this->LastExpr = IsLast;
      this->storeNextTemp(ExprResultTmp);
    } else if (ID == "~") {
      Res = unmarkValue(this->mod(), this->loadTemp(Res),
          this->BasicBlock);
      llvm::Value *ExprResultTmp =
          llvm::BinaryOperator::CreateNeg(Res, "",
              this->BasicBlock);
      ExprResultTmp = markValue(this->mod(), ExprResultTmp,
          this->BasicBlock);
      this->LastExpr = IsLast;
      this->storeNextTemp(ExprResultTmp);
    } else if (ID == "not") {
      Res = unmarkValue(this->mod(), this->loadTemp(Res),
          this->BasicBlock);
      this->LastExpr = IsLast;
      this->nextTemp();
      this->LastExpr = false;
      auto Ptr = this->getExprResult();
      auto TB = llvm::BasicBlock::Create(llvm::getGlobalContext());
      auto FB = llvm::BasicBlock::Create(llvm::getGlobalContext());
      auto End =
          llvm::BasicBlock::Create(llvm::getGlobalContext());
      auto Cond = new llvm::ICmpInst(*this->BasicBlock,
          llvm::CmpInst::ICMP_EQ,
                                    Res, getConst(this->mod(),
                                        1));
      llvm::BranchInst::Create(TB, FB, Cond, this->BasicBlock);
      TB->insertInto(this->peekFunc()->func());
      auto S = addVolatileStore(getConst(this->mod(),
          markValue(0)), Ptr, TB);
      S->setAlignment(alignment);
      llvm::BranchInst::Create(End, TB);

      FB->insertInto(this->peekFunc()->func());
```

```
      S = addVolatileStore(getConst(this->mod(), markValue(1)),
          Ptr, FB);
      S->setAlignment(alignment);
      llvm::BranchInst::Create(End, FB);

      this->BasicBlock = End;
      this->BasicBlock->insertInto(this->peekFunc()->func());
      // this->nextTemp() is called and result is stored.
      this->LastExpr = IsLast;
    } else if (ID == "ref") {
      auto Ptr = this->addAllocateCall(this->BasicBlock,
                                       getConst(this->mod(),
                                           alignment));
      auto ExprResultTmp = new llvm::PtrToIntInst(
          Ptr, getInteger64(this->mod()), "", this->BasicBlock);
      this->LastExpr = IsLast;
      this->storeNextTemp(ExprResultTmp);
      this->LastExpr = false;
#if 0
      auto S = new llvm::StoreInst(this->loadTemp(Res), Ptr, false,
                                   this->BasicBlock);
      S->setAlignment(alignment);
#else
      this->addGCStore(this->loadTemp(Res), ExprResultTmp,
          ExprResultTmp);
#endif
      this->LastExpr = IsLast;
    } else if (ID == "Array.length") {
      auto Ptr = new llvm::IntToPtrInst(this->loadTemp(Res),
                                        getInteger64Pointer(this->mod()),
                                            "",
                                        this->BasicBlock);
#if 0
      auto ExprResultTmp = new llvm::LoadInst(Ptr, "",
          this->BasicBlock);
#else
      auto ExprResultTmp = this->addGCLoad(Ptr, Ptr);
#endif
      this->LastExpr = IsLast;
      this->storeNextTemp(ExprResultTmp);
    } else if (ID == "!") {
      auto Ptr = new llvm::IntToPtrInst(this->loadTemp(Res),
                                        getInteger64Pointer(this->mod()),
                                            "",
                                        this->BasicBlock);
#if 0
      auto ExprResultTmp = new llvm::LoadInst(Ptr, "",
          this->BasicBlock);
#else
      auto ExprResultTmp = this->addGCLoad(Ptr, Ptr);
#endif
      this->LastExpr = IsLast;
      this->storeNextTemp(ExprResultTmp);
    } else {
      fatalExit("unexpected unary buintin function: " + ID);
    }
    return true;
  }

  return false;
}

```

```
void CodegenVisitor::visit(ApplyExpression *N) {
  if (this->tryBuiltinApplyExpr(N))
    return;

  auto FB = this->FalseBlock;
  auto TB = this->TrueBlock;

  bool IsLast = this->LastExpr;
  this->LastExpr = false;

  size_t Count = 0;
  llvm::Value *First = nullptr;
  llvm::Value *Prev = nullptr;
  bool FirstWasInstance = false;
  for (size_t I = 0; I < N->size(); ++I) {
    auto C = (*N)[I];

    if (N->size() == 1)
      this->LastExpr = IsLast;

    this->TrueBlock = nullptr;
    this->FalseBlock = nullptr;
    this->ExprWasTypeInstance = false;
    this->doVisit(C);
    if (!Count) {
      First = this->getExprResult();
      FirstWasInstance = this->ExprWasTypeInstance;
    }

    if (Count == 1 && FirstWasInstance) {
      auto Expr = new llvm::IntToPtrInst(this->loadTemp(First),
                                         getInteger64Pointer(this->mod()),
                                             "",
                                         this->BasicBlock);
      llvm::SmallVector<llvm::Value *, 1> Indices;
      Indices.push_back(getConst(this->mod(), 1));
      auto Ptr = llvm::GetElementPtrInst::Create(
          getInteger64(this->mod()), Expr, Indices, "",
              this->BasicBlock);
#if 0
      auto S =
          new llvm::StoreInst(this->loadTemp(), Ptr, false,
              this->BasicBlock);
      S->setAlignment(alignment);
#else
      this->addGCStore(this->loadTemp(), Ptr, Expr);
#endif
      this->setExprResult(First);
      assert(I == N->size() - 1);
      if (IsLast)
        this->setExprResult(this->loadTemp());
    } else if (Count) {
      // Prev is a pointer to a tuple (i64, i64) where the first
          is a pointer
      // to a function and the second is the pointer to the prev
          environment.
      // The prev environment should be the first argument to the
          function and
      // ExprResult should be second argument to the function.
      auto TupPtr = new llvm::IntToPtrInst(this->loadTemp(Prev),
                                           getInteger64Pointer(this->mod()),
                                               "",
```

```
                                                    this->BasicBlock);
#if 0
      auto FunInt = new llvm::LoadInst(TupPtr, "",
          this->BasicBlock);
#else
      auto FunInt = this->addGCLoad(TupPtr, TupPtr);
#endif
      auto Fun = new llvm::IntToPtrInst(
          FunInt,
          llvm::PointerType::getUnqual(getStandardFuncType(this->mod())),
              "",
          this->BasicBlock);

      llvm::SmallVector<llvm::Value *, 1> Indices;
      Indices.push_back(getConst(this->mod(), 1));
      auto EnvPtr = llvm::GetElementPtrInst::Create(
          getInteger64(this->mod()), TupPtr, Indices, "",
              this->BasicBlock);
#if 0
      auto Env = new llvm::LoadInst(EnvPtr, "", this->BasicBlock);
#else
      auto Env = this->addGCLoad(EnvPtr, TupPtr);
#endif

      llvm::SmallVector<llvm::Value *, 2> Args;
      Args.push_back(Env);
      Args.push_back(this->loadTemp());

      auto ExprResultTmp =
          llvm::CallInst::Create(Fun, Args, "", this->BasicBlock);
      ExprResultTmp->setCallingConv(llvm::CallingConv::C);
      ExprResultTmp->setTailCall();
      if (I == N->size() - 1)
        this->LastExpr = IsLast;
      this->storeNextTemp(ExprResultTmp);
    }

    Prev = this->getExprResult();
    ++Count;
  }

  if (Count > 1)
    this->ExprWasTypeInstance = false;
  // else this->ExprWasTypeInstance is set.
  this->LastExpr = IsLast;
  this->TrueBlock = TB;
  this->FalseBlock = FB;
}

void CodegenVisitor::visit(OrElseExpression *N) {
  assert(!!this->TrueBlock == !!this->FalseBlock);

  bool IsLast = this->LastExpr;
  this->LastExpr = false;

  llvm::BasicBlock *TB = nullptr;
  llvm::BasicBlock *FB = nullptr;
  llvm::Value *Res = nullptr;
  if (!this->TrueBlock) {
    FB = llvm::BasicBlock::Create(llvm::getGlobalContext());
    TB = llvm::BasicBlock::Create(llvm::getGlobalContext());
    this->TrueBlock = TB;
```

```
    this->FalseBlock = FB;
    Res = new llvm::AllocaInst(getInteger64(this->mod()), "orres",
                               this->BasicBlock);
  }
  auto OrigT = this->TrueBlock;
  auto OrigF = this->FalseBlock;

  auto TmpFB = llvm::BasicBlock::Create(llvm::getGlobalContext());
  this->FalseBlock = TmpFB;
  this->doVisit(N->getLeftExpr());
  auto Cond =
      new llvm::ICmpInst(*this->BasicBlock, llvm::CmpInst::ICMP_EQ,
                         this->loadTemp(), getConst(this->mod(),
                             markValue(1)));
  llvm::BranchInst::Create(OrigT, TmpFB, Cond, this->BasicBlock);

  this->BasicBlock = TmpFB;
  this->BasicBlock->insertInto(this->peekFunc()->func());

  this->TrueBlock = OrigT;
  this->FalseBlock = OrigF;
  this->doVisit(N->getRightExpr());
  if (Res) {
    Cond = new llvm::ICmpInst(*this->BasicBlock,
        llvm::CmpInst::ICMP_EQ,
                              this->loadTemp(),
                              getConst(this->mod(), markValue(1)));
    llvm::BranchInst::Create(OrigT, OrigF, Cond, this->BasicBlock);

    auto Merge =
        llvm::BasicBlock::Create(llvm::getGlobalContext());

    TB->insertInto(this->peekFunc()->func());
    auto Store = new llvm::StoreInst(getConst(this->mod(),
        markValue(1)), Res,
                                     false, TB);
    Store->setAlignment(alignment);
    llvm::BranchInst::Create(Merge, TB);

    FB->insertInto(this->peekFunc()->func());
    Store = new llvm::StoreInst(getConst(this->mod(),
        markValue(0)), Res, false,
                                FB);
    Store->setAlignment(alignment);
    llvm::BranchInst::Create(Merge, FB);

    this->BasicBlock = Merge;
    this->BasicBlock->insertInto(this->peekFunc()->func());
    this->LastExpr = IsLast;
    this->storeNextTemp(new llvm::LoadInst(Res, "",
        this->BasicBlock));
  }

  this->LastExpr = IsLast;
  this->ExprWasTypeInstance = false;
}

void CodegenVisitor::visit(AndAlsoExpression *N) {
  assert(!!this->TrueBlock == !!this->FalseBlock);

  bool IsLast = this->LastExpr;
  this->LastExpr = false;
```

```
    llvm::BasicBlock *TB = nullptr;
    llvm::BasicBlock *FB = nullptr;
    llvm::Value *Res = nullptr;
    if (!this->TrueBlock) {
      FB = llvm::BasicBlock::Create(llvm::getGlobalContext());
      TB = llvm::BasicBlock::Create(llvm::getGlobalContext());
      this->TrueBlock = TB;
      this->FalseBlock = FB;
      Res = new llvm::AllocaInst(getInteger64(this->mod()), "andres",
                                 this->BasicBlock);
    }
    auto OrigT = this->TrueBlock;
    auto OrigF = this->FalseBlock;

    auto TmpTB = llvm::BasicBlock::Create(llvm::getGlobalContext());
    this->TrueBlock = TmpTB;
    this->doVisit(N->getLeftExpr());
    auto Cond =
        new llvm::ICmpInst(*this->BasicBlock, llvm::CmpInst::ICMP_EQ,
                           this->loadTemp(), getConst(this->mod(),
                               markValue(1)));
    llvm::BranchInst::Create(TmpTB, OrigF, Cond, this->BasicBlock);

    this->BasicBlock = TmpTB;
    this->BasicBlock->insertInto(this->peekFunc()->func());

    this->TrueBlock = OrigT;
    this->FalseBlock = OrigF;
    this->doVisit(N->getRightExpr());
    if (Res) {
      Cond = new llvm::ICmpInst(*this->BasicBlock,
          llvm::CmpInst::ICMP_EQ,
                                this->loadTemp(),
                                getConst(this->mod(), markValue(1)));
      llvm::BranchInst::Create(OrigT, OrigF, Cond, this->BasicBlock);

      auto Merge =
          llvm::BasicBlock::Create(llvm::getGlobalContext());

      TB->insertInto(this->peekFunc()->func());
      auto Store = new llvm::StoreInst(getConst(this->mod(),
          markValue(1)), Res,
                                       false, TB);
      Store->setAlignment(alignment);
      llvm::BranchInst::Create(Merge, TB);

      FB->insertInto(this->peekFunc()->func());
      Store = new llvm::StoreInst(getConst(this->mod(),
          markValue(0)), Res, false,
                                  FB);
      Store->setAlignment(alignment);
      llvm::BranchInst::Create(Merge, FB);

      this->BasicBlock = Merge;
      this->BasicBlock->insertInto(this->peekFunc()->func());
      this->LastExpr = IsLast;
      this->storeNextTemp(new llvm::LoadInst(Res, "",
          this->BasicBlock));
    }

    this->LastExpr = IsLast;
```

```
  this->ExprWasTypeInstance = false;
}

void CodegenVisitor::visit(LetExpression *N) {
  auto FB = this->FalseBlock;
  auto TB = this->TrueBlock;
  this->enterScope();

  bool IsLast = this->LastExpr;
  this->LastExpr = false;

  bool PrevInLet = this->InsideLetDecls;
  this->InsideLetDecls = true;
  this->doVisit(N->getDecls());
  this->InsideLetDecls = PrevInLet;

  this->LastExpr = IsLast;
  this->TrueBlock = TB;
  this->FalseBlock = FB;
  this->doVisit(N->getExprs());

  this->leaveScope(!IsLast);
  this->ExprWasTypeInstance = false;
}

void CodegenVisitor::visit(IfExpression *N) {
  auto FB = this->FalseBlock;
  auto TB = this->TrueBlock;

  bool IsLast = this->LastExpr;

  auto NTB = llvm::BasicBlock::Create(llvm::getGlobalContext());
  auto NFB = llvm::BasicBlock::Create(llvm::getGlobalContext());
  auto End = llvm::BasicBlock::Create(llvm::getGlobalContext());

  auto Res = new llvm::AllocaInst(getInteger64(this->mod()),
      "ifres",
                                  this->BasicBlock);

  this->LastExpr = false;
  this->TrueBlock = nullptr;
  this->FalseBlock = nullptr;
  this->doVisit(N->getCondExpr());
  auto Cond =
      new llvm::ICmpInst(*this->BasicBlock, llvm::CmpInst::ICMP_EQ,
                         this->loadTemp(), getConst(this->mod(),
                             markValue(1)));
  llvm::BranchInst::Create(NTB, NFB, Cond, this->BasicBlock);

  this->TrueBlock = nullptr;
  this->FalseBlock = nullptr;
  this->BasicBlock = NTB;
  this->BasicBlock->insertInto(this->peekFunc()->func());
  this->LastExpr = IsLast;
  this->doVisit(N->getThenExpr());
  llvm::StoreInst *TStore;
  if (!IsLast)
    TStore =
        new llvm::StoreInst(this->loadTemp(), Res, false,
            this->BasicBlock);
  else
    TStore = new llvm::StoreInst(this->getExprResult(), Res, false,
```

```
                                  this->BasicBlock);
  TStore->setAlignment(alignment);
  llvm::BranchInst::Create(End, this->BasicBlock);

  this->TrueBlock = nullptr;
  this->FalseBlock = nullptr;
  this->BasicBlock = NFB;
  this->BasicBlock->insertInto(this->peekFunc()->func());
  this->LastExpr = IsLast;
  this->doVisit(N->getElseExpr());
  llvm::StoreInst *FStore;
  if (!IsLast)
    FStore =
        new llvm::StoreInst(this->loadTemp(), Res, false,
            this->BasicBlock);
  else
    FStore = new llvm::StoreInst(this->getExprResult(), Res, false,
                                  this->BasicBlock);
  FStore->setAlignment(alignment);
  llvm::BranchInst::Create(End, this->BasicBlock);

  this->BasicBlock = End;
  this->BasicBlock->insertInto(this->peekFunc()->func());

  this->LastExpr = IsLast;
  this->storeNextTemp(new llvm::LoadInst(Res, "",
      this->BasicBlock));
  this->ExprWasTypeInstance = false;
  this->TrueBlock = TB;
  this->FalseBlock = FB;
}

void CodegenVisitor::visit(WhileExpression *N) {
  auto FB = this->FalseBlock;
  auto TB = this->TrueBlock;

  bool IsLast = this->LastExpr;
  this->LastExpr = false;

  auto Start = llvm::BasicBlock::Create(llvm::getGlobalContext());
  auto NTB = llvm::BasicBlock::Create(llvm::getGlobalContext());
  auto NFB = llvm::BasicBlock::Create(llvm::getGlobalContext());

  llvm::BranchInst::Create(Start, this->BasicBlock);

  this->TrueBlock = nullptr;
  this->FalseBlock = nullptr;
  this->BasicBlock = Start;
  this->BasicBlock->insertInto(this->peekFunc()->func());
  this->doVisit(N->getCondExpr());
  auto Cond =
      new llvm::ICmpInst(*this->BasicBlock, llvm::CmpInst::ICMP_EQ,
                         this->loadTemp(), getConst(this->mod(),
                             markValue(1)));
  llvm::BranchInst::Create(NTB, NFB, Cond, this->BasicBlock);

  this->TrueBlock = nullptr;
  this->FalseBlock = nullptr;
  this->BasicBlock = NTB;
  this->BasicBlock->insertInto(this->peekFunc()->func());
  this->doVisit(N->getBodyExpr());
  llvm::BranchInst::Create(Start, this->BasicBlock);
```

```

  this->BasicBlock = NFB;
  this->BasicBlock->insertInto(this->peekFunc()->func());

  this->LastExpr = IsLast;
  this->storeNextTemp(getZeroConst(this->mod()));
  this->ExprWasTypeInstance = false;
  this->TrueBlock = TB;
  this->FalseBlock = FB;
}

void CodegenVisitor::visit(Match *N) {
  auto FB = this->FalseBlock;

  this->peekFunc()->restoreRememberedContext();
  this->PatternExpr =
      this->loadTemp(this->peekFunc()->getSecondArg());
  this->TrueBlock =
      llvm::BasicBlock::Create(llvm::getGlobalContext());
  this->doVisit(N->getPattern());
  this->TrueBlock->insertInto(this->peekFunc()->func());
  this->BasicBlock = this->TrueBlock;

  bool PrevLast = this->LastExpr;
  this->LastExpr = true;
  this->TrueBlock = nullptr;
  this->FalseBlock = nullptr;
  this->doVisit(N->getExpr());
  this->LastExpr = PrevLast;

#if 0
  {
    llvm::SmallVector<llvm::Value *, 1> Arg;
    Arg.push_back(this->ExprResultTmp);
    auto Fun = this->mod()->getFunction("print");
    assert(Fun);
    llvm::CallInst::Create(Fun, Arg, "", this->BasicBlock);
  }
#endif

  llvm::ReturnInst::Create(this->mod()->getContext(),
      this->getExprResult(),
                           this->BasicBlock);
  this->FalseBlock = FB;
}

void CodegenVisitor::visit(SeqMatch *N) {
  auto Size = N->size();
  for (size_t I = 0; I < Size; ++I) {
    if (I == Size - 1)
      this->FalseBlock =
          getMatchErrorBasicBlock(this->mod(),
              "lambda_match_fail");
    else
      this->FalseBlock =
          llvm::BasicBlock::Create(llvm::getGlobalContext());
    this->doVisit((*N)[I]);
    this->FalseBlock->insertInto(this->peekFunc()->func());
    this->BasicBlock = this->FalseBlock;
  }
#if 0
  llvm::ReturnInst::Create(this->mod()->getContext(),
```

```
      getZeroConst(this->mod()),
                              this->BasicBlock);
#endif
}

void CodegenVisitor::visit(LambdaExpression *N) {
  static size_t NextLambdaNumber = 0;
  ++NextLambdaNumber;

  bool IsLast = this->LastExpr;
  this->LastExpr = false;

  auto OrigBB = this->BasicBlock;
  auto OrigTrueBB = this->TrueBlock;
  auto OrigFalseBB = this->FalseBlock;

  // 1. Insert new function definition in module.
  // 2. Restore BBs.
  // 3. Allocate memory for function pointer and current frame.
  // 4. Put function pointer and frame in allocated memory.
  // 5. Let ExprResult = allocated memory.

  auto FunType = getStandardFuncType(this->mod());
  auto Lambda = llvm::Function::Create(
      FunType, llvm::GlobalValue::ExternalLinkage,
      "_" + std::to_string(NextLambdaNumber), this->mod());
  Lambda->setCallingConv(llvm::CallingConv::C);
  setGC(Lambda);

  auto EntryBB =
      llvm::BasicBlock::Create(this->mod()->getContext(),
          "entrylabel", Lambda);
  this->BasicBlock = EntryBB;
  this->TrueBlock = nullptr;
  this->FalseBlock = nullptr;

  this->pushFunc(Lambda, N->getFunctionVals());
  this->addFuncEntry(this->peekFunc(), false, false);
  this->addLocalFuncVal("context",
                         this->loadTemp(this->peekFunc()->getFirstArg()));
  this->enterScope();
  this->doVisit(N->getCases());
  this->leaveScope(false);
  this->popFunc();

  this->BasicBlock = OrigBB;
  this->TrueBlock = OrigTrueBB;
  this->FalseBlock = OrigFalseBB;

  auto Result =
      this->addAllocateCall(this->BasicBlock,
          getConst(this->mod(), 16));
  auto ExprResultTmp = new llvm::PtrToIntInst(Result,
      getInteger64(this->mod()),
                                                 "",
                                                   this->BasicBlock);
  this->LastExpr = IsLast;
  this->storeNextTemp(ExprResultTmp);
  this->LastExpr = false;

  auto Val = new llvm::PtrToIntInst(Lambda,
      getInteger64(this->mod()), "",
```

```
                                        this->BasicBlock);
#if 0
  auto Store = new llvm::StoreInst(Val, Result, false,
      this->BasicBlock);
  Store->setAlignment(alignment);
#else
  this->addGCStore(Val, ExprResultTmp, ExprResultTmp);
#endif

  llvm::SmallVector<llvm::Value *, 1> Indices;
  Indices.push_back(getConst(this->mod(), 1));
  auto Ptr =
      llvm::GetElementPtrInst::Create(getInteger64(this->mod()),
      Result,
                                                  Indices, "",
                                                      this->BasicBlock);
#if 0
  llvm::Value *Locals =
      new llvm::LoadInst(this->peekFunc()->getLocalPtr(), "",
          this->BasicBlock);
#else
  llvm::Value *Locals = this->loadLocalPtr();
#endif
  Val = new llvm::PtrToIntInst(Locals, getInteger64(this->mod()),
      "",
                                  this->BasicBlock);
#if 0
  Store = new llvm::StoreInst(Val, Ptr, false, this->BasicBlock);
  Store->setAlingment(alignment);
#else
  this->addGCStore(Val, Ptr, ExprResultTmp);
#endif

  // this->Temp is set from prev call to
      this->storeNextTemp(ExprResultTmp).
  this->LastExpr = IsLast;
  this->ExprWasTypeInstance = false;
}

void CodegenVisitor::visit(SeqPattern *N) {
  for (auto &&C : *N)
    this->doVisit(C);
}

void CodegenVisitor::visit(LiteralPattern *N) {
  auto Val = markValue(N->getValue()->toInt());
  // callPatternCheck(this->mod(), this->BasicBlock,
      this->PatternExpr);
  auto Cond = new llvm::ICmpInst(*this->BasicBlock,
      llvm::CmpInst::ICMP_EQ,
                                   this->PatternExpr,
                                       getConst(this->mod(), Val));
  llvm::BranchInst::Create(this->TrueBlock, this->FalseBlock, Cond,
                           this->BasicBlock);
}

void CodegenVisitor::visit(WildcardPattern *N) {
  llvm::BranchInst::Create(this->TrueBlock, this->BasicBlock);
}

void CodegenVisitor::addLocalFuncVal(const std::string &Identifier,
                                       llvm::Value *Val) {
```

```
  this->peekFunc()->addLocal(Identifier);
  auto idx = this->peekFunc()->getLocalIndex(Identifier);
  llvm::SmallVector<llvm::Value *, 1> Indices;
  Indices.push_back(getConst(this->mod(), idx));
#if 0
  llvm::Value *Loc =
      new llvm::LoadInst(this->peekFunc()->getLocalPtr(), "", this->BasicBlock);
#else
  llvm::Value *Loc = this->loadLocalPtr();
#endif
  auto Ptr =
      llvm::GetElementPtrInst::Create(getInteger64(this->mod()), Loc,
                                      Indices, "", this->BasicBlock);
// No need to store Loc in temp storage since GC cannot be triggered here.
#if 0
  auto Store = new llvm::StoreInst(Val, Ptr, false, this->BasicBlock);
  Store->setAlignment(alignment);
#else
  this->addGCStore(Val, Ptr, Loc);
#endif
  this->Names.insert(Identifier, {this->peekFunc()->getNest(), idx});
  this->peekFunc()->rememberContext();
  this->peekFunc()->appendLocalIndex(idx);
}

void CodegenVisitor::insertTypeInstances() {
  this->Instances.insert("::", 0, true);
  this->Names.insert("::", {this->peekFunc()->getNest(), -1});
  this->Instances.insert("nil", 1, false);
  this->Names.insert("nil", {this->peekFunc()->getNest(), -1});
}

void CodegenVisitor::visit(LongIdentifierPattern *N) {
  auto Identifier = N->getIDs()->toString();
  auto Name = this->Names.get(Identifier);
  if (Name.isTypeInstance()) {
    auto Struct = new llvm::IntToPtrInst(this->PatternExpr,
                                         getInteger64Pointer(this->mod()), "",
                                         this->BasicBlock);
#if 0
    llvm::Value *Val = new llvm::LoadInst(Struct, "", this->BasicBlock);
#else
    llvm::Value *Val = this->addGCLoad(Struct, Struct);
#endif
    Val = unmarkValue(this->mod(), Val, this->BasicBlock);
    // callPatternCheck(this->mod(), this->BasicBlock, Val);

    auto ConstValue = this->Instances.getValue(Identifier);
    auto Cond = new llvm::ICmpInst(*this->BasicBlock, llvm::CmpInst::ICMP_EQ,
                                   Val, getConst(this->mod(), ConstValue));
    llvm::BranchInst::Create(this->TrueBlock, this->FalseBlock, Cond,
```

```
                                this->BasicBlock);
  } else {
    auto Builtin = getSimpleBuiltinConst(this->mod(), Identifier, true);
    if (Builtin) {
      auto Cond = new llvm::ICmpInst(*this->BasicBlock, llvm::CmpInst::ICMP_EQ,
                                     this->PatternExpr, Builtin);
      llvm::BranchInst::Create(this->TrueBlock, this->FalseBlock, Cond,
                               this->BasicBlock);
    } else {
      this->addLocalFuncVal(Identifier, this->PatternExpr);
      llvm::BranchInst::Create(this->TrueBlock, this->BasicBlock);
    }
  }
}

void CodegenVisitor::visit(TypePattern *N) { this->doVisit(N->getPattern()); }

void CodegenVisitor::visit(ApplyPattern *N) {
  size_t Count = 0;
  auto OrigResult = this->PatternExpr;
  for (auto &&C : *N) {
    if (Count == 1) {
      this->BasicBlock = this->TrueBlock;
      this->BasicBlock->insertInto(this->peekFunc()->func());
      this->TrueBlock = llvm::BasicBlock::Create(llvm::getGlobalContext());
      auto Expr = new llvm::IntToPtrInst(
          OrigResult, getInteger64Pointer(this->mod()), "", this->BasicBlock);
      llvm::SmallVector<llvm::Value *, 1> Indices;
      Indices.push_back(getConst(this->mod(), 1));
      auto Ptr = llvm::GetElementPtrInst::Create(
          getInteger64(this->mod()), Expr, Indices, "", this->BasicBlock);
#if 0
      this->PatternExpr = new llvm::LoadInst(Ptr, "", this->BasicBlock);
#else
      this->PatternExpr = this->addGCLoad(Ptr, Expr);
#endif
    } else if (Count > 1) {
      fatalExit("too many apply pattern arguments");
    }
    this->doVisit(C);
    ++Count;
  }
}

void CodegenVisitor::visit(ListPattern *N) {
  llvm::Value *ListExpr = new llvm::IntToPtrInst(
      this->PatternExpr, getInteger64Pointer(this->mod()), "",
      this->BasicBlock);

  llvm::SmallVector<llvm::Value *, 1> Indices;
  Indices.push_back(getConst(this->mod(), 1));

  size_t Idx = 0;
  auto OrigTrueBB = this->TrueBlock;
```

```
  auto Seq = N->getPatterns();
  for (auto P : *Seq) {
#if 0
    llvm::Value *First = new llvm::LoadInst(ListExpr, "",
        this->BasicBlock);
#else
    llvm::Value *First = this->addGCLoad(ListExpr, ListExpr);
#endif
    First = unmarkValue(this->mod(), First, this->BasicBlock);
    // callPatternCheck(this->mod(), this->BasicBlock, First);
    auto Cond = new llvm::ICmpInst(*this->BasicBlock,
        llvm::CmpInst::ICMP_EQ,
                                   First, getConst(this->mod(),
                                       0));

    auto NextBlock =
        llvm::BasicBlock::Create(llvm::getGlobalContext());

    llvm::BranchInst::Create(NextBlock, this->FalseBlock, Cond,
                             this->BasicBlock);

    NextBlock->insertInto(this->peekFunc()->func());
    this->BasicBlock = NextBlock;

    llvm::SmallVector<llvm::Value *, 1> Indices;
    Indices.push_back(getConst(this->mod(), 1));
    llvm::Value *Tup = llvm::GetElementPtrInst::Create(
        getInteger64(this->mod()), ListExpr, Indices, "",
            this->BasicBlock);
#if 0
    Tup = new llvm::LoadInst(Tup, "", this->BasicBlock);
#else
    Tup = this->addGCLoad(Tup, ListExpr);
#endif
    Tup = new llvm::IntToPtrInst(Tup,
        getInteger64Pointer(this->mod()), "",
                                 this->BasicBlock);
#if 0
    this->PatternExpr = new llvm::LoadInst(Tup, "",
        this->BasicBlock);
#else
    this->PatternExpr = this->addGCLoad(Tup, Tup);
#endif

    this->TrueBlock =
        llvm::BasicBlock::Create(llvm::getGlobalContext());
    this->doVisit(P);
    this->BasicBlock = this->TrueBlock;
    this->BasicBlock->insertInto(this->peekFunc()->func());

    ListExpr =
        llvm::GetElementPtrInst::Create(getInteger64(this->mod()),
        Tup,
                                        Indices, "",
                                            this->BasicBlock);
#if 0
    ListExpr = new llvm::LoadInst(ListExpr, "", this->BasicBlock);
#else
    ListExpr = this->addGCLoad(ListExpr, Tup);
#endif
    ListExpr = new llvm::IntToPtrInst(
        ListExpr, getInteger64Pointer(this->mod()), "",
```

```
            this->BasicBlock);

    ++Idx;
  }

#if 0
  llvm::Value *End = new llvm::LoadInst(ListExpr, "",
      this->BasicBlock);
#else
  llvm::Value *End = this->addGCLoad(ListExpr, ListExpr);
#endif
  End = unmarkValue(this->mod(), End, this->BasicBlock);
  // callPatternCheck(this->mod(), this->BasicBlock, End);
  auto Cond = new llvm::ICmpInst(*this->BasicBlock,
      llvm::CmpInst::ICMP_EQ, End,
                                 getConst(this->mod(), 1));
  this->TrueBlock = OrigTrueBB;
  llvm::BranchInst::Create(this->TrueBlock, this->FalseBlock, Cond,
                           this->BasicBlock);
}

void CodegenVisitor::visit(TuplePattern *N) {
  auto Seq = N->getPatterns();
  llvm::Value *TupExpr = nullptr;
  if (Seq->size())
    TupExpr = new llvm::IntToPtrInst(this->PatternExpr,
                                     getInteger64Pointer(this->mod()),
                                         "",
                                     this->BasicBlock);
  size_t Idx = 0;
  auto OrigTrueBB = this->TrueBlock;
  for (auto P : *Seq) {
    llvm::SmallVector<llvm::Value *, 1> Indices;
    Indices.push_back(getConst(this->mod(), Idx));
    auto Ptr = llvm::GetElementPtrInst::Create(
        getInteger64(this->mod()), TupExpr, Indices, "",
            this->BasicBlock);

    if (Idx < Seq->size() - 1)
      this->TrueBlock =
          llvm::BasicBlock::Create(llvm::getGlobalContext());
    else
      this->TrueBlock = OrigTrueBB;
#if 0
    this->PatternExpr = new llvm::LoadInst(Ptr, "",
        this->BasicBlock);
#else
    this->PatternExpr = this->addGCLoad(Ptr, TupExpr);
#endif
    this->doVisit(P);
    if (Idx < Seq->size() - 1) {
      this->BasicBlock = this->TrueBlock;
      this->BasicBlock->insertInto(this->peekFunc()->func());
    }

    ++Idx;
  }

  if (!Seq->size())
    llvm::BranchInst::Create(this->TrueBlock, this->BasicBlock);
}

```

```
void CodegenVisitor::visit(SeqDeclaration *N) {
  for (auto &&C : *N)
    this->doVisit(C);
}

void CodegenVisitor::visit(Root *N) {
  beginMain(N->getFunctionVals());
  this->enterScope();
  this->addLocalFuncVal("context", getZeroConst(this->mod()));
  this->insertTypeInstances();
  for (auto &&C : *N)
    this->doVisit(C);
  this->leaveScope(false);
  endMain();

  if (llvm::verifyModule(*this->mod(), &llvm::errs()))
    fatalExit("internal code generation error");

  std::error_code Err;
  llvm::tool_output_file Out("llvm-bitcode.bc", Err,
      llvm::sys::fs::F_None);
  if (Err)
    fatalExit("error opening output file");
  llvm::WriteBitcodeToFile(this->mod(), Out.os());
  Out.keep();
}

void CodegenVisitor::visit(ValDeclaration *N) {
  this->TrueBlock = nullptr;
  this->FalseBlock = nullptr;
  this->doVisit(N->getSource());
  this->PatternExpr = this->loadTemp();
  this->FalseBlock = getMatchErrorBasicBlock(this->mod(),
      "val_match_fail");
  this->TrueBlock =
      llvm::BasicBlock::Create(llvm::getGlobalContext(),
          "val_true");
  assert(this->TrueBlock != this->FalseBlock);
  this->doVisit(N->getDest());
  this->FalseBlock->insertInto(this->peekFunc()->func());
  this->TrueBlock->insertInto(this->peekFunc()->func());
  this->BasicBlock = this->TrueBlock;
}

void CodegenVisitor::visit(NonfixDeclaration *N) {
  // Nop.
}

void CodegenVisitor::visit(InfixDeclaration *N) {
  // Nop.
}

void CodegenVisitor::visit(InfixRDeclaration *N) {
  // Nop.
}

void CodegenVisitor::visit(FunPatternDeclaration *N) {
  fatalExit("CodegenVisitor visit FunPatternDeclaration");
}

static std::string functionOperatorToID(const std::string &FunID) {
  std::string Ret = "userop_";
```

```
  for (size_t I = 0; I < FunID.size(); ++I) {
    switch (FunID[0]) {
    case '!':
      Ret += "ban";
      break;
    case '%':
      Ret += "per";
      break;
    case '&':
      Ret += "amp";
      break;
    case '$':
      Ret += "dol";
      break;
    case '#':
      Ret += "has";
      break;
    case '+':
      Ret += "plu";
      break;
    case '-':
      Ret += "min";
      break;
    case '/':
      Ret += "div";
      break;
    case ':':
      Ret += "col";
      break;
    case '<':
      Ret += "les";
      break;
    case '=':
      Ret += "equ";
      break;
    case '>':
      Ret += "gre";
      break;
    case '?':
      Ret += "que";
      break;
    case '@':
      Ret += "ats";
      break;
    case '\\':
      Ret += "bac";
      break;
    case '~':
      Ret += "til";
      break;
    case '`':
      Ret += "quo";
      break;
    case '^':
      Ret += "pow";
      break;
    case '|':
      Ret += "bar";
      break;
    case '*':
      Ret += "mul";
      break;
```

```
    default:
      return FunID;
    }
  }
  return Ret;
}

static std::string extractFunctionID(std::shared_ptr<Pattern>
    IDParams) {
  auto TypePat = IDParams->asTypePattern();
  assert(TypePat);
  auto ApplyPat = TypePat->getPattern()->asApplyPattern();
  assert(ApplyPat);
  auto IDPat = ApplyPat->front()->asLongIdentifierPattern();
  assert(IDPat);
  return IDPat->getIDs()->toString();
}

static std::vector<std::shared_ptr<Pattern>>
extractFunctionParams(std::shared_ptr<Pattern> IDParams) {
  auto TypePat = IDParams->asTypePattern();
  assert(TypePat);
  auto ApplyPat = TypePat->getPattern()->asApplyPattern();
  assert(ApplyPat);
  std::vector<std::shared_ptr<Pattern>> Ret(ApplyPat->begin(),
      ApplyPat->end());
  Ret.erase(Ret.begin());
  return Ret;
}

static std::string getFunParamID(size_t I, size_t J) {
#if 0
  return "_" + std::to_string(I) + "_" + std::to_string(J);
#else
  return "_" + std::to_string(J) + "_";
#endif
}

void CodegenVisitor::visit(FunDeclaration *N) {
  auto OrigBB = this->BasicBlock;
  auto OrigTrueBB = this->TrueBlock;
  auto OrigFalseBB = this->FalseBlock;

  bool PrevLast = this->LastExpr;
  this->LastExpr = false;

  auto RealFunID = extractFunctionID(N->front()->getIDParams());
  auto FunID = functionOperatorToID(RealFunID);
  std::vector<std::vector<std::shared_ptr<Pattern>>> Params;
  for (auto P : *N)
    Params.push_back(extractFunctionParams(P->getIDParams()));
  auto ParamCount = Params.front().size();
  assert(ParamCount);

  std::shared_ptr<FunctionVals> FunVals(new FunctionVals());
  for (size_t I = 0; I < Params.size(); ++I) {
    for (size_t J = 0; J < Params[I].size() - 1; ++J) {
      auto &V = Params[I];
      bool Val = V[J]->hasSimpleType();
      FunVals->addVal(!Val);
    }
    break;
```

```
  }

  static size_t NextFunNumber = 0;
  ++NextFunNumber;

  auto FunType = getStandardFuncType(this->mod());
  auto OuterFun = llvm::Function::Create(
      FunType, llvm::GlobalValue::ExternalLinkage,
      "_" + std::to_string(NextFunNumber) + FunID, this->mod());
  OuterFun->setCallingConv(llvm::CallingConv::C);
  setGC(OuterFun);

  auto Fun = OuterFun;
  if (ParamCount - 1) {
    this->pushFunc(Fun, FunVals);
    this->peekFunc()->setIsStepFunction();
  }
  for (size_t J = 0; J < ParamCount - 1; ++J) {
    this->changeFunc(Fun);
    this->TrueBlock = nullptr;
    this->FalseBlock = nullptr;
    auto EntryBB = this->BasicBlock =
        llvm::BasicBlock::Create(this->mod()->getContext(),
            "entrylabel", Fun);

    this->addFuncEntry(this->peekFunc(), false, J != 0);
    if (J == 0)
      this->addLocalFuncVal("context",
                            this->loadTemp(this->peekFunc()->getFirstArg()));

    llvm::Value *Param =
        this->loadTemp(this->peekFunc()->getSecondArg());
#if 0
    for (size_t I = 0; I < Params.size(); ++I) {
      this->addLocalFuncVal(getFunParamID(I, J), Param);
    }
#else
    this->addLocalFuncVal(getFunParamID(0, J), Param);
#endif

    ++NextFunNumber;
    Fun = llvm::Function::Create(FunType,
        llvm::GlobalValue::ExternalLinkage,
                                 "_" +
                                     std::to_string(NextFunNumber)
                                     + FunID,
                                 this->mod());
    Fun->setCallingConv(llvm::CallingConv::C);
    setGC(Fun);

    // TODO. Find out if necessary to use gcread and gcwrite here.
    auto RetVal = this->addAllocateCall(EntryBB,
        getConst(this->mod(), 16));
    auto FunInt =
        new llvm::PtrToIntInst(Fun, getInteger64(this->mod()), "",
            EntryBB);
    auto Store = addVolatileStore(FunInt, RetVal, EntryBB);
    Store->setAlignment(alignment);

    llvm::SmallVector<llvm::Value *, 1> Indices;
    Indices.push_back(getConst(this->mod(), 1));
    auto Snd =
```

```
        llvm::GetElementPtrInst::Create(getInteger64(this->mod()),
                                        RetVal, Indices,
                                        "", EntryBB);
#if 0
    llvm::Value *Context =
        new llvm::LoadInst(this->peekFunc()->getLocalPtr(), "",
            EntryBB);
#else
    llvm::Value *Context = this->loadLocalPtr();
#endif
    Context =
        new llvm::PtrToIntInst(Context, getInteger64(this->mod()),
            "", EntryBB);
    Store = addVolatileStore(Context, Snd, EntryBB);
    Store->setAlignment(alignment);

    auto Ret =
        new llvm::PtrToIntInst(RetVal, getInteger64(this->mod()),
            "", EntryBB);
    llvm::ReturnInst::Create(this->mod()->getContext(), Ret,
        EntryBB);
  }

  this->pushFunc(Fun, N->getFunctionVals(), RealFunID, OuterFun);
  this->BasicBlock =
      llvm::BasicBlock::Create(this->mod()->getContext(),
          "entrylabel", Fun);
  this->TrueBlock = nullptr;
  this->FalseBlock = nullptr;

  this->addFuncEntry(this->peekFunc(), false, false);
  this->addLocalFuncVal("context",
                        this->loadTemp(this->peekFunc()->getFirstArg()));
  this->enterScope();

  for (size_t I = 0; I < Params.size(); ++I) {
    if (I == Params.size() - 1)
      this->FalseBlock = getMatchErrorBasicBlock(this->mod(),
          "fun_match_fail");
    else
      this->FalseBlock =
          llvm::BasicBlock::Create(llvm::getGlobalContext());

    auto &ParamList = Params[I];
    for (size_t J = 0; J < ParamList.size(); ++J) {
      this->TrueBlock =
          llvm::BasicBlock::Create(llvm::getGlobalContext());

      if (J == ParamList.size() - 1) {
        this->PatternExpr =
            this->loadTemp(this->peekFunc()->getSecondArg());
      } else {
        NameMap::ScopeValue SV = this->Names.get(getFunParamID(I,
            J));
        assert(!SV.isTypeInstance());
        assert(SV.isExisting());
        this->PatternExpr = this->addLocalLookup(this->BasicBlock,
            SV);
      }
      this->doVisit(ParamList[J]);

      this->TrueBlock->insertInto(this->peekFunc()->func());
```

```
      this->BasicBlock = this->TrueBlock;

      if (J == ParamList.size() - 1) {
        auto FB = this->FalseBlock;
        auto Expr = (*N)[I]->getDef();
        this->FalseBlock = nullptr;
        this->TrueBlock = nullptr;
        this->setExprResult(nullptr);
        this->LastExpr = true;
        this->doVisit(Expr);
        llvm::ReturnInst::Create(this->mod()->getContext(),
                                 this->getExprResult(),
                                     this->BasicBlock);

        this->FalseBlock = FB;
      }
    }

    this->FalseBlock->insertInto(this->peekFunc()->func());
    this->BasicBlock = this->FalseBlock;
  }
#if 0
  llvm::ReturnInst::Create(this->mod()->getContext(),
      getZeroConst(this->mod()),
                           this->BasicBlock);
#endif

  this->leaveScope(false);

  if (ParamCount - 1)
    this->popFunc(); // pop inner.
  this->popFunc();   // pop outer.

  this->BasicBlock = OrigBB;
  this->TrueBlock = OrigTrueBB;
  this->FalseBlock = OrigFalseBB;

  this->LastExpr = PrevLast;

  auto Ptr = this->addAllocateCall(this->BasicBlock,
      getConst(this->mod(), 16));
  auto PtrAsInt = new llvm::PtrToIntInst(Ptr,
      getInteger64(this->mod()), "",
                                         this->BasicBlock);
  this->storeNextTemp(PtrAsInt);

  auto FunInt = new llvm::PtrToIntInst(OuterFun,
      getInteger64(this->mod()), "",
                                       this->BasicBlock);
#if 0
  auto Store = new llvm::StoreInst(FunInt, Ptr, false,
      this->BasicBlock);
  Store->setAlignment(alignment);
#else
  this->addGCStore(FunInt, PtrAsInt, PtrAsInt);
#endif

  llvm::SmallVector<llvm::Value *, 1> Indices;
  Indices.push_back(getConst(this->mod(), 1));
  auto Snd =
      llvm::GetElementPtrInst::Create(getInteger64(this->mod()),
      Ptr,
```

```
                                                    Indices, "",
                                                        this->BasicBlock);
#if 0
  llvm::Value *Context =
      new llvm::LoadInst(this->peekFunc()->getLocalPtr(), "",
          this->BasicBlock);
#else
  llvm::Value *Context = this->loadLocalPtr();
#endif
  Context = new llvm::PtrToIntInst(Context,
      getInteger64(this->mod()), "",
                                     this->BasicBlock);
#if 0
  Store = new llvm::StoreInst(Context, Snd, false,
      this->BasicBlock);
  Store->setAlignment(alignment);
#else
  this->addGCStore(Context, Snd, PtrAsInt);
#endif
  this->addLocalFuncVal(RealFunID, PtrAsInt);
}

void CodegenVisitor::visit(DatatypeBareInstanceDeclaration *N) {
  this->Instances.insert(*N->getID()->getID(),
      this->DatatypeInstanceValue,
                           false);
  this->Names.insert(*N->getID()->getID(),
      {this->peekFunc()->getNest(), -1});
}

void CodegenVisitor::visit(DatatypeTypedInstanceDeclaration *N) {
  this->Instances.insert(*N->getID()->getID(),
      this->DatatypeInstanceValue,
                           true);
  this->Names.insert(*N->getID()->getID(),
      {this->peekFunc()->getNest(), -1});
}

void CodegenVisitor::visit(BareDatatypeDeclaration *N) {
  this->DatatypeTypename = N->getID()->getID();
  this->DatatypeInstanceValue = 0;
  for (auto &&C : *N->getDef()) {
    this->doVisit(C);
    ++this->DatatypeInstanceValue;
  }
}

void CodegenVisitor::visit(TypedDatatypeDeclaration *N) {
  this->DatatypeTypename = N->getID()->getID();
  this->DatatypeInstanceValue = 0;
  for (auto &&C : *N->getDef()) {
    this->doVisit(C);
    ++this->DatatypeInstanceValue;
  }
}

void CodegenVisitor::visit(LongIdentifierType *N) {
  // Nop.
}

void CodegenVisitor::visit(VariableType *N) {
  // Nop.
```

```
}

void CodegenVisitor::visit(SeqType *N) {
  // Nop.
}

void CodegenVisitor::visit(SeqVariableType *N) {
  // Nop.
}

void CodegenVisitor::visit(TupleType *N) {
  // Nop.
}

void CodegenVisitor::visit(ApplyType *N) {
  // Nop.
}

void CodegenVisitor::visit(ProductType *N) {
  // Nop.
}

void CodegenVisitor::visit(FunctionType *N) {
  // Nop.
}
```

Listing 44: lib/Codegen/NameMap.cpp

```
#include "NameMap.h"

#include <cassert>

using namespace ssml::codegen;

void NameMap::insert(const std::string &K, ScopeValue V) {
  Maps.back()[K] = V;
}

NameMap::ScopeValue NameMap::get(const std::string &K) {
  for (auto It = Maps.rbegin(), End = Maps.rend(); It != End; ++It)
    if (It->count(K))
      return It->find(K)->second;
  return {-1, -1};
}

void NameMap::enterScope() {
  Maps.push_back(MapType());
}

void NameMap::leaveScope() {
  Maps.pop_back();
}
```

Listing 45: lib/Common/ErrorMessages.cpp

```
#include "ssml/Common/ErrorMessages.h"

#include "ssml/Common/Commandline.h"

#include "llvm/Support/raw_ostream.h"
#include "llvm/ADT/StringRef.h"
```

```
#include <cstdlib>

using namespace ssml;
using namespace llvm;

void ssml::errorExit(StringRef FileName, SourceLocation Loc,
                     llvm::StringRef Msg) {
  errs() << FileName << ':' << Loc << ": error: " << Msg << '\n';
  exit(1);
}

void ssml::errorExit(SourceLocation Loc, llvm::StringRef Msg) {
  return errorExit(*Commandline.getFilename(), Loc, Msg);
}
```

Listing 46: lib/Common/Compare.cpp

```
#include "ssml/Common/Compare.h"

#include <string>

namespace ssml {
template struct SharedPtrLess<std::string>;
} // End namespace ssml.
```

Listing 47: lib/Common/Commandline.cpp

```
#include "ssml/Common/Commandline.h"

using namespace ssml;

CommandlineImpl ssml::Commandline;

static std::shared_ptr<std::string> StdinFilename(new
    std::string("<stdin>"));

const std::shared_ptr<std::string> CommandlineImpl::getFilename()
    const {
  return StdinFilename;
}
```

Listing 48: lib/Common/ErrorExit.cpp

```
#include "ssml/Common/FatalExit.h"

#include "llvm/Support/raw_ostream.h"

#include <exception>

using namespace ssml;
using namespace llvm;

void ssml::fatalExit(Twine Msg) {
  errs() << "fatal error: " << Msg << '\n';
  std::terminate();
}

void ssml::fatalExitOutOfMem() {
```

```
  fatalExit("out of memory");
}
```

Listing 49: lib/Common/StringManipulation.cpp

```
#include "ssml/Common/StringManipulation.h"

#include "ssml/Common/FatalExit.h"

#include <cstring>

using namespace ssml;

char *ssml::duplicate(const char *str) {
  char *dup = strdup(str);
  if (!dup)
    fatalExitOutOfMem();
  return dup;
}
```

Listing 50: lib/Common/SourceLocation.cpp

```
#include "ssml/Common/SourceLocation.h"

#include "llvm/Support/raw_ostream.h"

#include <sstream>

using namespace llvm;
using namespace ssml;

std::string SourceLocation::toString() const {
  return std::to_string(this->Line) + ':'
      +std::to_string(this->Column);
}

raw_ostream &llvm::operator<<(raw_ostream &Out, SourceLocation L) {
  return Out << L.toString();
}
```

Listing 51: lib/AST/Type.cpp

```
#include "ssml/AST/Type.h"

#include "ssml/AST/Visitor.h"
#include "ssml/AST/Identifier.h"

#include <cassert>

using namespace ssml::ast;

Type::Type(SourceLocation L) : Node(L) {}

TupleType *Type::asTupleType() { return nullptr; }

LongIdentifierType::LongIdentifierType(SourceLocation L,
                                       std::shared_ptr<LongIdentifier>
                                           ID)
    : Type(L), ID(ID) {}
```

```

void LongIdentifierType::accept(Visitor *V) { V->visit(this); }

VariableType::VariableType(SourceLocation L,
                           std::shared_ptr<ShortIdentifier> ID)
    : Type(L), ID(ID) {}

void VariableType::accept(Visitor *V) { V->visit(this); }

SeqType::SeqType(SourceLocation L, std::shared_ptr<Type> F)
    : Type(L), std::vector<std::shared_ptr<Type>>{F} {}

void SeqType::accept(Visitor *V) { V->visit(this); }

SeqVariableType::SeqVariableType(SourceLocation L,
                                 std::shared_ptr<VariableType> F)
    : Type(L), std::vector<std::shared_ptr<VariableType>>{F} {}

void SeqVariableType::accept(Visitor *V) { V->visit(this); }

TupleType::TupleType(SourceLocation L, std::shared_ptr<SeqType>
    Types)
    : Type(L), Types(Types) {}

void TupleType::accept(Visitor *V) { V->visit(this); }

TupleType *TupleType::asTupleType() { return this; }

IdentifierColonType::IdentifierColonType(SourceLocation L,
                                         std::shared_ptr<Identifier>
                                             ID,
                                         std::shared_ptr<Type> T)
    : Type(L), Iden(ID), Typ(T) {}

void IdentifierColonType::accept(Visitor *V) { V->visit(this); }

RecordType::RecordType(SourceLocation L,
    std::shared_ptr<IdentifierColonType> F)
    : Type(L),
        std::vector<std::shared_ptr<IdentifierColonType>>{F} {}

void RecordType::accept(Visitor *V) { V->visit(this); }

ApplyType::ApplyType(SourceLocation L, std::shared_ptr<Type> F)
    : Type(L), std::vector<std::shared_ptr<Type>>{F} {}

void ApplyType::accept(Visitor *V) { V->visit(this); }

ProductType::ProductType(SourceLocation L, std::shared_ptr<Type> F)
    : Type(L), std::vector<std::shared_ptr<Type>>{F} {}

void ProductType::accept(Visitor *V) { V->visit(this); }

FunctionType::FunctionType(SourceLocation L, std::shared_ptr<Type>
    F)
    : Type(L), std::vector<std::shared_ptr<Type>>{F} {}

void FunctionType::accept(Visitor *V) { V->visit(this); }
```

Listing 52: lib/AST/Literal.cpp

```
#include "ssml/AST/Literal.h"
#include "ssml/AST/Visitor.h"

#include "ssml/Common/ErrorMessages.h"
#include "ssml/Common/FatalExit.h"

#include "llvm/ADT/StringRef.h"

#include <cstdlib>
#include <cassert>
#include <cerrno>

using namespace ssml;
using namespace ssml::ast;

Literal::Literal(SourceLocation L, std::shared_ptr<std::string> Value)
    : Node(L), Value(Value) {}

IntLiteral::IntLiteral(SourceLocation L, std::shared_ptr<std::string> Value)
    : Literal(L, Value), IntValue(0) {
  const char *RawVal = Value->c_str();
  size_t Len = Value->size();
  int Base;
  if (Len > 2 && RawVal[1] == 'x')
    Base = 16;
  else
    Base = 10;

  errno = 0;
  long long V = std::strtoll(RawVal, nullptr, Base);
  if (errno == ERANGE) {
    errorExit(this->getLocation(), "int literal out of bounds: " + *Value);
  } else {
    assert(!errno);
    this->IntValue = V;
  }
}

void IntLiteral::accept(Visitor *V) { V->visit(this); }

std::int64_t IntLiteral::getIntValue() const { return this->IntValue; }

int64_t IntLiteral::toInt() { return this->getIntValue(); }

RealLiteral::RealLiteral(SourceLocation L, std::shared_ptr<std::string> Value)
    : Literal(L, Value) {}

void RealLiteral::accept(Visitor *V) { V->visit(this); }

int64_t RealLiteral::toInt() {
  fatalExit("cannot convert real to int");
  return 0;
}

CharLiteral::CharLiteral(SourceLocation L, std::shared_ptr<std::string> Value)
    : Literal(L, Value) {}
```

```
void CharLiteral::accept(Visitor *V) { V->visit(this); }

int64_t CharLiteral::toInt() {
  fatalExit("cannot convert char to int");
  return 0;
}

StringLiteral::StringLiteral(SourceLocation L,
                             std::shared_ptr<std::string> Value)
    : Literal(L, Value) {}

void StringLiteral::accept(Visitor *V) { V->visit(this); }

int64_t StringLiteral::toInt() {
  fatalExit("cannot convert string to int");
  return 0;
}
```

Listing 53: lib/AST/Identifier.cpp

```
#include "ssml/AST/Identifier.h"

#include "ssml/AST/Visitor.h"
#include "ssml/AST/Literal.h"

#include <sstream>
#include <cassert>

using namespace ssml::ast;

Identifier::Identifier(SourceLocation L) : Node(L) {}

ShortIdentifier::ShortIdentifier(SourceLocation L,
                                 std::shared_ptr<std::string> ID)
    : Identifier(L), ID(ID) {}

void ShortIdentifier::accept(Visitor *V) { V->visit(this); }

std::string ShortIdentifier::toString() { return *this->getID(); }

LongIdentifier::LongIdentifier(SourceLocation L,
                               std::shared_ptr<ShortIdentifier>
                                   First)
    : Identifier(L),
        std::vector<std::shared_ptr<ShortIdentifier>>{First} {}

void LongIdentifier::accept(Visitor *V) { V->visit(this); }

std::string LongIdentifier::toString() {
  std::stringstream S;

  auto Stop = this->size();
  assert(Stop);
  --Stop;

  size_t I;
  for (I = 0; I < Stop; ++I)
    S << *(*this)[I]->getID() << '.';
  S << *(*this)[I]->getID();

```

```
  return S.str();
}

IntIdentifier::IntIdentifier(SourceLocation L,
    std::shared_ptr<IntLiteral> Lit)
    : Identifier(L), Literal(Lit) {}

void IntIdentifier::accept(Visitor *V) { V->visit(this); }

std::string IntIdentifier::toString() {
  return *this->getLiteral()->getValueString();
}

SeqShortIdentifier::SeqShortIdentifier(SourceLocation L,
                                        std::shared_ptr<ShortIdentifier>
                                            First)
    : Identifier(L),
        std::vector<std::shared_ptr<ShortIdentifier>>{First} {}

void SeqShortIdentifier::accept(Visitor *V) { V->visit(this); }

std::string SeqShortIdentifier::toString() {
  std::stringstream S;

  auto Stop = this->size();
  assert(Stop);
  --Stop;

  size_t I;
  for (I = 0; I < Stop; ++I)
    S << *(*this)[I]->getID() << ' ';
  S << *(*this)[I]->getID();

  return S.str();
}

SeqLongIdentifier::SeqLongIdentifier(SourceLocation L,
                                      std::shared_ptr<LongIdentifier>
                                          First)
    : Identifier(L),
        std::vector<std::shared_ptr<LongIdentifier>>{First} {}

void SeqLongIdentifier::accept(Visitor *V) { V->visit(this); }

std::string SeqLongIdentifier::toString() {
  std::stringstream S;

  auto Stop = this->size();
  assert(Stop);
  --Stop;

  size_t I;
  for (I = 0; I < Stop; ++I)
    S << (*this)[I]->toString() << ' ';
  S << (*this)[I]->toString();

  return S.str();
}
```

Listing 54: lib/AST/Match.cpp

```
#include "ssml/AST/Match.h"

#include "ssml/AST/Pattern.h"
#include "ssml/AST/Expression.h"
#include "ssml/AST/Visitor.h"

using namespace ssml::ast;

Match::Match(SourceLocation L, std::shared_ptr<Pattern> P,
    std::shared_ptr<Expression> E)
    : Node(L), Pat(P), Expr(E) {}

void Match::accept(Visitor *V) { V->visit(this); }

SeqMatch::SeqMatch(SourceLocation L, std::shared_ptr<Match> F)
    : Node(L), std::vector<std::shared_ptr<Match> >{F} {}

void SeqMatch::accept(Visitor *V) { V->visit(this); }
```

Listing 55: lib/AST/Visitor.cpp

```
#include "ssml/AST/Visitor.h"

#include "ssml/Common/FatalExit.h"

using namespace ssml;
using namespace ssml::ast;

Visitor::~Visitor() {}

void Visitor::visit(RealLiteral *) {
  fatalExit("unimplemented feature");
}

void Visitor::visit(CharLiteral *) {
  fatalExit("unimplemented feature");
}

void Visitor::visit(StringLiteral *) {
  fatalExit("unimplemented feature");
}

void Visitor::visit(IntIdentifier *) {
  fatalExit("unimplemented feature");
}

void Visitor::visit(IdentifierEqualsExpression *) {
  fatalExit("unimplemented feature");
}

void Visitor::visit(RecordExpression *) {
  fatalExit("unimplemented feature");
}

void Visitor::visit(SelectorExpression *) {
  fatalExit("unimplemented feature");
}

void Visitor::visit(TypeExpression *) {
  fatalExit("unimplemented feature");
}
```

```
void Visitor::visit(CaseExpression *) {
  fatalExit("unimplemented feature");
}

void Visitor::visit(ShortIdentifierPattern *) {
  fatalExit("unimplemented feature");
}

void Visitor::visit(AsPattern *) {
  fatalExit("unimplemented feature");
}

void Visitor::visit(IdentifierEqualsPattern *) {
  fatalExit("unimplemented feature");
}

void Visitor::visit(RecordPattern *) {
  fatalExit("unimplemented feature");
}

void Visitor::visit(OpenDeclaration *) {
  fatalExit("unimplemented feature");
}

void Visitor::visit(RestrictedStructDeclaration *) {
  fatalExit("unimplemented feature");
}

void Visitor::visit(BareStructDeclaration *) {
  fatalExit("unimplemented feature");
}

void Visitor::visit(AbstractValDeclaration *) {
  fatalExit("unimplemented feature");
}

void Visitor::visit(AbstractTypeDeclaration *) {
  fatalExit("unimplemented feature");
}

void Visitor::visit(AbstractStructDeclaration *) {
  fatalExit("unimplemented feature");
}

void Visitor::visit(SharingTypeDeclaration *) {
  fatalExit("unimplemented feature");
}

void Visitor::visit(SigDeclaration *) {
  fatalExit("unimplemented feature");
}

void Visitor::visit(TypeDeclaration *) {
  fatalExit("unimplemented feature");
}

void Visitor::visit(ShortBareFunctorDeclaration *) {
  fatalExit("unimplemented feature");
}

void Visitor::visit(ShortRestrictedFunctorDeclaration *) {
```

```
  fatalExit("unimplemented feature");
}

void Visitor::visit(LongBareFunctorDeclaration *) {
  fatalExit("unimplemented feature");
}

void Visitor::visit(LongRestrictedFunctorDeclaration *) {
  fatalExit("unimplemented feature");
}

void Visitor::visit(LongIdentifierDefinition *) {
  fatalExit("unimplemented feature");
}

void Visitor::visit(StructDefinition *) {
  fatalExit("unimplemented feature");
}

void Visitor::visit(AnnotationDefinition *) {
  fatalExit("unimplemented feature");
}

void Visitor::visit(ShortFunctorDefinition *) {
  fatalExit("unimplemented feature");
}

void Visitor::visit(LongFunctorDefinition *) {
  fatalExit("unimplemented feature");
}

void Visitor::visit(SigDefinition *) {
  fatalExit("unimplemented feature");
}

void Visitor::visit(IdentifierColonType *) {
  fatalExit("unimplemented feature");
}

void Visitor::visit(RecordType *) {
  fatalExit("unimplemented feature");
}
```

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

```
#include "ssml/AST/Fixup/Fixup.h"

#include "FixupVisitor.h"

#include "ssml/AST/Declaration.h"
#include "ssml/AST/Visitor.h"

using namespace ssml::ast::fix;

void ssml::ast::fixup(Root *R) {
  FixupVisitor V;
  V.visit(R);
}
```

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

```cpp
#include "OperatorDefinition.h"

ssml::ast::fix::OperatorDefinition::OperatorDefinition(
    std::shared_ptr<const std::string> ID, Associativity A,
        Precedence P)
    : Assoc(A), Prec(P) {}
```

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

```cpp
#include "OperatorDefVisitor.h"

#include "OperatorDefMap.h"

#include "ssml/AST/Node.h"
#include "ssml/AST/Declaration.h"
#include "ssml/AST/Identifier.h"
#include "ssml/AST/Literal.h"
#include "ssml/AST/Match.h"
#include "ssml/AST/Pattern.h"
#include "ssml/AST/Expression.h"
#include "ssml/AST/Type.h"
#include "ssml/AST/Definition.h"

#include "ssml/Common/FatalExit.h"
#include "ssml/Common/ErrorMessages.h"

using namespace ssml::ast;
using namespace ssml::ast::fix;

OperatorDefVisitor::OperatorDefVisitor(OperatorDefMap &M) :
    OpDefs(M) {}

OperatorDefinition
OperatorDefVisitor::getOperatorDefinition(Node &N, OperatorDefMap
    &M) {
  OperatorDefVisitor V(M);
  N.accept(&V);
  return V.OpDef;
}

OperatorDefinition
OperatorDefVisitor::getShortIdentifierOpDef(ShortIdentifier &ID) {
  return this->OpDefs.get(ID.getID());
}

OperatorDefinition OperatorDefVisitor::getLongIdentifierOpDef(
    LongIdentifier &IDs, SourceLocation L, bool opKeyPrefixed) {
  auto S = IDs.size();
  assert(S >= 1);

  if (opKeyPrefixed) {
#if 0
    if (S > 1) {
      errorExit(L, "invalid 'op' keyword before long identifier");
    } else {
      return OperatorDefinition();
    }
#endif
    return OperatorDefinition();
  } else if (S > 1) {
```

```
    return OperatorDefinition();
  } else {
    return this->OpDefs.get(IDs.back()->getID());
  }
  return OperatorDefinition();
}

void OperatorDefVisitor::visit(IntLiteral *N) { fatalExit("internal error"); }

void OperatorDefVisitor::visit(Match *N) { fatalExit("internal error"); }

void OperatorDefVisitor::visit(SeqMatch *N) { fatalExit("internal error"); }

void OperatorDefVisitor::visit(ShortIdentifier *N) {
  fatalExit("internal error");
}

void OperatorDefVisitor::visit(LongIdentifier *N) {
  fatalExit("internal error");
}

void OperatorDefVisitor::visit(IntIdentifier *N) {
  fatalExit("internal error");
}

void OperatorDefVisitor::visit(SeqShortIdentifier *N) {
  fatalExit("internal error");
}

void OperatorDefVisitor::visit(SeqLongIdentifier *N) {
  fatalExit("internal error");
}

void OperatorDefVisitor::visit(SeqExpression *N) {
  for (auto C : *N)
    C->accept(this);
}

void OperatorDefVisitor::visit(LiteralExpression *N) {
  this->OpDef = OperatorDefinition();
}

void OperatorDefVisitor::visit(LongIdentifierExpression *N) {
  this->OpDef = this->getLongIdentifierOpDef(*N->getIDs(), N->getLocation(),
                                              N->isPrefixedWithOpKey());
}

void OperatorDefVisitor::visit(TupleExpression *N) {
  this->OpDef = OperatorDefinition();
}

void OperatorDefVisitor::visit(ListExpression *N) {
  this->OpDef = OperatorDefinition();
}

void OperatorDefVisitor::visit(ApplyExpression *N) {
  this->OpDef = OperatorDefinition();
}
```

```

void OperatorDefVisitor::visit(OrElseExpression *N) {
  this->OpDef = OperatorDefinition();
}

void OperatorDefVisitor::visit(AndAlsoExpression *N) {
  this->OpDef = OperatorDefinition();
}

void OperatorDefVisitor::visit(LetExpression *N) {
  this->OpDef = OperatorDefinition();
}

void OperatorDefVisitor::visit(IfExpression *N) {
  this->OpDef = OperatorDefinition();
}

void OperatorDefVisitor::visit(WhileExpression *N) {
  this->OpDef = OperatorDefinition();
}

void OperatorDefVisitor::visit(CaseExpression *N) {
  this->OpDef = OperatorDefinition();
}

void OperatorDefVisitor::visit(LambdaExpression *N) {
  this->OpDef = OperatorDefinition();
}

void OperatorDefVisitor::visit(SeqPattern *N) {
  assert(N->size() == 1);
  N->back()->accept(this);
}

void OperatorDefVisitor::visit(LiteralPattern *N) {
  this->OpDef = OperatorDefinition();
}

void OperatorDefVisitor::visit(WildcardPattern *N) {
  this->OpDef = OperatorDefinition();
}

void OperatorDefVisitor::visit(ShortIdentifierPattern *N) {
  this->OpDef = this->getShortIdentifierOpDef(*N->getID());
}

void OperatorDefVisitor::visit(LongIdentifierPattern *N) {
  this->OpDef = this->getLongIdentifierOpDef(*N->getIDs(),
      N->getLocation(),
                                                N->isPrefixedWithOpKey());
}

void OperatorDefVisitor::visit(TypePattern *N) {
  this->OpDef = OperatorDefinition();
}

void OperatorDefVisitor::visit(ApplyPattern *N) {
  this->OpDef = OperatorDefinition();
}

void OperatorDefVisitor::visit(ListPattern *N) {
  this->OpDef = OperatorDefinition();
```

```
}

void OperatorDefVisitor::visit(TuplePattern *N) {
  this->OpDef = OperatorDefinition();
}

void OperatorDefVisitor::visit(SeqDeclaration *N) {
  fatalExit("internal error");
}

void OperatorDefVisitor::visit(Root *N) { fatalExit("internal
    error"); }

void OperatorDefVisitor::visit(ValDeclaration *N) {
  fatalExit("internal error");
}

void OperatorDefVisitor::visit(OpenDeclaration *N) {
  fatalExit("internal error");
}

void OperatorDefVisitor::visit(NonfixDeclaration *N) {
  fatalExit("internal error");
}

void OperatorDefVisitor::visit(InfixDeclaration *N) {
  fatalExit("internal error");
}

void OperatorDefVisitor::visit(InfixRDeclaration *N) {
  fatalExit("internal error");
}

void OperatorDefVisitor::visit(BareStructDeclaration *N) {
  fatalExit("internal error");
}

void OperatorDefVisitor::visit(RestrictedStructDeclaration *N) {
  fatalExit("internal error");
}

void OperatorDefVisitor::visit(AbstractValDeclaration *N) {
  fatalExit("internal error");
}

void OperatorDefVisitor::visit(AbstractTypeDeclaration *N) {
  fatalExit("internal error");
}

void OperatorDefVisitor::visit(AbstractStructDeclaration *N) {
  fatalExit("internal error");
}

void OperatorDefVisitor::visit(SigDeclaration *N) {
  fatalExit("internal error");
}

void OperatorDefVisitor::visit(FunPatternDeclaration *N) {
  fatalExit("internal error");
}

void OperatorDefVisitor::visit(FunDeclaration *N) {
```

```
  fatalExit("internal error");
}

void OperatorDefVisitor::visit(TypeDeclaration *N) {
  fatalExit("internal error");
}

void OperatorDefVisitor::visit(DatatypeBareInstanceDeclaration *N)
    {
  fatalExit("internal error");
}

void OperatorDefVisitor::visit(DatatypeTypedInstanceDeclaration
    *N) {
  fatalExit("internal error");
}

void OperatorDefVisitor::visit(BareDatatypeDeclaration *N) {
  fatalExit("internal error");
}

void OperatorDefVisitor::visit(TypedDatatypeDeclaration *N) {
  fatalExit("internal error");
}

void OperatorDefVisitor::visit(ShortBareFunctorDeclaration *N) {
  fatalExit("internal error");
}

void OperatorDefVisitor::visit(ShortRestrictedFunctorDeclaration
    *N) {
  fatalExit("internal error");
}

void OperatorDefVisitor::visit(LongBareFunctorDeclaration *N) {
  fatalExit("internal error");
}

void OperatorDefVisitor::visit(LongRestrictedFunctorDeclaration
    *N) {
  fatalExit("internal error");
}

void OperatorDefVisitor::visit(LongIdentifierDefinition *N) {
  fatalExit("internal error");
}

void OperatorDefVisitor::visit(StructDefinition *N) {
  fatalExit("internal error");
}

void OperatorDefVisitor::visit(AnnotationDefinition *N) {
  fatalExit("internal error");
}

void OperatorDefVisitor::visit(ShortFunctorDefinition *N) {
  fatalExit("internal error");
}

void OperatorDefVisitor::visit(LongFunctorDefinition *N) {
  fatalExit("internal error");
}
```

```
void OperatorDefVisitor::visit(SigDefinition *N) {
  fatalExit("internal error");
}

void OperatorDefVisitor::visit(LongIdentifierType *N) {
  fatalExit("internal error");
}

void OperatorDefVisitor::visit(VariableType *N) {
    fatalExit("internal error"); }

void OperatorDefVisitor::visit(SeqType *N) { fatalExit("internal
    error"); }

void OperatorDefVisitor::visit(SeqVariableType *N) {
  fatalExit("internal error");
}

void OperatorDefVisitor::visit(TupleType *N) { fatalExit("internal
    error"); }

void OperatorDefVisitor::visit(ApplyType *N) { fatalExit("internal
    error"); }

void OperatorDefVisitor::visit(ProductType *N) {
    fatalExit("internal error"); }

void OperatorDefVisitor::visit(FunctionType *N) {
    fatalExit("internal error"); }
```

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

```
#include "OperatorDefMap.h"

using namespace ssml::ast::fix;

OperatorDefinition OperatorDefMap::get(std::shared_ptr<const
    std::string> K) {
  for (auto It = Maps.rbegin(), End = Maps.rend(); It != End;
      ++It) {
    auto &M = *It;
    if (M.count(K))
      return M[K];
  }
  return Maps.front()[K];
}

void OperatorDefMap::put(std::shared_ptr<const std::string> K,
                         OperatorDefinition Def) {
  Maps.back()[K] = Def;
}

void OperatorDefMap::enterScope() {
  this->Maps.push_back(MapType());
}

void OperatorDefMap::leaveScope() {
  this->Maps.pop_back();
}
```

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

```cpp
#define DEBUG_TYPE "OperatorFixupParser"

#include "FixupVisitor.h"

#include "OperatorFixupParser.h"
#include "OperatorDefVisitor.h"

#include "ssml/AST/Node.h"
#include "ssml/AST/Declaration.h"
#include "ssml/AST/Identifier.h"
#include "ssml/AST/Literal.h"
#include "ssml/AST/Match.h"
#include "ssml/AST/Pattern.h"
#include "ssml/AST/Expression.h"
#include "ssml/AST/Type.h"
#include "ssml/AST/Definition.h"

#include "ssml/Common/ErrorMessages.h"

#include "llvm/Support/raw_ostream.h"
#include "llvm/Support/Debug.h"

using namespace ssml::ast;
using namespace ssml::ast::fix;

template <typename VectorT, typename ItemT, typename TupleT, typename SeqT>
std::shared_ptr<VectorT> FixupVisitor::operatorFixup(VectorT &Initial) {
  OperatorFixupParser<VectorT, ItemT, TupleT, SeqT> Parser(Initial,
                                                            this->OpDefs);
  return Parser.parse();
}

void FixupVisitor::visit(IntLiteral *N) {}

void FixupVisitor::visit(Match *N) {
  N->getPattern()->accept(this);
  N->getExpr()->accept(this);
}

void FixupVisitor::visit(SeqMatch *N) {
  for (auto C : *N)
    C->accept(this);
}

void FixupVisitor::visit(ShortIdentifier *N) {}

void FixupVisitor::visit(LongIdentifier *N) {}

void FixupVisitor::visit(IntIdentifier *N) {}

void FixupVisitor::visit(SeqShortIdentifier *N) {}

void FixupVisitor::visit(SeqLongIdentifier *N) {}

void FixupVisitor::visit(SeqExpression *N) {
  for (auto C : *N)
    C->accept(this);
}

```

```
void FixupVisitor::visit(LiteralExpression *N) {
  this->CurrOpDef = OperatorDefinition();
}

void FixupVisitor::visit(LongIdentifierExpression *N) {
  this->CurrOpDef = OperatorDefVisitor::getOperatorDefinition(*N,
      this->OpDefs);
}

void FixupVisitor::visit(TupleExpression *N) {
  N->getExprs()->accept(this);
  this->CurrOpDef = OperatorDefinition();
}

void FixupVisitor::visit(ListExpression *N) {
  N->getExprs()->accept(this);
  this->CurrOpDef = OperatorDefinition();
}

void FixupVisitor::visit(ApplyExpression *N) {
  DEBUG(llvm::errs() << "Fixup ApplyExpression\n");
  for (auto C : *N)
    C->accept(this);
  auto V = this->operatorFixup<ApplyExpression, Expression,
      TupleExpression,
                               SeqExpression>(*N);
  N->clear();
  N->insert(N->begin(), V->begin(), V->end());
  this->CurrOpDef = OperatorDefinition();
}

void FixupVisitor::visit(OrElseExpression *N) {
  N->getLeftExpr()->accept(this);
  N->getRightExpr()->accept(this);
  this->CurrOpDef = OperatorDefinition();
}

void FixupVisitor::visit(AndAlsoExpression *N) {
  N->getLeftExpr()->accept(this);
  N->getRightExpr()->accept(this);
  this->CurrOpDef = OperatorDefinition();
}

void FixupVisitor::visit(LetExpression *N) {
  this->OpDefs.enterScope();
  N->getDecls()->accept(this);
  N->getExprs()->accept(this);
  this->CurrOpDef = OperatorDefinition();
  this->OpDefs.leaveScope();
}

void FixupVisitor::visit(IfExpression *N) {
  N->getCondExpr()->accept(this);
  N->getThenExpr()->accept(this);
  N->getElseExpr()->accept(this);
  this->CurrOpDef = OperatorDefinition();
}

void FixupVisitor::visit(WhileExpression *N) {
  N->getCondExpr()->accept(this);
  N->getBodyExpr()->accept(this);
  this->CurrOpDef = OperatorDefinition();
```

```
}

void FixupVisitor::visit(CaseExpression *N) {
  N->getExpr()->accept(this);
  N->getCases()->accept(this);
  this->CurrOpDef = OperatorDefinition();
}

void FixupVisitor::visit(LambdaExpression *N) {
  N->getCases()->accept(this);
  this->CurrOpDef = OperatorDefinition();
}

void FixupVisitor::visit(SeqPattern *N) {
  for (auto C : *N)
    C->accept(this);
}

void FixupVisitor::visit(LiteralPattern *N) {
  this->CurrOpDef = OperatorDefinition();
}

void FixupVisitor::visit(WildcardPattern *N) {
  this->CurrOpDef = OperatorDefinition();
}

void FixupVisitor::visit(ShortIdentifierPattern *N) {
  this->CurrOpDef = OperatorDefVisitor::getOperatorDefinition(*N,
      this->OpDefs);
}

void FixupVisitor::visit(LongIdentifierPattern *N) {
  this->CurrOpDef = OperatorDefVisitor::getOperatorDefinition(*N,
      this->OpDefs);
}

void FixupVisitor::visit(TypePattern *N) {
  N->getPattern()->accept(this);
  N->getType()->accept(this);
  this->CurrOpDef = OperatorDefinition();
}

void FixupVisitor::visit(ApplyPattern *N) {
  DEBUG(llvm::errs() << "Fixup ApplyPattern\n");
  for (auto C : *N)
    C->accept(this);
  auto V =
      this->operatorFixup<ApplyPattern, Pattern, TuplePattern,
          SeqPattern>(*N);
  N->clear();
  N->insert(N->begin(), V->begin(), V->end());
  this->CurrOpDef = OperatorDefinition();
}

void FixupVisitor::visit(ListPattern *N) {
  N->getPatterns()->accept(this);
  this->CurrOpDef = OperatorDefinition();
}

void FixupVisitor::visit(TuplePattern *N) {
  N->getPatterns()->accept(this);
  this->CurrOpDef = OperatorDefinition();
```

```
}

void FixupVisitor::visit(SeqDeclaration *N) {
  for (auto C : *N)
    C->accept(this);
}

void FixupVisitor::visit(Root *N) {
  this->OpDefs.enterScope();

  std::shared_ptr<ShortIdentifier> Cons =
      std::shared_ptr<ShortIdentifier>(new ShortIdentifier(
          SourceLocation(0, 0),
              std::make_shared<std::string>("::")));

  std::shared_ptr<ShortIdentifier> Plus =
      std::shared_ptr<ShortIdentifier>(new ShortIdentifier(
          SourceLocation(0, 0),
              std::make_shared<std::string>("+")));

  std::shared_ptr<ShortIdentifier> Minus =
      std::shared_ptr<ShortIdentifier>(new ShortIdentifier(
          SourceLocation(0, 0),
              std::make_shared<std::string>("-")));

  std::shared_ptr<ShortIdentifier> Multiply =
      std::shared_ptr<ShortIdentifier>(new ShortIdentifier(
          SourceLocation(0, 0),
              std::make_shared<std::string>("*")));

  std::shared_ptr<ShortIdentifier> Division =
      std::shared_ptr<ShortIdentifier>(new ShortIdentifier(
          SourceLocation(0, 0),
              std::make_shared<std::string>("div")));

  std::shared_ptr<ShortIdentifier> Modulo =
      std::shared_ptr<ShortIdentifier>(new ShortIdentifier(
          SourceLocation(0, 0),
              std::make_shared<std::string>("mod")));

  std::shared_ptr<ShortIdentifier> Refassign =
      std::shared_ptr<ShortIdentifier>(new ShortIdentifier(
          SourceLocation(0, 0),
              std::make_shared<std::string>(":=")));

  std::shared_ptr<ShortIdentifier> Equals =
      std::shared_ptr<ShortIdentifier>(new ShortIdentifier(
          SourceLocation(0, 0),
              std::make_shared<std::string>("=")));

  std::shared_ptr<ShortIdentifier> NotEquals =
      std::shared_ptr<ShortIdentifier>(new ShortIdentifier(
          SourceLocation(0, 0),
              std::make_shared<std::string>("<>")));

  std::shared_ptr<ShortIdentifier> Less =
      std::shared_ptr<ShortIdentifier>(new ShortIdentifier(
          SourceLocation(0, 0),
              std::make_shared<std::string>("<")));

  std::shared_ptr<ShortIdentifier> Greater =
      std::shared_ptr<ShortIdentifier>(new ShortIdentifier(
```

```
        SourceLocation(0, 0),
            std::make_shared<std::string>(">")));

  std::shared_ptr<ShortIdentifier> LessEquals =
      std::shared_ptr<ShortIdentifier>(new ShortIdentifier(
          SourceLocation(0, 0),
              std::make_shared<std::string>("<=")));

  std::shared_ptr<ShortIdentifier> GreaterEquals =
      std::shared_ptr<ShortIdentifier>(new ShortIdentifier(
          SourceLocation(0, 0),
              std::make_shared<std::string>(">=")));

  SeqShortIdentifier Infixr3Seq(SourceLocation(0, 0), Refassign);
  IntLiteral Infixr3Lit(SourceLocation(0, 0),
                        std::make_shared<std::string>("3"));

  SeqShortIdentifier Infix4Seq(SourceLocation(0, 0), Equals);
  Infix4Seq.push_back(NotEquals);
  Infix4Seq.push_back(Less);
  Infix4Seq.push_back(Greater);
  Infix4Seq.push_back(LessEquals);
  Infix4Seq.push_back(GreaterEquals);
  IntLiteral Infix4Lit(SourceLocation(0, 0),
                       std::make_shared<std::string>("4"));

  SeqShortIdentifier Infixr5Seq(SourceLocation(0, 0), Cons);
  IntLiteral Infixr5Lit(SourceLocation(0, 0),
                        std::make_shared<std::string>("5"));

  SeqShortIdentifier Infix6Seq(SourceLocation(0, 0), Plus);
  Infix6Seq.push_back(Minus);
  IntLiteral Infix6Lit(SourceLocation(0, 0),
                       std::make_shared<std::string>("6"));

  SeqShortIdentifier Infix7Seq(SourceLocation(0, 0), Multiply);
  Infix7Seq.push_back(Division);
  Infix7Seq.push_back(Modulo);
  IntLiteral Infix7Lit(SourceLocation(0, 0),
                       std::make_shared<std::string>("7"));

  setOpDef(Infixr3Seq, OperatorDefinition::INFIXR, Infixr3Lit);
  setOpDef(Infix4Seq, OperatorDefinition::INFIX, Infix4Lit);
  setOpDef(Infixr5Seq, OperatorDefinition::INFIXR, Infixr5Lit);
  setOpDef(Infix6Seq, OperatorDefinition::INFIX, Infix6Lit);
  setOpDef(Infix7Seq, OperatorDefinition::INFIX, Infix7Lit);

  this->OpDefs.enterScope();
  for (auto C : *N)
    C->accept(this);
  this->OpDefs.leaveScope();
  this->OpDefs.leaveScope();
}

void FixupVisitor::visit(ValDeclaration *N) {
  N->getDest()->accept(this);
  N->getSource()->accept(this);
}

void FixupVisitor::setNonfixOpDef(SeqShortIdentifier &IDs) {
  for (auto ID : IDs)
    this->OpDefs.put(ID->getID(), OperatorDefinition());
```

```
}

void FixupVisitor::setOpDef(SeqShortIdentifier &IDs,
                            OperatorDefinition::Associativity A,
                            IntLiteral &Prec) {
  auto P = Prec.getIntValue();
  if (P < OperatorDefinition::minPrecedence() ||
      P > OperatorDefinition::maxPrecedence()) {
    errorExit(Prec.getLocation(),
              "invalid operator precedence: " +
                  *Prec.getValueString());
    P = 0;
  }
  for (auto ID : IDs)
    this->OpDefs.put(ID->getID(), OperatorDefinition(ID->getID(),
        A, P));
}

void FixupVisitor::visit(NonfixDeclaration *N) {
  DEBUG(llvm::errs() << "Nonfix declaration\n");
  this->setNonfixOpDef(*N->getIDs());
}

void FixupVisitor::visit(InfixDeclaration *N) {
  DEBUG(llvm::errs() << "Infix declaration\n");
  setOpDef(*N->getIDs(), OperatorDefinition::INFIX,
      *N->getPrecedence());
}

void FixupVisitor::visit(InfixRDeclaration *N) {
  DEBUG(llvm::errs() << "Infixr declaration\n");
  setOpDef(*N->getIDs(), OperatorDefinition::INFIXR,
      *N->getPrecedence());
}

void FixupVisitor::visit(BareStructDeclaration *N) {
  N->getID()->accept(this);
  N->getStructDef()->accept(this);
}

void FixupVisitor::visit(RestrictedStructDeclaration *N) {
  N->getID()->accept(this);
  N->getSigDef()->accept(this);
  N->getStructDef()->accept(this);
}

void FixupVisitor::visit(AbstractValDeclaration *N) {
  N->getID()->accept(this);
  N->getType()->accept(this);
}

void FixupVisitor::visit(AbstractTypeDeclaration *N) {
  N->getID()->accept(this);
}

void FixupVisitor::visit(AbstractStructDeclaration *N) {
  N->getID()->accept(this);
  N->getSigDef()->accept(this);
}

void FixupVisitor::visit(SigDeclaration *N) {
  N->getID()->accept(this);
```

```
  N->getSigDef()->accept(this);
}

void FixupVisitor::visit(FunPatternDeclaration *N) {
  N->getIDParams()->accept(this);
  N->getDef()->accept(this);
}

void FixupVisitor::visit(FunDeclaration *N) {
  for (auto C : *N)
    C->accept(this);
}

void FixupVisitor::visit(TypeDeclaration *N) {
  N->getID()->accept(this);
  N->getType()->accept(this);
}

void FixupVisitor::visit(DatatypeBareInstanceDeclaration *N) {
  N->getID()->accept(this);
}

void FixupVisitor::visit(DatatypeTypedInstanceDeclaration *N) {
  N->getID()->accept(this);
  N->getType()->accept(this);
}

void FixupVisitor::visit(BareDatatypeDeclaration *N) {
  N->getID()->accept(this);
  N->getDef()->accept(this);
}

void FixupVisitor::visit(TypedDatatypeDeclaration *N) {
  N->getTypeParams()->accept(this);
  N->getID()->accept(this);
  N->getDef()->accept(this);
}

void FixupVisitor::visit(ShortBareFunctorDeclaration *N) {
  N->getFunctorID()->accept(this);
  N->getParamID()->accept(this);
  N->getParamSigDef()->accept(this);
  N->getFunctorStructDef()->accept(this);
}

void FixupVisitor::visit(ShortRestrictedFunctorDeclaration *N) {
  N->getFunctorID()->accept(this);
  N->getParamID()->accept(this);
  N->getParamSigDef()->accept(this);
  N->getFunctorSigDef()->accept(this);
  N->getFunctorStructDef()->accept(this);
}

void FixupVisitor::visit(LongBareFunctorDeclaration *N) {
  N->getFunctorID()->accept(this);
  N->getParams()->accept(this);
  N->getFunctorStructDef()->accept(this);
}

void FixupVisitor::visit(LongRestrictedFunctorDeclaration *N) {
  N->getFunctorID()->accept(this);
  N->getParams()->accept(this);
```

```
  N->getFunctorSigDef()->accept(this);
  N->getFunctorStructDef()->accept(this);
}

void FixupVisitor::visit(LongIdentifierDefinition *N) {}

void FixupVisitor::visit(StructDefinition *N) {
  this->OpDefs.enterScope();
  N->getDecls()->accept(this);
  this->OpDefs.leaveScope();
}

void FixupVisitor::visit(AnnotationDefinition *N) {
  N->getStructDef()->accept(this);
  N->getSigDef()->accept(this);
}

void FixupVisitor::visit(ShortFunctorDefinition *N) {
  N->getFunctorID()->accept(this);
  N->getStructDef()->accept(this);
}

void FixupVisitor::visit(LongFunctorDefinition *N) {
  N->getFunctorID()->accept(this);
  N->getDecls()->accept(this);
}

void FixupVisitor::visit(SigDefinition *N) {}

void FixupVisitor::visit(LongIdentifierType *N) {}

void FixupVisitor::visit(VariableType *N) {}

void FixupVisitor::visit(SeqType *N) {}

void FixupVisitor::visit(SeqVariableType *N) {}

void FixupVisitor::visit(TupleType *N) {}

void FixupVisitor::visit(ApplyType *N) {}

void FixupVisitor::visit(ProductType *N) {}

void FixupVisitor::visit(FunctionType *N) {
  if (N->size() > 2) {
    std::vector<std::shared_ptr<Type>> Vec = *N;
    N->clear();
    FunctionType *Fun = N;
    Fun->push_back(Vec[0]);
    Vec.erase(Vec.begin());
    while (Vec.size() >= 2) {
      auto Next = std::shared_ptr<FunctionType>(
          new FunctionType(Vec[0]->getLocation(), Vec[0]));
      Fun->push_back(Next);
      Fun = Next.get();
      Vec.erase(Vec.begin());
    }
    assert(Vec.size() == 1);
    Fun->push_back(Vec[0]);
  }
}
```

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

```
#define DEBUG_TYPE "OperatorFixupParser"

#include "OperatorFixupParser.h"
#include "OperatorDefVisitor.h"
#include "OperatorDefMap.h"

#include "ssml/AST/Node.h"
#include "ssml/AST/Expression.h"
#include "ssml/AST/Pattern.h"

#include "ssml/Common/ErrorMessages.h"
#include "ssml/Common/SourceLocation.h"

#include "llvm/Support/raw_ostream.h"
#include "llvm/Support/ErrorHandling.h"
#include "llvm/Support/Debug.h"

#include <cassert>
#include <set>

#define FIXUP_DLOG(msg) DEBUG(llvm::errs() << DEBUG_TYPE ":: " <<
    msg << '\n')

using namespace llvm;
using namespace ssml;
using namespace ssml::ast;
using namespace ssml::ast::fix;

template <typename VectorT, typename ItemT, typename TupleT,
    typename SeqT>
OperatorFixupParser<VectorT, ItemT, TupleT,
    SeqT>::OperatorFixupParser(
    VectorT &Operators, OperatorDefMap &OpDefs)
    : Operators(Operators), Current(Operators.begin()),
        End(Operators.end()),
      OpDefs(OpDefs) {}

template <typename VectorT, typename ItemT, typename TupleT,
    typename SeqT>
void OperatorFixupParser<VectorT, ItemT, TupleT,
    SeqT>::errorExitIfInfix(
    SharedPtrT P) {
  switch (getDef(P).getAssoc()) {
  case OperatorDefinition::INFIX:
    errorExit(P->getLocation(), "unexpected infix operator");
  case OperatorDefinition::INFIXR:
    errorExit(P->getLocation(), "unexpected infixr operator");
  default:
    break;
  }
}

template <typename VectorT, typename ItemT, typename TupleT,
    typename SeqT>
auto OperatorFixupParser<VectorT, ItemT, TupleT,
    SeqT>::infixToApply(
    Precedence OpPrec, SharedPtrVec LHS, SharedPtrT BinOp) ->
        SharedPtrVec {
  FIXUP_DLOG("convert binary expression into application
      expression");

```

```
  SharedPtrVec Apply(new VectorT(BinOp->getLocation(), BinOp));

  auto RHS = this->next();
  if (!RHS)
    errorExit(BinOp->getLocation(), "expected operand after
        operator");
  errorExitIfInfix(RHS);

  SharedPtrSeq Args(new SeqT(LHS->getLocation(), LHS));
  SharedPtrVec Next(new VectorT(RHS->getLocation(), RHS));
  if (OpPrec > OperatorDefinition::maxPrecedence())
    Args->push_back(this->application(Next));
  else
    Args->push_back(this->binary(OpPrec, Next));

  SharedPtrTup Tup(new TupleT(Args->getLocation(), Args));
  Apply->push_back(Tup);

  return Apply;
}

template <typename VectorT, typename ItemT, typename TupleT,
    typename SeqT>
auto OperatorFixupParser<VectorT, ItemT, TupleT, SeqT>::binary(
    Precedence OpPrec, SharedPtrVec LHS) -> SharedPtrVec {
  FIXUP_DLOG("binary expression begin");

  while (auto BinOp = this->peek()) {
    FIXUP_DLOG("binary expression loop start");

    auto BinOpDef = getDef(BinOp);

    if (BinOpDef.getAssoc() == OperatorDefinition::NONFIX) {
      FIXUP_DLOG("operator is NONFIX");
      LHS = application(LHS);
      continue;
    }

    if (BinOpDef.getPrec() < OpPrec)
      break;

    if (BinOpDef.getPrec() > OpPrec) {
      FIXUP_DLOG("goto next operator precedence");
      LHS = binary(BinOpDef.getPrec(), LHS);
      continue;
    }

    this->next();
    FIXUP_DLOG("found correct operator precedence");
    if (BinOpDef.getAssoc() == OperatorDefinition::INFIX)
      LHS = infixToApply(OpPrec + 1, LHS, BinOp);
    else /* INFIXR */
      LHS = infixToApply(OpPrec, LHS, BinOp);
  }

  FIXUP_DLOG("binary expression return");
  return LHS;
}

template <typename VectorT, typename ItemT, typename TupleT,
    typename SeqT>
auto OperatorFixupParser<VectorT, ItemT, TupleT,
```

```
    SeqT>::application(
    SharedPtrVec Op) -> SharedPtrVec {
  FIXUP_DLOG("application expression begin");
  SharedPtrT Next;
  while ((Next = peek()) &&
         getDef(Next).getAssoc() == OperatorDefinition::NONFIX) {
    FIXUP_DLOG("application expression apply argument");
    Op->push_back(this->next());
  }
  FIXUP_DLOG("application expression return");
  return Op;
}

namespace {
struct PrecedenceLess {
  bool operator()(OperatorDefinition L, OperatorDefinition R) const {
    return L.getPrec() < R.getPrec();
  }
};
} // End annonymous namespace.

template <typename VectorT, typename ItemT, typename TupleT, typename SeqT>
void OperatorFixupParser<VectorT, ItemT, TupleT,
                         SeqT>::errorExitIfIllegalOperatorMix() {
  std::set<OperatorDefinition, PrecedenceLess> OpSet;
  for (auto X : this->Operators) {
    auto Def = getDef(X);
    if (Def.getAssoc() == OperatorDefinition::NONFIX)
      continue;

    if (OpSet.count(Def)) {
      if (OpSet.find(Def)->getAssoc() != Def.getAssoc())
        errorExit(Operators.back()->getLocation(), "mix of infix and infixr "
                                                   "operators with same "
                                                   "precedence");
    }

    OpSet.insert(Def);
  }
}

template <typename VectorT, typename ItemT, typename TupleT, typename SeqT>
auto OperatorFixupParser<VectorT, ItemT, TupleT, SeqT>::parse()
    -> SharedPtrVec {
  FIXUP_DLOG("parse");

  auto First = next();
  this->errorExitIfInfix(First);
  SharedPtrVec LHS(new VectorT(First->getLocation(), First));
  auto Ret = this->binary(OperatorDefinition::minPrecedence(), LHS);

  this->errorExitIfIllegalOperatorMix();
  return Ret;
}

template <typename VectorT, typename ItemT, typename TupleT,
```

```
    typename SeqT>
OperatorDefinition
OperatorFixupParser<VectorT, ItemT, TupleT,
    SeqT>::getDef(SharedPtrT P) {
  return OperatorDefVisitor::getOperatorDefinition(*P,
      this->OpDefs);
}

template <typename VectorT, typename ItemT, typename TupleT,
    typename SeqT>
auto OperatorFixupParser<VectorT, ItemT, TupleT, SeqT>::peek() ->
    SharedPtrT {
  if (this->Current == this->End)
    return nullptr;
  return *this->Current;
}

template <typename VectorT, typename ItemT, typename TupleT,
    typename SeqT>
auto OperatorFixupParser<VectorT, ItemT, TupleT, SeqT>::next() ->
    SharedPtrT {
  if (this->Current == this->End)
    return nullptr;
  SharedPtrT Ret = *this->Current;
  ++Current;
  return Ret;
}

template class ssml::ast::fix::OperatorFixupParser<
    ApplyExpression, Expression, TupleExpression, SeqExpression>;
template class ssml::ast::fix::OperatorFixupParser<ApplyPattern,
    Pattern,
                                                   TuplePattern,
                                                       SeqPattern>;
```

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

```
#ifndef LLVM_TOOLS_SSML_LIB_AST_FIXUP_OPERATORDEFVISITOR_H
#define LLVM_TOOLS_SSML_LIB_AST_FIXUP_OPERATORDEFVISITOR_H

#include "OperatorDefinition.h"

#include "ssml/AST/Visitor.h"

#include "ssml/Common/SourceLocation.h"

namespace ssml {
namespace ast {
class Node;
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
namespace fix {
class OperatorDefMap;
} // End namespace fix.
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
```

```cpp
namespace ast {
namespace fix {
class OperatorDefVisitor : public Visitor {
private:
  OperatorDefinition OpDef;
  OperatorDefMap &OpDefs;

private:
  OperatorDefVisitor(OperatorDefMap &M);
  OperatorDefinition getShortIdentifierOpDef(ShortIdentifier &ID);
  OperatorDefinition getLongIdentifierOpDef(LongIdentifier &IDs,
                                            SourceLocation L,
                                            bool opKeyPrefixed);

public:
  static OperatorDefinition getOperatorDefinition(ssml::ast::Node &N,
                                                  OperatorDefMap &M);

  void visit(IntLiteral *) override;

  void visit(ShortIdentifier *) override;
  void visit(LongIdentifier *) override;
  void visit(IntIdentifier *) override;
  void visit(SeqShortIdentifier *) override;
  void visit(SeqLongIdentifier *) override;

  void visit(Match *) override;
  void visit(SeqMatch *) override;

  void visit(SeqExpression *) override;
  void visit(LiteralExpression *) override;
  void visit(LongIdentifierExpression *) override;
  void visit(TupleExpression *) override;
  void visit(ListExpression *) override;
  void visit(ApplyExpression *) override;
  void visit(OrElseExpression *) override;
  void visit(AndAlsoExpression *) override;
  void visit(LetExpression *) override;
  void visit(IfExpression *) override;
  void visit(WhileExpression *) override;
  void visit(CaseExpression *) override;
  void visit(LambdaExpression *) override;

  void visit(SeqPattern *) override;
  void visit(LiteralPattern *) override;
  void visit(WildcardPattern *) override;
  void visit(ShortIdentifierPattern *) override;
  void visit(LongIdentifierPattern *) override;
  void visit(TypePattern *) override;
  void visit(ApplyPattern *) override;
  void visit(ListPattern *) override;
  void visit(TuplePattern *) override;

  void visit(SeqDeclaration *) override;
  void visit(Root *) override;
  void visit(ValDeclaration *) override;
  void visit(OpenDeclaration *) override;
  void visit(NonfixDeclaration *) override;
  void visit(InfixDeclaration *) override;
  void visit(InfixRDeclaration *) override;
```

```
  void visit(RestrictedStructDeclaration *) override;
  void visit(BareStructDeclaration *) override;
  void visit(AbstractValDeclaration *) override;
  void visit(AbstractTypeDeclaration *) override;
  void visit(AbstractStructDeclaration *) override;
  void visit(SigDeclaration *) override;
  void visit(FunPatternDeclaration *) override;
  void visit(FunDeclaration *) override;
  void visit(TypeDeclaration *) override;
  void visit(DatatypeBareInstanceDeclaration *) override;
  void visit(DatatypeTypedInstanceDeclaration *) override;
  void visit(BareDatatypeDeclaration *) override;
  void visit(TypedDatatypeDeclaration *) override;
  void visit(ShortBareFunctorDeclaration *) override;
  void visit(ShortRestrictedFunctorDeclaration *) override;
  void visit(LongBareFunctorDeclaration *) override;
  void visit(LongRestrictedFunctorDeclaration *) override;

  void visit(LongIdentifierDefinition *) override;
  void visit(StructDefinition *) override;
  void visit(AnnotationDefinition *) override;
  void visit(ShortFunctorDefinition *) override;
  void visit(LongFunctorDefinition *) override;
  void visit(SigDefinition *) override;

  void visit(LongIdentifierType *) override;
  void visit(VariableType *) override;
  void visit(SeqType *) override;
  void visit(SeqVariableType *) override;
  void visit(TupleType *) override;
  void visit(ApplyType *) override;
  void visit(ProductType *) override;
  void visit(FunctionType *) override;
};
} // End namespace fix.
} // End namespace ast.
} // End namespace ssml.

#endif // LLVM_TOOLS_SSML_LIB_AST_FIXUP_OPERATORDEFVISITOR_H
```

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

```
#ifndef LLVM_TOOLS_SSML_LIB_AST_FIXUP_OPERATORDEFMAP_H
#define LLVM_TOOLS_SSML_LIB_AST_FIXUP_OPERATORDEFMAP_H

#include "OperatorDefinition.h"

#include "ssml/Common/Compare.h"

#include <map>
#include <vector>
#include <string>

namespace ssml {
namespace ast {
namespace fix {
class OperatorDefMap {
private:
  using MapType =
      std::map<std::shared_ptr<const std::string>,
          OperatorDefinition,
```

```
                ssml::SharedPtrLess<std::string>>;

private:
  std::vector<MapType> Maps;

public:
  OperatorDefinition get(std::shared_ptr<const std::string> K);
  void put(std::shared_ptr<const std::string> K,
      OperatorDefinition Def);
  void enterScope();
  void leaveScope();
};
} // End namespace fix.
} // End namespace ast.
} // End namespace ssml.

#endif // LLVM_TOOLS_SSML_LIB_AST_FIXUP_OPERATORDEFMAP_H
```

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

```
#ifndef LLVM_TOOLS_SSML_LIB_AST_FIXUP_OPERATORFIXUPPARSER_H
#define LLVM_TOOLS_SSML_LIB_AST_FIXUP_OPERATORFIXUPPARSER_H

#include "OperatorDefinition.h"

#include <vector>
#include <memory>
#include <string>

// parse: binary0
//
// binary0: binary0 'infix0' binary1
//         | binary1 'infixr0' binary0
//         | binary1
//
// binary1: binary1 'infix1' binary2
//         | binary2 'infixr1' binary1
//         | binary2
//
// ...
//
// binary9: binary9 'infix9' nonfix
//         | application 'infixr9' binary9
//         | application
//
// application: application 'nonfix'
//             | 'nonfix'

namespace ssml {
namespace ast {
namespace fix {
class OperatorDefMap;
} // End namespace fix.
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
class Expression;
class ApplyExpression;
class SeqExpression;
```

```cpp
class TupleExpression;
class Pattern;
class ApplyPattern;
class SeqPattern;
class TuplePattern;
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace ast {
namespace fix {
template <typename VectorT, typename ItemT, typename TupleT,
    typename SeqT>
class OperatorFixupParser {
private:
  // using VectorT = ApplyExpression;
  // using ItemT = Expression;
  using SharedPtrT = std::shared_ptr<ItemT>;
  using SharedPtrVec = std::shared_ptr<VectorT>;
  // using TupleT = TupleExpression;
  using SharedPtrTup = std::shared_ptr<TupleT>;
  // using SeqT = SeqExpression;
  using SharedPtrSeq = std::shared_ptr<SeqT>;
  using Precedence = OperatorDefinition::Precedence;

private:
  VectorT &Operators;
  typename std::vector<SharedPtrT>::iterator Current;
  typename std::vector<SharedPtrT>::iterator End;
  OperatorDefMap &OpDefs;

private:
  SharedPtrT peek();
  SharedPtrT next();

  SharedPtrVec infixToApply(Precedence OpPrec, SharedPtrVec LHS,
                            SharedPtrT BinOp);
  SharedPtrVec binary(Precedence CurrPrec, SharedPtrVec LHS);
  SharedPtrVec application(SharedPtrVec Op);

  OperatorDefinition getDef(SharedPtrT);

  void errorExitIfInfix(SharedPtrT P);

  void errorExitIfIllegalOperatorMix();

public:
  OperatorFixupParser(VectorT &Operators, OperatorDefMap &OpDefs);
  SharedPtrVec parse();
};
} // End namespace fix.
} // End namespace ast.
} // End namespace ssml.

extern template class ssml::ast::fix::OperatorFixupParser<
    ssml::ast::ApplyExpression, ssml::ast::Expression,
    ssml::ast::TupleExpression, ssml::ast::SeqExpression>;
extern template class ssml::ast::fix::OperatorFixupParser<
    ssml::ast::ApplyPattern, ssml::ast::Pattern,
        ssml::ast::TuplePattern,
    ssml::ast::SeqPattern>;

```

```
#endif // LLVM_TOOLS_SSML_LIB_AST_FIXUP_OPERATORFIXUPPARSER_H
```

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

```
#ifndef LLVM_TOOLS_SSML_LIB_AST_FIXUP_OPERATORDEFINITION_H
#define LLVM_TOOLS_SSML_LIB_AST_FIXUP_OPERATORDEFINITION_H

#include <memory>
#include <string>

namespace ssml {
namespace ast {
namespace fix {
class OperatorDefinition {
public:
  enum Associativity { NONFIX, INFIX, INFIXR };
  using Precedence = unsigned;
  constexpr static Precedence maxPrecedence() { return 9; }
  constexpr static Precedence minPrecedence() { return 0; }

private:
  std::shared_ptr<std::string> ID;
  Associativity Assoc;
  Precedence Prec;

public:
  explicit OperatorDefinition(
      std::shared_ptr<const std::string> ID =
          std::shared_ptr<const std::string>(nullptr),
      Associativity A = NONFIX,
      Precedence P = OperatorDefinition::minPrecedence());
  std::shared_ptr<std::string> getID() const { return this->ID; }
  Associativity getAssoc() const { return this->Assoc; }
  Precedence getPrec() const { return this->Prec; }
};
} // End namespace fix.
} // End namespace ast.
} // End namespace ssml.

#endif // LLVM_TOOLS_SSML_LIB_AST_FIXUP_OPERATORDEFINITION_H
```

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

```
#ifndef LLVM_TOOLS_SSML_LIB_AST_FIXUP_FIXUPVISITOR_H
#define LLVM_TOOLS_SSML_LIB_AST_FIXUP_FIXUPVISITOR_H

#include "OperatorDefMap.h"
#include "OperatorDefinition.h"

#include "ssml/AST/Visitor.h"
#include "ssml/Common/SourceLocation.h"

#include <memory>
#include <vector>

namespace ssml {
namespace ast {
class Node;
} // End namespace ast.
} // End namespace ssml.

```

```
namespace ssml {
namespace ast {
namespace fix {
class FixupVisitor : public Visitor {
private:
  OperatorDefinition CurrOpDef;
  OperatorDefMap OpDefs;

private:
  template <typename VectorT, typename ItemT, typename TupleT,
      typename SeqT>
  std::shared_ptr<VectorT> operatorFixup(VectorT &Initial);
  void setNonfixOpDef(SeqShortIdentifier &IDs);
  void setOpDef(SeqShortIdentifier &IDs,
      OperatorDefinition::Associativity,
                IntLiteral &Prec);

public:
  void visit(IntLiteral *) override;

  void visit(ShortIdentifier *) override;
  void visit(LongIdentifier *) override;
  void visit(IntIdentifier *) override;
  void visit(SeqShortIdentifier *) override;
  void visit(SeqLongIdentifier *) override;

  void visit(Match *) override;
  void visit(SeqMatch *) override;

  void visit(SeqExpression *) override;
  void visit(LiteralExpression *) override;
  void visit(LongIdentifierExpression *) override;
  void visit(TupleExpression *) override;
  void visit(ListExpression *) override;
  void visit(ApplyExpression *) override;
  void visit(OrElseExpression *) override;
  void visit(AndAlsoExpression *) override;
  void visit(LetExpression *) override;
  void visit(IfExpression *) override;
  void visit(WhileExpression *) override;
  void visit(CaseExpression *) override;
  void visit(LambdaExpression *) override;

  void visit(SeqPattern *) override;
  void visit(LiteralPattern *) override;
  void visit(WildcardPattern *) override;
  void visit(ShortIdentifierPattern *) override;
  void visit(LongIdentifierPattern *) override;
  void visit(TypePattern *) override;
  void visit(ApplyPattern *) override;
  void visit(ListPattern *) override;
  void visit(TuplePattern *) override;

  void visit(SeqDeclaration *) override;
  void visit(Root *) override;
  void visit(ValDeclaration *) override;
  void visit(NonfixDeclaration *) override;
  void visit(InfixDeclaration *) override;
  void visit(InfixRDeclaration *) override;
  void visit(RestrictedStructDeclaration *) override;
  void visit(BareStructDeclaration *) override;
  void visit(AbstractValDeclaration *) override;
```

```
  void visit(AbstractTypeDeclaration *) override;
  void visit(AbstractStructDeclaration *) override;
  void visit(SigDeclaration *) override;
  void visit(FunPatternDeclaration *) override;
  void visit(FunDeclaration *) override;
  void visit(TypeDeclaration *) override;
  void visit(DatatypeBareInstanceDeclaration *) override;
  void visit(DatatypeTypedInstanceDeclaration *) override;
  void visit(BareDatatypeDeclaration *) override;
  void visit(TypedDatatypeDeclaration *) override;
  void visit(ShortBareFunctorDeclaration *) override;
  void visit(ShortRestrictedFunctorDeclaration *) override;
  void visit(LongBareFunctorDeclaration *) override;
  void visit(LongRestrictedFunctorDeclaration *) override;

  void visit(LongIdentifierDefinition *) override;
  void visit(StructDefinition *) override;
  void visit(AnnotationDefinition *) override;
  void visit(ShortFunctorDefinition *) override;
  void visit(LongFunctorDefinition *) override;
  void visit(SigDefinition *) override;

  void visit(LongIdentifierType *) override;
  void visit(VariableType *) override;
  void visit(SeqType *) override;
  void visit(SeqVariableType *) override;
  void visit(TupleType *) override;
  void visit(ApplyType *) override;
  void visit(ProductType *) override;
  void visit(FunctionType *) override;
};
} // End namespace fix.
} // End namespace ast.
} // End namespace ssml.

#endif // LLVM_TOOLS_SSML_LIB_AST_FIXUP_FIXUPVISITOR_H
```

Listing 67: lib/AST/Node.cpp

```
#include "ssml/AST/Node.h"

using namespace ssml::ast;

Node::Node(SourceLocation L) : Location(L) {}

Node::~Node() = default;
```

Listing 68: lib/AST/DumpVisitor.cpp

```
#include "ssml/AST/DumpVisitor.h"
#include "ssml/AST/Declaration.h"
#include "ssml/AST/Definition.h"
#include "ssml/AST/Pattern.h"
#include "ssml/AST/Expression.h"
#include "ssml/AST/Literal.h"
#include "ssml/AST/Identifier.h"
#include "ssml/AST/Type.h"
#include "ssml/AST/Match.h"
#include "ssml/Common/SourceLocation.h"

#include "llvm/ADT/StringRef.h"
```

```
#include "llvm/Support/raw_ostream.h"

using namespace llvm;
using namespace ssml;
using namespace ssml::ast;

static constexpr unsigned IndentationSize = 2;

void DumpVisitor::printIndent() {
  for (unsigned i = 0; i < this->Indent; ++i)
    outs() << ' ';
}

void DumpVisitor::incrementIndent() { this->Indent += IndentationSize; }

void DumpVisitor::decrementIndent() { this->Indent -= IndentationSize; }

void DumpVisitor::printAttribute(const char *Name, const std::string &Value) {
  outs() << ' ' << Name << "=" << Value;
}

void DumpVisitor::printAttribute(const char *Name,
                                 std::shared_ptr<const std::string> Value) {
  this->printAttribute(Name, *Value);
}

void DumpVisitor::openBeginTag(const char *Tag, SourceLocation L) {
  this->printIndent();
  outs() << '<' << Tag;
  this->printAttribute("location", L.toString());
}

void DumpVisitor::closeBeginTag() {
  outs() << ">\n";
  this->incrementIndent();
}

void DumpVisitor::openCloseEndTag(const char *Tag) {
  this->decrementIndent();
  this->printIndent();
  outs() << "</" << Tag << ">\n";
}

void DumpVisitor::visit(IntLiteral *N) {
  constexpr auto Tag = "IntLiteral";
  this->openBeginTag(Tag, N->getLocation());
  this->printAttribute("value", N->getValueString());
  this->closeBeginTag();
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(RealLiteral *N) {
  constexpr auto Tag = "RealLiteral";
  this->openBeginTag(Tag, N->getLocation());
  this->printAttribute("value", N->getValueString());
  this->closeBeginTag();
  this->openCloseEndTag(Tag);
}
```

```cpp

void DumpVisitor::visit(CharLiteral *N) {
  constexpr auto Tag = "CharLiteral";
  this->openBeginTag(Tag, N->getLocation());
  this->printAttribute("value", N->getValueString());
  this->closeBeginTag();
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(StringLiteral *N) {
  constexpr auto Tag = "StringLiteral";
  this->openBeginTag(Tag, N->getLocation());
  this->printAttribute("value", N->getValueString());
  this->closeBeginTag();
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(Match *N) {
  constexpr auto Tag = "Match";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  N->getPattern()->accept(this);
  N->getExpr()->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(SeqMatch *N) {
  constexpr auto Tag = "SeqMatch";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  for (auto M : *N)
    M->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(ShortIdentifier *N) {
  constexpr auto Tag = "ShortIdentifier";
  this->openBeginTag(Tag, N->getLocation());
  this->printAttribute("value", N->getID());
  this->closeBeginTag();
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(LongIdentifier *N) {
  constexpr auto Tag = "LongIdentifier";
  this->openBeginTag(Tag, N->getLocation());
  this->printAttribute("value", N->toString());
  this->closeBeginTag();
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(IntIdentifier *N) {
  constexpr auto Tag = "IntIdentifier";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  N->getLiteral()->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(SeqShortIdentifier *N) {
  constexpr auto Tag = "SeqShortIdentifier";
  this->openBeginTag(Tag, N->getLocation());
```

```
  this->closeBeginTag();
  for (auto ID : *N)
    ID->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(SeqLongIdentifier *N) {
  constexpr auto Tag = "SeqLongIdentifier";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  for (auto ID : *N)
    ID->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(SeqExpression *N) {
  constexpr auto Tag = "SeqExpression";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  for (auto E : *N)
    E->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(LiteralExpression *N) {
  constexpr auto Tag = "LiteralExpression";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  N->getValue()->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(LongIdentifierExpression *N) {
  constexpr auto Tag = "LongIdentifierExpression";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  N->getIDs()->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(TupleExpression *N) {
  constexpr auto Tag = "TupleExpression";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  N->getExprs()->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(ListExpression *N) {
  constexpr auto Tag = "ListExpression";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  N->getExprs()->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(IdentifierEqualsExpression *N) {
  constexpr auto Tag = "IdentifierEqualsExpression";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  N->getID()->accept(this);
  N->getExpr()->accept(this);
```

```
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(RecordExpression *N) {
  constexpr auto Tag = "RecordExpression";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  for (auto E : *N)
    E->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(SelectorExpression *N) {
  constexpr auto Tag = "SelectorExpression";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  N->getID()->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(ApplyExpression *N) {
  constexpr auto Tag = "ApplyExpression";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  for (auto E : *N)
    E->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(TypeExpression *N) {
  constexpr auto Tag = "TypeExpression";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  N->getExpr()->accept(this);
  N->getType()->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(OrElseExpression *N) {
  constexpr auto Tag = "OrElseExpression";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  N->getLeftExpr()->accept(this);
  N->getRightExpr()->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(AndAlsoExpression *N) {
  constexpr auto Tag = "AndAlsoExpression";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  N->getLeftExpr()->accept(this);
  N->getRightExpr()->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(LetExpression *N) {
  constexpr auto Tag = "LetExpression";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  N->getDecls()->accept(this);
  N->getExprs()->accept(this);
```

```
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(IfExpression *N) {
  constexpr auto Tag = "IfExpression";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  N->getCondExpr()->accept(this);
  N->getThenExpr()->accept(this);
  N->getElseExpr()->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(WhileExpression *N) {
  constexpr auto Tag = "WhileExpression";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  N->getCondExpr()->accept(this);
  N->getBodyExpr()->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(CaseExpression *N) {
  constexpr auto Tag = "CaseExpression";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  N->getExpr()->accept(this);
  N->getCases()->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(LambdaExpression *N) {
  constexpr auto Tag = "LambdaExpression";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  N->getCases()->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(SeqPattern *N) {
  constexpr auto Tag = "SeqPattern";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  for (auto P : *N)
    P->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(LiteralPattern *N) {
  constexpr auto Tag = "LiteralPattern";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  N->getValue()->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(WildcardPattern *N) {
  constexpr auto Tag = "WildcardPattern";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  this->openCloseEndTag(Tag);
}
```

```
void DumpVisitor::visit(ShortIdentifierPattern *N) {
  constexpr auto Tag = "ShortIdentifierPattern";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  N->getID()->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(LongIdentifierPattern *N) {
  constexpr auto Tag = "LongIdentifierPattern";
  this->openBeginTag(Tag, N->getLocation());
  if (N->isPrefixedWithOpKey())
    this->printAttribute("opPrefix", "true");
  else
    this->printAttribute("opPrefix", "false");
  this->closeBeginTag();
  N->getIDs()->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(TypePattern *N) {
  constexpr auto Tag = "TypePattern";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  N->getPattern()->accept(this);
  N->getType()->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(AsPattern *N) {
  constexpr auto Tag = "AsPattern";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  N->getLeftPattern()->accept(this);
  N->getRightPattern()->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(ApplyPattern *N) {
  constexpr auto Tag = "ApplyPattern";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  for (auto P : *N)
    P->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(IdentifierEqualsPattern *N) {
  constexpr auto Tag = "IdentifierEqualsPattern";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  N->getID()->accept(this);
  N->getPattern()->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(RecordPattern *N) {
  constexpr auto Tag = "RecordPattern";
  this->openBeginTag(Tag, N->getLocation());
  if (N->isEllipsisTerminated())
    this->printAttribute("ellipsis", "true");
```

```
  else
    this->printAttribute("ellipsis", "false");
  this->closeBeginTag();
  for (auto P : *N)
    P->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(ListPattern *N) {
  constexpr auto Tag = "ListPattern";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  N->getPatterns()->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(TuplePattern *N) {
  constexpr auto Tag = "TuplePattern";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  N->getPatterns()->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(SeqDeclaration *N) {
  constexpr auto Tag = "SeqDeclaration";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  for (auto D : *N)
    D->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(Root *N) {
  constexpr auto Tag = "Root";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  for (auto D : *N)
    D->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(ValDeclaration *N) {
  constexpr auto Tag = "ValDeclaration";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  N->getDest()->accept(this);
  N->getSource()->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(OpenDeclaration *N) {
  constexpr auto Tag = "OpenDeclaration";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  N->getIDs()->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(NonfixDeclaration *N) {
  constexpr auto Tag = "NonfixDeclaration";
  this->openBeginTag(Tag, N->getLocation());
```

```
  this->closeBeginTag();
  N->getIDs()->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(InfixDeclaration *N) {
  constexpr auto Tag = "InfixDeclaration";
  this->openBeginTag(Tag, N->getLocation());
  this->printAttribute("precedence",
      N->getPrecedence()->getValueString());
  this->closeBeginTag();
  N->getIDs()->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(InfixRDeclaration *N) {
  constexpr auto Tag = "InfixRDeclaration";
  this->openBeginTag(Tag, N->getLocation());
  this->printAttribute("precedence",
      N->getPrecedence()->getValueString());
  this->closeBeginTag();
  N->getIDs()->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(BareStructDeclaration *N) {
  constexpr auto Tag = "BareStructDeclaration";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  N->getID()->accept(this);
  N->getStructDef()->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(RestrictedStructDeclaration *N) {
  constexpr auto Tag = "RestrictedStructDeclaration";
  this->openBeginTag(Tag, N->getLocation());
  if (N->isSigTransparent())
    this->printAttribute("sigTransparent", "true");
  else
    this->printAttribute("sigTransparent", "false");
  this->closeBeginTag();
  N->getID()->accept(this);
  N->getSigDef()->accept(this);
  N->getStructDef()->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(AbstractValDeclaration *N) {
  constexpr auto Tag = "AbstractValDeclaration";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  N->getID()->accept(this);
  N->getType()->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(AbstractTypeDeclaration *N) {
  constexpr auto Tag = "AbstractTypeDeclaration";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  N->getID()->accept(this);
```

```
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(AbstractStructDeclaration *N) {
  constexpr auto Tag = "AbstractStructDeclaration";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  N->getID()->accept(this);
  N->getSigDef()->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(SharingTypeDeclaration *N) {
  constexpr auto Tag = "SharingTypeDeclaration";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  for (auto T : *N)
    T->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(SigDeclaration *N) {
  constexpr auto Tag = "SigDeclaration";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  N->getID()->accept(this);
  N->getSigDef()->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(FunPatternDeclaration *N) {
  constexpr auto Tag = "FunPatternDeclaration";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  N->getIDParams()->accept(this);
  N->getDef()->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(FunDeclaration *N) {
  constexpr auto Tag = "FunDeclaration";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  for (auto P : *N)
    P->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(TypeDeclaration *N) {
  constexpr auto Tag = "TypeDeclaration";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  N->getID()->accept(this);
  N->getType()->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(DatatypeBareInstanceDeclaration *N) {
  constexpr auto Tag = "DatatypeBareInstanceDeclaration";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  N->getID()->accept(this);
```

```
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(DatatypeTypedInstanceDeclaration *N) {
  constexpr auto Tag = "DatatypeTypedInstanceDeclaration";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  N->getID()->accept(this);
  N->getType()->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(BareDatatypeDeclaration *N) {
  constexpr auto Tag = "BareDatatypeDeclaration";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  N->getID()->accept(this);
  N->getDef()->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(TypedDatatypeDeclaration *N) {
  constexpr auto Tag = "TypedDatatypeDeclaration";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  N->getTypeParams()->accept(this);
  N->getID()->accept(this);
  N->getDef()->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(ShortBareFunctorDeclaration *N) {
  constexpr auto Tag = "ShortBareFunctorDeclaration";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  N->getFunctorID()->accept(this);
  N->getParamID()->accept(this);
  N->getParamSigDef()->accept(this);
  N->getFunctorStructDef()->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(ShortRestrictedFunctorDeclaration *N) {
  constexpr auto Tag = "ShortRestrictedFunctorDeclaration";
  this->openBeginTag(Tag, N->getLocation());
  if (N->isFunctorSigDefTransparent())
    this->printAttribute("transparent", "true");
  else
    this->printAttribute("transparent", "false");
  this->closeBeginTag();
  N->getFunctorID()->accept(this);
  N->getParamID()->accept(this);
  N->getParamSigDef()->accept(this);
  N->getFunctorSigDef()->accept(this);
  N->getFunctorStructDef()->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(LongBareFunctorDeclaration *N) {
  constexpr auto Tag = "LongBareFunctorDeclaration";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
```

```
  N->getFunctorID()->accept(this);
  N->getParams()->accept(this);
  N->getFunctorStructDef()->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(LongRestrictedFunctorDeclaration *N) {
  constexpr auto Tag = "LongRestrictedFunctorDeclaration";
  this->openBeginTag(Tag, N->getLocation());
  if (N->isFunctorSigDefTransparent())
    this->printAttribute("transparent", "true");
  else
    this->printAttribute("transparent", "false");
  this->closeBeginTag();
  N->getFunctorID()->accept(this);
  N->getParams()->accept(this);
  N->getFunctorSigDef()->accept(this);
  N->getFunctorStructDef()->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(LongIdentifierDefinition *N) {
  constexpr auto Tag = "LongIdentifierDefinition";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  N->getID()->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(StructDefinition *N) {
  constexpr auto Tag = "StructDefinition";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  N->getDecls()->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(AnnotationDefinition *N) {
  constexpr auto Tag = "AnnotationDefinition";
  this->openBeginTag(Tag, N->getLocation());
  if (N->isTransparent())
    this->printAttribute("transparent", "true");
  else
    this->printAttribute("transparent", "false");
  this->closeBeginTag();
  N->getStructDef()->accept(this);
  N->getSigDef()->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(ShortFunctorDefinition *N) {
  constexpr auto Tag = "ShortFunctorDefinition";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  N->getFunctorID()->accept(this);
  N->getStructDef()->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(LongFunctorDefinition *N) {
  constexpr auto Tag = "LongFunctorDefinition";
  this->openBeginTag(Tag, N->getLocation());
```

```
  this->closeBeginTag();
  N->getFunctorID()->accept(this);
  N->getDecls()->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(SigDefinition *N) {
  constexpr auto Tag = "SigDefinition";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  N->getDecls()->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(LongIdentifierType *N) {
  constexpr auto Tag = "LongIdentifierType";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  N->getIDs()->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(VariableType *N) {
  constexpr auto Tag = "VariableType";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  N->getID()->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(SeqType *N) {
  constexpr auto Tag = "SeqType";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  for (auto T : *N)
    T->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(SeqVariableType *N) {
  constexpr auto Tag = "SeqVariableType";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  for (auto T : *N)
    T->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(TupleType *N) {
  constexpr auto Tag = "TupleType";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  N->getTypes()->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(IdentifierColonType *N) {
  constexpr auto Tag = "IdentifierColonType";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  N->getIDs()->accept(this);
  N->getType()->accept(this);
```

```
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(RecordType *N) {
  constexpr auto Tag = "RecordType";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  for (auto T : *N)
    T->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(ApplyType *N) {
  constexpr auto Tag = "ApplyType";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  for (auto T : *N)
    T->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(ProductType *N) {
  constexpr auto Tag = "ProductType";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  for (auto T : *N)
    T->accept(this);
  this->openCloseEndTag(Tag);
}

void DumpVisitor::visit(FunctionType *N) {
  constexpr auto Tag = "FunctionType";
  this->openBeginTag(Tag, N->getLocation());
  this->closeBeginTag();
  for (auto T : *N)
    T->accept(this);
  this->openCloseEndTag(Tag);
}
```

Listing 69: lib/AST/Definition.cpp

```
#include "ssml/AST/Definition.h"

#include "ssml/AST/Identifier.h"
#include "ssml/AST/Visitor.h"
#include "ssml/AST/Declaration.h"

using namespace ssml::ast;

Definition::Definition(SourceLocation L) : Node(L) {}

LongIdentifierDefinition::LongIdentifierDefinition(
    SourceLocation L, std::shared_ptr<LongIdentifier> ID)
    : Definition(L), ID(ID) {}

void LongIdentifierDefinition::accept(Visitor *V) {
    V->visit(this); }

StructDefinition::StructDefinition(SourceLocation L,
                                   std::shared_ptr<SeqDeclaration>
                                       Ds)
```

```
    : Definition(L), Decls(Ds) {}

void StructDefinition::accept(Visitor *V) { V->visit(this); }

AnnotationDefinition::AnnotationDefinition(SourceLocation L,
                                           std::shared_ptr<Definition>
                                               Struct,
                                           std::shared_ptr<Definition>
                                               Sig,
                                           bool T)
    : Definition(L), StructDef(Struct), SigDef(Sig),
        Transparent(T) {}

void AnnotationDefinition::accept(Visitor *V) { V->visit(this); }

ShortFunctorDefinition::ShortFunctorDefinition(
    SourceLocation L, std::shared_ptr<LongIdentifier> ID,
    std::shared_ptr<Definition> Struct)
    : Definition(L), FunctorID(ID), StructDef(Struct) {}

void ShortFunctorDefinition::accept(Visitor *V) { V->visit(this); }

LongFunctorDefinition::LongFunctorDefinition(SourceLocation L,
                                             std::shared_ptr<LongIdentifier>
                                                 ID,
                                             std::shared_ptr<SeqDeclaration>
                                                 Ds)
    : Definition(L), FunctorID(ID), Decls(Ds) {}

void LongFunctorDefinition::accept(Visitor *V) { V->visit(this); }

SigDefinition::SigDefinition(SourceLocation L,
                             std::shared_ptr<SeqDeclaration> Ds)
    : Definition(L), Decls(Ds) {}

void SigDefinition::accept(Visitor *V) { V->visit(this); }
```

Listing 70: lib/AST/Declaration.cpp

```
#include "ssml/AST/Declaration.h"

#include "ssml/AST/Visitor.h"
#include "ssml/AST/Pattern.h"
#include "ssml/AST/Expression.h"
#include "ssml/AST/Identifier.h"
#include "ssml/AST/Definition.h"
#include "ssml/AST/Type.h"

using namespace ssml::ast;

Declaration::Declaration(SourceLocation L) : Node(L) {}

SeqDeclaration::SeqDeclaration(SourceLocation L) : Declaration(L)
    {}

SeqDeclaration::SeqDeclaration(SourceLocation L,
                               std::shared_ptr<Declaration> First)
    : Declaration(L),
        std::vector<std::shared_ptr<Declaration>>{First} {}

void SeqDeclaration::accept(Visitor *V) { V->visit(this); }
```

```

Root::Root(SourceLocation L) : SeqDeclaration(L) {}

Root::Root(SourceLocation L, std::shared_ptr<Declaration> F)
    : SeqDeclaration(L, F) {}

void Root::accept(Visitor *V) { V->visit(this); }

ValDeclaration::ValDeclaration(SourceLocation L, std::shared_ptr<Pattern> Dest,
                               std::shared_ptr<Expression> Src)
    : Declaration(L), Dest(Dest), Source(Src) {}

void ValDeclaration::accept(Visitor *V) { V->visit(this); }

OpenDeclaration::OpenDeclaration(SourceLocation L,
                                 std::shared_ptr<SeqLongIdentifier> IDs)
    : Declaration(L), IDs(IDs) {}

void OpenDeclaration::accept(Visitor *V) { V->visit(this); }

NonfixDeclaration::NonfixDeclaration(SourceLocation L,
                                     std::shared_ptr<SeqShortIdentifier> IDs)
    : Declaration(L), IDs(IDs) {}

void NonfixDeclaration::accept(Visitor *V) { V->visit(this); }

InfixDeclaration::InfixDeclaration(SourceLocation L,
                                   std::shared_ptr<IntLiteral> Prec,
                                   std::shared_ptr<SeqShortIdentifier> IDs)
    : Declaration(L), Precedence(Prec), Idens(IDs) {}

void InfixDeclaration::accept(Visitor *V) { V->visit(this); }

InfixRDeclaration::InfixRDeclaration(SourceLocation L,
                                     std::shared_ptr<IntLiteral> Prec,
                                     std::shared_ptr<SeqShortIdentifier> IDs)
    : Declaration(L), Precedence(Prec), Idens(IDs) {}

void InfixRDeclaration::accept(Visitor *V) { V->visit(this); }

RestrictingSigDefinition::RestrictingSigDefinition(
    std::shared_ptr<Definition> D, bool B)
    : Def(D), IsTransparent(B) {}

BareStructDeclaration::BareStructDeclaration(
    SourceLocation L, std::shared_ptr<ShortIdentifier> ID,
    std::shared_ptr<Definition> Def)
    : Declaration(L), ID(ID), StructDef(Def) {}

void BareStructDeclaration::accept(Visitor *V) { V->visit(this); }

RestrictedStructDeclaration::RestrictedStructDeclaration(
    SourceLocation L, std::shared_ptr<ShortIdentifier> ID,
    RestrictingSigDefinition SigDef, std::shared_ptr<Definition> StructDef)
```

```
    : BareStructDeclaration(L, ID, StructDef), SigDef(SigDef) {}

void RestrictedStructDeclaration::accept(Visitor *V) { V->visit(this); }

AbstractValDeclaration::AbstractValDeclaration(
    SourceLocation L, std::shared_ptr<ShortIdentifier> ID,
    std::shared_ptr<Type> T)
    : Declaration(L), ID(ID), Typ(T) {}

void AbstractValDeclaration::accept(Visitor *V) { V->visit(this); }

AbstractTypeDeclaration::AbstractTypeDeclaration(
    SourceLocation L, std::shared_ptr<ShortIdentifier> ID)
    : Declaration(L), ID(ID) {}

void AbstractTypeDeclaration::accept(Visitor *V) { V->visit(this); }

AbstractStructDeclaration::AbstractStructDeclaration(
    SourceLocation L, std::shared_ptr<ShortIdentifier> ID,
    std::shared_ptr<Definition> D)
    : Declaration(L), ID(ID), SigDef(D) {}

void AbstractStructDeclaration::accept(Visitor *V) { V->visit(this); }

SharingTypeDeclaration::SharingTypeDeclaration(
    SourceLocation L, std::shared_ptr<LongIdentifier> First)
    : Declaration(L), std::vector<std::shared_ptr<LongIdentifier>>{First} {}

void SharingTypeDeclaration::accept(Visitor *V) { V->visit(this); }

SigDeclaration::SigDeclaration(SourceLocation L,
                               std::shared_ptr<ShortIdentifier> ID,
                               std::shared_ptr<Definition> Def)
    : Declaration(L), ID(ID), Def(Def) {}

void SigDeclaration::accept(Visitor *V) { V->visit(this); }

FunPatternDeclaration::FunPatternDeclaration(SourceLocation L,
                                             std::shared_ptr<Pattern> P,
                                             std::shared_ptr<Expression> Def)
    : Declaration(L), IDParams(P), Def(Def) {}

void FunPatternDeclaration::accept(Visitor *V) { V->visit(this); }

FunDeclaration::FunDeclaration(SourceLocation L,
                               std::shared_ptr<FunPatternDeclaration> F)
    : Declaration(L), std::vector<std::shared_ptr<FunPatternDeclaration>>{F} {}

void FunDeclaration::accept(Visitor *V) { V->visit(this); }

TypeDeclaration::TypeDeclaration(SourceLocation L,
                                 std::shared_ptr<ShortIdentifier> ID,
                                 std::shared_ptr<Type> Typ)
```

```
    : Declaration(L), ID(ID), Typ(Typ) {}

void TypeDeclaration::accept(Visitor *V) { V->visit(this); }

DatatypeBareInstanceDeclaration::DatatypeBareInstanceDeclaration(
    SourceLocation L, std::shared_ptr<ShortIdentifier> ID)
    : Declaration(L), ID(ID) {}

void DatatypeBareInstanceDeclaration::accept(Visitor *V) {
    V->visit(this); }

DatatypeTypedInstanceDeclaration::DatatypeTypedInstanceDeclaration(
    SourceLocation L, std::shared_ptr<ShortIdentifier> ID,
    std::shared_ptr<Type> Typ)
    : DatatypeBareInstanceDeclaration(L, ID), Typ(Typ) {}

void DatatypeTypedInstanceDeclaration::accept(Visitor *V) {
    V->visit(this); }

BareDatatypeDeclaration::BareDatatypeDeclaration(
    SourceLocation L, std::shared_ptr<ShortIdentifier> ID,
    std::shared_ptr<SeqDeclaration> D)
    : Declaration(L), ID(ID), Def(D) {}

void BareDatatypeDeclaration::accept(Visitor *V) { V->visit(this);
    }

TypedDatatypeDeclaration::TypedDatatypeDeclaration(
    SourceLocation L, std::shared_ptr<SeqVariableType> TypeParams,
    std::shared_ptr<ShortIdentifier> ID,
        std::shared_ptr<SeqDeclaration> Def)
    : BareDatatypeDeclaration(L, ID, Def), TypeParams(TypeParams)
        {}

void TypedDatatypeDeclaration::accept(Visitor *V) {
    V->visit(this); }

ShortBareFunctorDeclaration::ShortBareFunctorDeclaration(
    SourceLocation L, std::shared_ptr<ShortIdentifier> FunctorID,
    std::shared_ptr<ShortIdentifier> ParamID,
    std::shared_ptr<Definition> ParamSig,
        std::shared_ptr<Definition> StructDef)
    : Declaration(L), FunctorID(FunctorID), ParamID(ParamID),
      ParamSigDef(ParamSig), FunctorStructDef(StructDef) {}

void ShortBareFunctorDeclaration::accept(Visitor *V) {
    V->visit(this); }

ShortRestrictedFunctorDeclaration::ShortRestrictedFunctorDeclaration(
    SourceLocation L, std::shared_ptr<ShortIdentifier> FunctorID,
    std::shared_ptr<ShortIdentifier> ParamID,
    std::shared_ptr<Definition> ParamSig,
    RestrictingSigDefinition FunctorSigDef,
    std::shared_ptr<Definition> StructDef)
    : ShortBareFunctorDeclaration(L, FunctorID, ParamID, ParamSig,
        StructDef),
      FunctorSigDef(FunctorSigDef) {}

void ShortRestrictedFunctorDeclaration::accept(Visitor *V) {
    V->visit(this); }

LongBareFunctorDeclaration::LongBareFunctorDeclaration(
```

```
    SourceLocation L, std::shared_ptr<ShortIdentifier> FunctorID,
    std::shared_ptr<SeqDeclaration> Params,
    std::shared_ptr<Definition> StructDef)
    : Declaration(L), FunctorID(FunctorID), Params(Params),
      FunctorStructDef(StructDef) {}

void LongBareFunctorDeclaration::accept(Visitor *V) {
    V->visit(this); }

LongRestrictedFunctorDeclaration::LongRestrictedFunctorDeclaration(
    SourceLocation L, std::shared_ptr<ShortIdentifier> FunctorID,
    std::shared_ptr<SeqDeclaration> Params,
        RestrictingSigDefinition SigDef,
    std::shared_ptr<Definition> StructDef)
    : LongBareFunctorDeclaration(L, FunctorID, Params, StructDef),
      FunctorSigDef(SigDef) {}

void LongRestrictedFunctorDeclaration::accept(Visitor *V) {
    V->visit(this); }
```

Listing 71: lib/AST/Expression.cpp

```
#include "ssml/AST/Expression.h"
#include "ssml/AST/Visitor.h"
#include "ssml/AST/Identifier.h"
#include "ssml/AST/Type.h"
#include "ssml/AST/Declaration.h"
#include "ssml/AST/Match.h"

#include <cassert>

using namespace ssml::ast;

Expression::Expression(SourceLocation L) : Node(L) {}

LongIdentifierExpression *Expression::asLongIdentifierExpression()
    {
  return nullptr;
}

TupleExpression *Expression::asTupleExpression() { return nullptr;
    }

SeqExpression::SeqExpression(SourceLocation L) : Expression(L) {}

SeqExpression::SeqExpression(SourceLocation L,
    std::shared_ptr<Expression> F)
    : Expression(L), std::vector<std::shared_ptr<Expression>>{F} {}

void SeqExpression::accept(Visitor *V) { V->visit(this); }

LiteralExpression::LiteralExpression(SourceLocation L,
                                     std::shared_ptr<Literal> Lit)
    : Expression(L), Value(Lit) {}

void LiteralExpression::accept(Visitor *V) { V->visit(this); }

LongIdentifierExpression::LongIdentifierExpression(
    SourceLocation L, std::shared_ptr<LongIdentifier> ID, bool
        OpKey)
    : Expression(L), ID(ID), opKeyPrefixed(OpKey) {}
```

```
LongIdentifierExpression *
LongIdentifierExpression::asLongIdentifierExpression() {
  return this;
}

void LongIdentifierExpression::accept(Visitor *V) { V->visit(this); }

TupleExpression::TupleExpression(SourceLocation L,
                                 std::shared_ptr<SeqExpression> Es)
    : Expression(L), Expressions(Es) {
  assert(Es->size() != 1);
}

TupleExpression *TupleExpression::asTupleExpression() { return this; }

void TupleExpression::accept(Visitor *V) { V->visit(this); }

ListExpression::ListExpression(SourceLocation L,
                               std::shared_ptr<SeqExpression> Es)
    : Expression(L), Expressions(Es) {}

void ListExpression::accept(Visitor *V) { V->visit(this); }

IdentifierEqualsExpression::IdentifierEqualsExpression(
    SourceLocation L, std::shared_ptr<Identifier> LHS,
    std::shared_ptr<Expression> RHS)
    : Expression(L), Iden(LHS), Expr(RHS) {}

void IdentifierEqualsExpression::accept(Visitor *V) { V->visit(this); }

RecordExpression::RecordExpression(
    SourceLocation L, std::shared_ptr<IdentifierEqualsExpression> F)
    : Expression(L),
      std::vector<std::shared_ptr<IdentifierEqualsExpression>>{F} {}

void RecordExpression::accept(Visitor *V) { V->visit(this); }

SelectorExpression::SelectorExpression(SourceLocation L,
                                       std::shared_ptr<Identifier> ID)
    : Expression(L), ID(ID) {}

void SelectorExpression::accept(Visitor *V) { V->visit(this); }

ApplyExpression::ApplyExpression(SourceLocation L,
                                 std::shared_ptr<Expression> F)
    : Expression(L), std::vector<std::shared_ptr<Expression>>{F} {}

void ApplyExpression::accept(Visitor *V) { V->visit(this); }

TypeExpression::TypeExpression(SourceLocation L, std::shared_ptr<Expression> E,
                               std::shared_ptr<Type> T)
    : Expression(L), Expr(E), Typ(T) {}

void TypeExpression::accept(Visitor *V) { V->visit(this); }
```

```

OrElseExpression::OrElseExpression(SourceLocation L,
                                   std::shared_ptr<Expression> LHS,
                                   std::shared_ptr<Expression> RHS)
    : Expression(L), Left(LHS), Right(RHS) {}

void OrElseExpression::accept(Visitor *V) { V->visit(this); }

AndAlsoExpression::AndAlsoExpression(SourceLocation L,
                                     std::shared_ptr<Expression>
                                         LHS,
                                     std::shared_ptr<Expression>
                                         RHS)
    : Expression(L), Left(LHS), Right(RHS) {}

void AndAlsoExpression::accept(Visitor *V) { V->visit(this); }

LetExpression::LetExpression(SourceLocation L,
                             std::shared_ptr<SeqDeclaration> D,
                             std::shared_ptr<SeqExpression> E)
    : Expression(L), Declarations(D), Expressions(E) {}

void LetExpression::accept(Visitor *V) { V->visit(this); }

IfExpression::IfExpression(SourceLocation L,
    std::shared_ptr<Expression> C,
                           std::shared_ptr<Expression> T,
                           std::shared_ptr<Expression> E)
    : Expression(L), CondExpr(C), ThenExpr(T), ElseExpr(E) {}

void IfExpression::accept(Visitor *V) { V->visit(this); }

WhileExpression::WhileExpression(SourceLocation L,
                                 std::shared_ptr<Expression> C,
                                 std::shared_ptr<Expression> B)
    : Expression(L), CondExpr(C), BodyExpr(B) {}

void WhileExpression::accept(Visitor *V) { V->visit(this); }

CaseExpression::CaseExpression(SourceLocation L,
    std::shared_ptr<Expression> E,
                               std::shared_ptr<SeqMatch> M)
    : Expression(L), Expr(E), Cases(M) {}

void CaseExpression::accept(Visitor *V) { V->visit(this); }

LambdaExpression::LambdaExpression(SourceLocation L,
                                   std::shared_ptr<SeqMatch> M)
    : Expression(L), Cases(M) {}

void LambdaExpression::accept(Visitor *V) { V->visit(this); }
```

Listing 72: lib/AST/Pattern.cpp

```
#include "ssml/AST/Pattern.h"
#include "ssml/AST/Visitor.h"
#include "ssml/AST/Literal.h"
#include "ssml/AST/Identifier.h"
#include "ssml/AST/Type.h"

#include <cassert>
```

```
using namespace ssml::ast;

Pattern::Pattern(SourceLocation L) : Node(L) {}

TypePattern *Pattern::asTypePattern() { return nullptr; }

LongIdentifierPattern *Pattern::asLongIdentifierPattern() { return
    nullptr; }

ApplyPattern *Pattern::asApplyPattern() { return nullptr; }

SeqPattern::SeqPattern(SourceLocation L) : Pattern(L) {}

SeqPattern::SeqPattern(SourceLocation L, std::shared_ptr<Pattern>
    First)
    : Pattern(L), std::vector<std::shared_ptr<Pattern>>{First} {}

void SeqPattern::accept(Visitor *V) { V->visit(this); }

LiteralPattern::LiteralPattern(SourceLocation L,
    std::shared_ptr<Literal> Lit)
    : Pattern(L), Value(Lit) {}

void LiteralPattern::accept(Visitor *V) { V->visit(this); }

WildcardPattern::WildcardPattern(SourceLocation L) : Pattern(L) {}

void WildcardPattern::accept(Visitor *V) { V->visit(this); }

LongIdentifierPattern::LongIdentifierPattern(SourceLocation L,
                                             std::shared_ptr<LongIdentifier>
                                                 ID,
                                             bool opPrefixed)
    : Pattern(L), ID(ID), opKeyPrefixed(opPrefixed) {}

void LongIdentifierPattern::accept(Visitor *V) { V->visit(this); }

LongIdentifierPattern
    *LongIdentifierPattern::asLongIdentifierPattern() {
  return this;
}

TypePattern::TypePattern(SourceLocation L,
    std::shared_ptr<Pattern> P,
                         std::shared_ptr<Type> T)
    : Pattern(L), Pat(P), Typ(T) {}

TypePattern *TypePattern::asTypePattern() { return this; }

void TypePattern::accept(Visitor *V) { V->visit(this); }

AsPattern::AsPattern(SourceLocation L, std::shared_ptr<Pattern>
    LHS,
                     std::shared_ptr<Pattern> RHS)
    : Pattern(L), LeftPattern(LHS), RightPattern(RHS) {}

void AsPattern::accept(Visitor *V) { V->visit(this); }

ApplyPattern::ApplyPattern(SourceLocation L,
    std::shared_ptr<Pattern> First)
    : Pattern(L), std::vector<std::shared_ptr<Pattern>>{First} {}
```

```

void ApplyPattern::accept(Visitor *V) { V->visit(this); }

ApplyPattern *ApplyPattern::asApplyPattern() { return this; }

IdentifierEqualsPattern::IdentifierEqualsPattern(
    SourceLocation L, std::shared_ptr<Identifier> LHS,
    std::shared_ptr<Pattern> RHS)
    : Pattern(L), Iden(LHS), Pat(RHS) {}

void IdentifierEqualsPattern::accept(Visitor *V) { V->visit(this);
    }

ShortIdentifierPattern::ShortIdentifierPattern(
    SourceLocation L, std::shared_ptr<ShortIdentifier> ID)
    : Pattern(L), ID(ID) {}

void ShortIdentifierPattern::accept(Visitor *V) { V->visit(this); }

RecordPattern::RecordPattern(SourceLocation L,
    std::shared_ptr<Pattern> First)
    : Pattern(L), std::vector<std::shared_ptr<Pattern>>{First} {}

void RecordPattern::accept(Visitor *V) { V->visit(this); }

ListPattern::ListPattern(SourceLocation L,
    std::shared_ptr<SeqPattern> P)
    : Pattern(L), Patterns(P) {}

void ListPattern::accept(Visitor *V) { V->visit(this); }

TuplePattern::TuplePattern(SourceLocation L,
    std::shared_ptr<SeqPattern> P)
    : Pattern(L), Patterns(P) {
  assert(P->size() != 1);
}

void TuplePattern::accept(Visitor *V) { V->visit(this); }
```

Listing 73: lib/GCPrinter/SSMLGC.cpp

```
#include "llvm/CodeGen/GCs.h"
#include "llvm/CodeGen/GCStrategy.h"
#include "llvm/CodeGen/MachineInstrBuilder.h"
#include "llvm/MC/MCContext.h"
#include "llvm/MC/MCSymbol.h"
#include "llvm/Target/TargetInstrInfo.h"
#include "llvm/Target/TargetMachine.h"
#include "llvm/Target/TargetSubtargetInfo.h"

using namespace llvm;

namespace {
class SSMLGC : public GCStrategy {
public:
  SSMLGC();
};
}

static GCRegistry::Add<SSMLGC> X("ssml", "ssml garbage collector
    strategy");
```

```
SSMLGC::SSMLGC() {
  this->UsesMetadata = true;
  this->InitRoots = true;
  this->NeededSafePoints = 1 << GC::PostCall;
}
```

Listing 74: lib/GCPrinter/SSMLGCPrinter.cpp

```
#include "llvm/CodeGen/AsmPrinter.h"
#include "llvm/CodeGen/GCMetadataPrinter.h"
#include "llvm/CodeGen/GCs.h"
#include "llvm/IR/DataLayout.h"
#include "llvm/IR/Function.h"
#include "llvm/IR/Instruction.h"
#include "llvm/IR/IntrinsicInst.h"
#include "llvm/IR/Metadata.h"
#include "llvm/MC/MCAsmInfo.h"
#include "llvm/MC/MCContext.h"
#include "llvm/MC/MCSectionELF.h"
#include "llvm/MC/MCStreamer.h"
#include "llvm/MC/MCSymbol.h"
#include "llvm/Target/TargetLoweringObjectFile.h"
#include "llvm/Target/TargetMachine.h"
#include "llvm/Target/TargetSubtargetInfo.h"

#include <stdint.h>

using namespace llvm;

namespace {

class SSMLGCPrinter : public GCMetadataPrinter {
public:
  void finishAssembly(Module &M, GCModuleInfo &Info, AsmPrinter
      &AP) override;
};
}

static GCMetadataPrinterRegistry::Add<SSMLGCPrinter>
    X("ssml", "ssml garbage collector strategy");

void SSMLGCPrinter::finishAssembly(Module &M, GCModuleInfo &Info,
                                   AsmPrinter &AP) {
  MCStreamer &OS = *AP.OutStreamer;
  unsigned IntPtrSize = M.getDataLayout().getPointerSize();

  // Add end of code label.
  AP.OutStreamer->SwitchSection(AP.getObjFileLowering().getTextSection());
  MCSymbol *CodeEndLabel =
      AP.OutContext.getOrCreateSymbol("code_end");
  AP.OutStreamer->EmitLabel(CodeEndLabel);

  // Put GC Map in a custom section.
  OS.SwitchSection(AP.getObjFileLowering().getContext().getELFSection(
      "ssmlgcmap", ELF::SHT_PROGBITS, ELF::SHF_ALLOC));

#if 0
  MCSymbol *GCLabel =
      AP.OutContext.getOrCreateSymbol("ssml_gcmap");
  AP.OutStreamer->EmitSymbolAttribute(GCLabel, MCSA_Global);
```

```
  AP.OutStreamer->EmitLabel(GCLabel);
#endif

  std::vector<MCSymbol *> TableSyms;
  std::string SymPrefix("gcmap_sym_");
  size_t SymIdx = 0;

  // For each function...
  for (GCModuleInfo::FuncInfoVec::iterator FI = Info.funcinfo_begin(),
                                            IE = Info.funcinfo_end();
       FI != IE; ++FI) {
    GCFunctionInfo &MD = **FI;
    if (MD.getStrategy().getName() != getStrategy().getName())
      continue; // this function is managed by some other GC

    /** A compact GC layout. Emit this data structure:
     *
     * struct {
     *   int16_t StackFrameSize; (in words)
     *   int16_t LiveCount;
     *   int16_t LiveOffsets[LiveCount];
     * } gcmap_sym_${SymIdx};
     **/

    AP.EmitAlignment(3);

    MCSymbol *GCMapSym =
        AP.OutContext.getOrCreateSymbol(SymPrefix + std::to_string(SymIdx));
    ++SymIdx;
    AP.OutStreamer->EmitLabel(GCMapSym);
    TableSyms.push_back(GCMapSym);

    // Stack information never change in safe points! Only print info from the
    // first call-site.
    GCFunctionInfo::iterator PI = MD.begin();

    // Emit the stack frame size.
    OS.AddComment("stack frame size (in words)");
    AP.EmitInt16(MD.getFrameSize() / IntPtrSize);

    // Emit the number of live roots in the function.
    OS.AddComment("live root count");
    AP.EmitInt16(MD.live_size(PI));

    // And for each live root...
    for (GCFunctionInfo::live_iterator LI = MD.live_begin(PI),
                                       LE = MD.live_end(PI);
         LI != LE; ++LI) {
      // Emit live root's offset within the stack frame.
      OS.AddComment("stack index (offset / wordsize)");
      AP.EmitInt16(LI->StackOffset / IntPtrSize);
    }
  }

  // Put lookup table in custom section.
  OS.SwitchSection(AP.getObjFileLowering().getContext().getELFSection(
      "ssmlgclookup", ELF::SHT_PROGBITS, ELF::SHF_ALLOC));

```

```
  AP.EmitAlignment(3);
  MCSymbol *SizeLabel =
      AP.OutContext.getOrCreateSymbol("ssml_gclookup_size");
  AP.OutStreamer->EmitSymbolAttribute(SizeLabel, MCSA_Global);
  AP.OutStreamer->EmitLabel(SizeLabel);

  // Emit number of functions.
  OS.AddComment("function count");
  assert(TableSyms.size() <= INT16_MAX);
  AP.EmitInt16(TableSyms.size());

  AP.EmitAlignment(3);
  MCSymbol *MapLabel =
      AP.OutContext.getOrCreateSymbol("ssml_gclookup_map");
  AP.OutStreamer->EmitSymbolAttribute(MapLabel, MCSA_Global);
  AP.OutStreamer->EmitLabel(MapLabel);

  size_t Idx = 0;
  for (GCModuleInfo::FuncInfoVec::iterator FI =
      Info.funcinfo_begin(),
                                              IE =
                                                  Info.funcinfo_end();
       FI != IE; ++FI) {
    GCFunctionInfo &MD = **FI;
    MCSymbol *NextFun = AP.getSymbol(&MD.getFunction());
    AP.EmitLabelReference(NextFun, 8);
    AP.EmitLabelReference(TableSyms[Idx], 8);
    ++Idx;
  }
  AP.EmitLabelReference(CodeEndLabel, 8);
}
```

Listing 75: lib/Typecheck/Typedef.h

```
#ifndef LLVM_TOOLS_SSML_LIB_TYPECHECK_TYPEDEF_H
#define LLVM_TOOLS_SSML_LIB_TYPECHECK_TYPEDEF_H

#include "ssml/Common/SourceLocation.h"

#include "Typemap.h"

#include <string>
#include <memory>
#include <vector>

namespace ssml {
namespace typecheck {
class Typedef {
public:
  enum Kind {
    SimpleBuildin,
    Product,
    Function,
    Tuple,
    Datatype,
    Variable,
    DatatypeInstance
  };

  using VartypeVector =
      std::vector<std::pair<std::string,
```

```
          std::shared_ptr<Typedef>>>;

public:
  virtual ~Typedef();
  virtual Typedef *clone() const = 0;
  virtual Kind getKind() const = 0;
  virtual void setArgument(SourceLocation,
      std::shared_ptr<Typedef>) = 0;
  virtual void unmarkInScope() = 0;
  virtual std::shared_ptr<Typedef>
      funApplication(SourceLocation, std::shared_ptr<Typedef>)
          const;
  virtual VartypeVector getVariableTypes(SourceLocation,
                                          std::shared_ptr<Typedef>)
                                              const = 0;
  virtual bool canBeAssignWith(const Typedef &, const
      VartypeVector *,
                               bool ScopedVarIsVar) const;
  virtual bool cannotBeAssignWith(const Typedef &, const
      VartypeVector *,
                                  bool ScopedVarIsVar) const;
  virtual bool isTypeComplete() const = 0;
  virtual bool isTypeMissingArgument() const;
  virtual bool isType() const;
  virtual bool operator==(const Typedef &) const;
  virtual bool operator!=(const Typedef &) const;
  virtual bool hasVariable() const = 0;
  virtual std::shared_ptr<Typedef>
  withoutVariableTypedefs(const VartypeVector &) = 0;
};
} // End namespace typecheck.
} // End namespace ssml.

namespace ssml {
namespace typecheck {
class SimpleBuildinTypedef : public Typedef {
private:
  std::string Typename;

public:
  SimpleBuildinTypedef(std::string Name);
  SimpleBuildinTypedef *clone() const override;
  Typedef::Kind getKind() const final;
  void setArgument(SourceLocation, std::shared_ptr<Typedef>)
      override;
  void unmarkInScope() override;
  Typedef::VartypeVector getVariableTypes(SourceLocation,
                                           std::shared_ptr<Typedef>)
                                               const;
  bool canBeAssignWith(const Typedef &, const
      Typedef::VartypeVector *,
                       bool ScopedVarIsVar) const override;
  bool operator==(const Typedef &) const override;
  bool isTypeComplete() const override;
  bool hasVariable() const override;
  std::shared_ptr<Typedef>
  withoutVariableTypedefs(const VartypeVector &) override;
};
} // End namespace typecheck.
} // End namespace ssml.

namespace ssml {
```

```
namespace typecheck {
class ProductTypedef : public Typedef {
protected:
  std::vector<std::shared_ptr<Typedef>> Typedefs;

public:
  ProductTypedef *clone() const override;
  Typedef::Kind getKind() const override;
  void setArgument(SourceLocation, std::shared_ptr<Typedef>)
      override;
  void unmarkInScope() override;
  Typedef::VartypeVector getVariableTypes(SourceLocation,
                                          std::shared_ptr<Typedef>)
                                              const;
  bool canBeAssignWith(const Typedef &, const
      Typedef::VartypeVector *,
                       bool ScopedVarIsVar) const override;
  bool operator==(const Typedef &) const override;
  virtual void append(std::shared_ptr<Typedef>);
  bool isTypeComplete() const override;
  bool hasVariable() const override;

  const std::vector<std::shared_ptr<Typedef>> &getTypedefs() const
      {
    return this->Typedefs;
  }
  std::shared_ptr<Typedef>
  withoutVariableTypedefs(const VartypeVector &) override;
};
} // End namespace typecheck.
} // End namespace ssml.

namespace ssml {
namespace typecheck {
class FunctionTypedef : public ProductTypedef {
private:
  mutable Typedef::VartypeVector ApplicationVarTypes;
  bool HasCreatedAppendFunc = false;

public:
  FunctionTypedef *clone() const override;
  Typedef::Kind getKind() const final;
  bool canBeAssignWith(const Typedef &, const
      Typedef::VartypeVector *,
                       bool ScopedVarIsVar) const override;
  std::shared_ptr<Typedef>
  withoutVariableTypedefs(const VartypeVector &) override;
  std::shared_ptr<Typedef>
      funApplication(SourceLocation, std::shared_ptr<Typedef>)
          const override;
  void append(std::shared_ptr<Typedef>) override;
};
} // End namespace typecheck.
} // End namespace ssml.

namespace ssml {
namespace typecheck {
class TupleTypedef : public Typedef {
private:
  std::vector<std::shared_ptr<Typedef>> Typedefs;

public:
```

```
  TupleTypedef *clone() const override;
  Typedef::Kind getKind() const final;
  void append(std::shared_ptr<Typedef>);
  void setArgument(SourceLocation, std::shared_ptr<Typedef>)
      override;
  void unmarkInScope() override;
  Typedef::VartypeVector getVariableTypes(SourceLocation,
                                          std::shared_ptr<Typedef>)
                                              const;
  bool hasVariable() const override;
  bool canBeAssignWith(const Typedef &, const
      Typedef::VartypeVector *,
                       bool ScopedVarIsVar) const override;
  bool operator==(const Typedef &) const override;
  bool isTypeComplete() const override;
  bool isType() const override;

  const std::vector<std::shared_ptr<Typedef>> getTypedefs() const {
    return this->Typedefs;
  }
  std::shared_ptr<Typedef>
  withoutVariableTypedefs(const VartypeVector &) override;
};
} // End namespace typecheck.
} // End namespace ssml.

namespace ssml {
namespace typecheck {
class DatatypeTypedef : public Typedef {
public:
  std::string Typename;
  std::vector<std::pair<std::string, std::shared_ptr<Typedef>>>
      Params;
  bool IsArgumentSet = false;

public:
  DatatypeTypedef(std::string Name, std::vector<std::string>
      ParamNames);
  DatatypeTypedef(
      std::string Name,
      const std::vector<std::pair<std::string,
          std::shared_ptr<Typedef>>>
          &ParamNames);
  DatatypeTypedef *clone() const override;
  Typedef::Kind getKind() const final;
  void setArgument(SourceLocation, std::shared_ptr<Typedef>)
      override;
  void unmarkInScope() override;
  void setVariableType(const std::string &Name,
      std::shared_ptr<Typedef> T);
  Typedef::VartypeVector getVariableTypes(SourceLocation,
                                          std::shared_ptr<Typedef>)
                                              const;
  bool canBeAssignWith(const Typedef &, const
      Typedef::VartypeVector *,
                       bool ScopedVarIsVar) const override;
  bool hasVariable() const override;
  bool operator==(const Typedef &) const override;
  bool isTypeComplete() const override;
  bool isTypeMissingArgument() const override;
  std::shared_ptr<Typedef>
  withoutVariableTypedefs(const VartypeVector &) override;
```

```
  const std::vector<std::pair<std::string, std::shared_ptr<Typedef>>> &
  getParams() const {
    return this->Params;
  }

  const std::string &getUserTypename() const { return this->Typename; }
};
} // End namespace typecheck.
} // End namespace ssml.

namespace ssml {
namespace typecheck {
extern std::shared_ptr<DatatypeTypedef> getListTypedef();
extern std::shared_ptr<DatatypeTypedef> getRefTypedef();
extern std::shared_ptr<DatatypeTypedef> getArrayTypedef();
} // End namespace typecheck.
} // End namespace ssml.

namespace ssml {
namespace typecheck {
class VariableTypedef : public Typedef {
private:
  std::string UserTypename;
  std::string InternTypename;
  bool IsInScope = false;

public:
  VariableTypedef(std::string UserName, std::string InternName);
  VariableTypedef *clone() const override;
  Typedef::Kind getKind() const final;
  void setArgument(SourceLocation, std::shared_ptr<Typedef>) override;
  void unmarkInScope() override;
  Typedef::VartypeVector getVariableTypes(SourceLocation,
                                          std::shared_ptr<Typedef>) const;
  bool canBeAssignWith(const Typedef &, const Typedef::VartypeVector *,
                       bool ScopedVarIsVar) const override;
  bool hasVariable() const override;
  bool operator==(const Typedef &) const override;
  bool isTypeComplete() const override;
  void setIsInScope(bool B) { this->IsInScope = B; }
  bool isInScope() const { return this->IsInScope; }
  std::shared_ptr<Typedef>
  withoutVariableTypedefs(const VartypeVector &) override;

  const std::string &getUserTypename() const { return this->UserTypename; }
  const std::string &getInternTypename() const { return this->InternTypename; }
};
} // End namespace typecheck.
} // End namespace ssml.

namespace ssml {
namespace typecheck {
class DatatypeInstanceTypedef : public Typedef {
private:
```

```
  DatatypeTypedef Datatypedef;
  std::shared_ptr<Typedef> ArgumentTypedef;
  std::shared_ptr<Typedef> Argument;

public:
  DatatypeInstanceTypedef(const DatatypeTypedef &Datatype,
                          std::shared_ptr<Typedef> Argument =
                              nullptr);
  DatatypeInstanceTypedef *clone() const override;
  Typedef::Kind getKind() const final;
  void setArgument(SourceLocation, std::shared_ptr<Typedef>)
      override;
  void unmarkInScope() override;
  std::shared_ptr<Typedef>
      funApplication(SourceLocation, std::shared_ptr<Typedef>)
          const override;
  Typedef::VartypeVector getVariableTypes(SourceLocation,
                                          std::shared_ptr<Typedef>)
                                              const;
  bool canBeAssignWith(const Typedef &, const
      Typedef::VartypeVector *,
                       bool ScopedVarIsVar) const override;
  bool operator==(const Typedef &) const override;
  bool isTypeComplete() const override;
  bool isTypeMissingArgument() const override;
  bool hasVariable() const override;
  std::shared_ptr<Typedef>
  withoutVariableTypedefs(const VartypeVector &) override;

  const DatatypeTypedef &getDatatypedef() const { return
      this->Datatypedef; }
  std::shared_ptr<Typedef> getArgumentTypedef() {
    return this->ArgumentTypedef;
  }
};
} // End namespace typecheck.
} // End namespace ssml.

#endif // LLVM_TOOLS_SSML_LIB_TYPECHECK_TYPEDEF_H
```

Listing 76: lib/Typecheck/Typecheck.cpp

```
#include "ssml/Typecheck/Typecheck.h"

#include "TypecheckVisitor.h"

#include "ssml/AST/Declaration.h"

using namespace ssml::typecheck;

void ssml::typecheck::typecheck(ssml::ast::Root *R) {
  TypecheckVisitor V;
  R->accept(&V);
}
```

Listing 77: lib/Typecheck/Typemap.h

```
#ifndef LLVM_TOOLS_SSML_LIB_TYPECHECK_TYPEMAP_H
#define LLVM_TOOLS_SSML_LIB_TYPECHECK_TYPEMAP_H

#include "ssml/Common/Compare.h"
```

```

#include <string>
#include <memory>
#include <map>
#include <vector>

namespace ssml {
namespace typecheck {
class Typedef;
} // End namespace typecheck.
} // End namespace ssml.

namespace ssml {
namespace typecheck {
class Typemap {
public:
  using ValueType = std::shared_ptr<Typedef>;
  using MapType = std::map<std::string, ValueType>;

private:
  std::vector<MapType> Maps;

public:
  void insert(std::string K, ValueType V, size_t Scope);
  ValueType get(const std::string &K) const;
  void mergeWith(MapType, size_t Scope);
  void enterScope();
  void leaveScope();
  void erase(const std::string &S, size_t Scope) {
    Maps[Maps.size() - 1 - Scope].erase(S);
  }
  void erase(const MapType &M, size_t Scope) {
    for (auto &&P : M)
      this->erase(P.first, Scope);
  }
};
} // End namespace typecheck.
} // End namespace ssml.

#endif // LLVM_TOOLS_SSML_LIB_TYPECHECK_TYPEMAP_H
```

Listing 78: lib/Typecheck/TypecheckVisitor.cpp

```
#define DEBUG_TYPE "Typecheck"

#include "TypecheckVisitor.h"
#include "Patternmatch.h"

#include "ssml/AST/All.h"
#include "ssml/Common/ErrorMessages.h"
#include "ssml/Common/FatalExit.h"

#include "llvm/Support/raw_ostream.h"
#include "llvm/Support/Debug.h"

#define TYPECHECK_DLOG(msg)                                              \
  DEBUG(llvm::errs() << DEBUG_TYPE ":: " << msg << '\n')

using namespace ssml;
using namespace ssml::ast;
```

```
using namespace ssml::typecheck;

std::shared_ptr<Typedef> TypecheckVisitor::getPrevPatternType() {
  assert(!!this->PrevPatternmatch);
  auto T = this->PrevPatternmatch->getPreeMatchPatternType();
  if (T && T->isTypeComplete())
    return T;
  return nullptr;
}

bool TypecheckVisitor::cannotBeAssignWith(std::shared_ptr<Typedef> DestType,
                                          SourceLocation SourceL,
                                          std::shared_ptr<Typedef> SourceType,
                                          bool ExprHasValApplication,
                                          bool ScopedVarIsVar,
                                          bool IsRet) {
  bool B;
  if (SourceType->getKind() == Typedef::Function) {
    auto VarTs = SourceType->getVariableTypes(SourceL, DestType);
    B = DestType->cannotBeAssignWith(*SourceType, &VarTs, ScopedVarIsVar);
  } else {
    B = DestType->cannotBeAssignWith(*SourceType, nullptr, ScopedVarIsVar);
  }

  if (ExprHasValApplication && !IsRet)
    if (DestType->hasVariable())
      errorExit(SourceL, "expression involving function application cannot "
                         "be of variable type");

  return B;
}

void TypecheckVisitor::enterScope() {
  static unsigned ScopeCount = 0;
  if (ScopeCount)
    this->TypenamePrefix =
        "?" + std::to_string(ScopeCount) + "." + this->TypenamePrefix;
  ++ScopeCount;
  this->TypenameTypes.enterScope();
  this->ValnameTypes.enterScope();
  this->ConstnameTypes.enterScope();
}

void TypecheckVisitor::leaveScope() {
  auto Idx = this->TypenamePrefix.find_first_of(".");
  if (Idx != std::string::npos)
    this->TypenamePrefix = this->TypenamePrefix.substr(Idx + 1);
  this->TypenameTypes.leaveScope();
  this->ValnameTypes.leaveScope();
  this->ConstnameTypes.leaveScope();
}

void TypecheckVisitor::doVisit(std::shared_ptr<ssml::ast::Node> N) {
  this->doVisit(N.get());
```

```
}

void TypecheckVisitor::doVisit(ssml::ast::Node *N) {
  this->PrevType.reset();
  this->PrevPatternmatch.reset();
  N->accept(this);
}

void TypecheckVisitor::visit(IntLiteral *N) {
  this->PrevType = this->IntTypedef;
}

void TypecheckVisitor::visit(ShortIdentifier *N) {
  fatalExit("TypecheckVisitor visit ShortIdentifier");
}

void TypecheckVisitor::visit(LongIdentifier *N) {
  fatalExit("TypecheckVisitor visit LongIdentifier");
}

void TypecheckVisitor::visit(SeqShortIdentifier *N) {
  fatalExit("TypecheckVisitor visit SeqShortIdentifier");
}

void TypecheckVisitor::visit(SeqLongIdentifier *N) {
  fatalExit("TypecheckVisitor visit SeqLongIdentifier");
}

void TypecheckVisitor::visit(SeqExpression *N) {
  for (auto C : *N)
    this->doVisit(C);
  // this->PrevType is set to the last expression in the list:
  // let ... in ... end
}

void TypecheckVisitor::visit(LiteralExpression *N) {
  this->doVisit(N->getValue());
  // this->PrevType is set.
  this->ApplyExprWasVal = true;
}

void TypecheckVisitor::visit(LongIdentifierExpression *N) {
  this->PrevType = this->ValnameTypes.get(N->getIDs()->toString());
  if (!this->PrevType)
    this->PrevType =
        this->ConstnameTypes.get(N->getIDs()->toString());
  else
    this->ApplyExprWasVal = true;
  if (!this->PrevType)
    errorExit(N->getIDs()->getLocation(), "undeclared identifier");
}

void TypecheckVisitor::visit(TupleExpression *N) {
  auto T = std::make_shared<ProductTypedef>();
  for (auto C : *N->getExprs()) {
    this->doVisit(C);
    T->append(this->PrevType);
  }
  this->PrevType = T;
  this->ApplyExprWasVal = true;
}

```

```
void TypecheckVisitor::visit(ListExpression *N) {
  auto List = getListTypedef();
  std::shared_ptr<Typedef> ListType;
  for (auto E : *N->getExprs()) {
    this->doVisit(E);
    if (ListType) {
      if (this->cannotBeAssignWith(ListType, N->getLocation(),
          this->PrevType,
                                   false, false) &&
          this->cannotBeAssignWith(this->PrevType,
              N->getLocation(), ListType,
                                   false, false))
        errorExit(E->getLocation(), "contradicting expression
            types in list");
    }
    if (!ListType || this->PrevType->isTypeComplete())
      ListType = this->PrevType;
  }
  if (ListType)
    List->setArgument(N->getLocation(), ListType);
  this->PrevType = List;
  this->ApplyExprWasVal = true;
}

void TypecheckVisitor::visit(ApplyExpression *N) {
  size_t Count = 0;
  std::shared_ptr<Typedef> Fun;
  bool FirstWasValname = false;
  for (auto C : *N) {
    this->ApplyExprWasVal = false;
    this->doVisit(C);
    if (!this->PrevType->isType())
      fatalExit("Expression prev type is not a real type");

    if (Count) {
      if (!this->ApplyExprWasVal &&
          this->PrevType->isTypeMissingArgument())
        errorExit(C->getLocation(), "missing argument");
      Fun = Fun->funApplication(C->getLocation(), this->PrevType);
    } else {
      Fun = this->PrevType;
      FirstWasValname = this->ApplyExprWasVal;
    }
    ++Count;
  }
  if (Fun->getKind() == Typedef::DatatypeInstance &&
      Fun->isTypeMissingArgument())
    errorExit(N->getLocation(), "missing argument");

  this->PrevType = Fun;
  if (FirstWasValname && Count > 1)
    this->ExprHasValnameApply = true;
}

void TypecheckVisitor::visit(OrElseExpression *N) {
  this->doVisit(N->getLeftExpr());
  if (*this->PrevType != *this->BoolTypedef)
    errorExit(N->getLeftExpr()->getLocation(), "unexpected
        expression type");
  this->doVisit(N->getRightExpr());
  if (*this->PrevType != *this->BoolTypedef)
    errorExit(N->getLeftExpr()->getLocation(), "unexpected
```

```
        expression type");
  this->PrevType.reset(this->BoolTypedef->clone());
  this->ApplyExprWasVal = true;
}

void TypecheckVisitor::visit(AndAlsoExpression *N) {
  this->doVisit(N->getLeftExpr());
  if (*this->PrevType != *this->BoolTypedef)
    errorExit(N->getLeftExpr()->getLocation(), "unexpected
        expression type");
  this->doVisit(N->getRightExpr());
  if (*this->PrevType != *this->BoolTypedef)
    errorExit(N->getLeftExpr()->getLocation(), "unexpected
        expression type");
  this->PrevType.reset(this->BoolTypedef->clone());
  this->ApplyExprWasVal = true;
}

void TypecheckVisitor::visit(LetExpression *N) {
  this->enterScope();
  this->doVisit(N->getDecls());
  this->doVisit(N->getExprs());
  // this->PrevType is set.
  this->leaveScope();
  // this->ApplyExprWasVal is set.
}

void TypecheckVisitor::visit(IfExpression *N) {
  this->doVisit(N->getCondExpr());
  if (*this->PrevType != *this->BoolTypedef)
    errorExit(N->getCondExpr()->getLocation(), "unexpected
        expression type");
  this->doVisit(N->getThenExpr());
  auto Ret = this->PrevType;
  this->doVisit(N->getElseExpr());
  if (this->cannotBeAssignWith(this->PrevType,
      N->getThenExpr()->getLocation(),
                               Ret, false, false) &&
      this->cannotBeAssignWith(Ret,
          N->getElseExpr()->getLocation(),
                               this->PrevType, false, false))
    errorExit(N->getLocation(), "conflicting if expression types");

  this->PrevType = Ret;
  this->ApplyExprWasVal = true;
}

void TypecheckVisitor::visit(WhileExpression *N) {
  this->doVisit(N->getCondExpr());
  if (*this->PrevType != *this->BoolTypedef)
    errorExit(N->getCondExpr()->getLocation(), "unexpected
        expression type");
  this->doVisit(N->getBodyExpr());
  this->PrevType.reset(this->UnitTypedef->clone());
  this->ApplyExprWasVal = true;
}

void TypecheckVisitor::visit(Match *N) {
  bool PrevAdd = this->AddVariableTypeToTypenames;

  this->AddVariableTypeToTypenames = true;
  this->doVisit(N->getPattern());
```

```
  auto Pat = this->PrevPatternmatch;
  auto NameTypes = Pat->getNameTypes();

  for (auto &S : this->PatternIdenInsertionOrder) {
    assert(NameTypes.count(S));
    auto Def = NameTypes[S];
    if (Def->getKind() == Typedef::SimpleBuildin)
      this->peekFuncVals()->addVal(false);
    else
      this->peekFuncVals()->addVal(true);
  }
  this->PatternIdenInsertionOrder.clear();

  this->enterScope();
  this->ValnameTypes.mergeWith(NameTypes, 0);
  this->ConstnameTypes.erase(NameTypes, 0);

  this->doVisit(N->getExpr());
  // this->PrevType is set.
  this->PrevPatternmatch = Pat;

  this->AddVariableTypeToTypenames = PrevAdd;
  this->leaveScope();
}

void TypecheckVisitor::visit(SeqMatch *N) {
  fatalExit("TypecheckVisitor visit SeqMatch");
}

void TypecheckVisitor::visit(LambdaExpression *N) {
  bool PrevHasApply = this->ExprHasValnameApply;

  this->pushFuncVals();
  N->setFunctionVals(this->peekFuncVals());
  this->enterScope();

  std::shared_ptr<Typedef> PatternType;
  std::shared_ptr<Typedef> CaseType;
  SourceLocation CaseTypeLocation;
  SourceLocation PatternTypeLocation;
  auto Matches = N->getCases();
  for (auto &&M : *Matches) {
    this->ExprHasValnameApply = false;
    this->doVisit(M);
    auto T = this->PrevType;
    auto PatType = this->getPrevPatternType();
    if (!PatType)
      errorExit(M->getLocation(), "pattern missing type
          specification");
    if (CaseType &&
        this->cannotBeAssignWith(T, CaseTypeLocation, CaseType,
            false, true) &&
        this->cannotBeAssignWith(CaseType, M->getLocation(),
            this->PrevType,
                                 false, true))
      errorExit(N->getLocation(), "conflicting fn case types");
    bool HasApply = this->ExprHasValnameApply;
    if (PatternType &&
        this->cannotBeAssignWith(PatType, PatternTypeLocation,
            PatternType,
                                 HasApply, false) &&
        this->cannotBeAssignWith(PatternType, M->getLocation(),
```

```
            PatType,
                                    HasApply, false))
      errorExit(M->getLocation(), "conflicting fn pattern types");
    if (!PatternType) {
      PatternType = PatType;
      PatternTypeLocation = M->getLocation();
    }
    if (!CaseType || (!CaseType->isTypeComplete() &&
        T->isTypeComplete())) {
      CaseType = T;
      CaseTypeLocation = M->getLocation();
    }
  }

  auto Fun = std::make_shared<FunctionTypedef>();
  Fun->append(PatternType);
  Fun->append(CaseType);
  this->PrevType = Fun;
  this->ApplyExprWasVal = true;
  this->ExprHasValnameApply = PrevHasApply;

  this->leaveScope();
  this->popFuncVals();
}

void TypecheckVisitor::visit(SeqPattern *N) {
  for (auto C : *N)
    this->doVisit(C);
  if (N->size() > 1)
    this->PrevType.reset();
}

void TypecheckVisitor::visit(LiteralPattern *N) {
  this->doVisit(N->getValue());
  auto T = this->PrevType;
  this->PrevPatternmatch.reset(
      new LiteralPatternmatch(N->getValue()->getLocation(), T));
  this->PrevType.reset();
}

void TypecheckVisitor::visit(WildcardPattern *N) {
  this->PrevPatternmatch.reset(new
      WildcardPatternmatch(N->getLocation()));
  this->PrevType.reset();
}

void TypecheckVisitor::visit(LongIdentifierPattern *N) {
  if (!this->ConstnameTypes.get(N->getIDs()->toString()))
    this->PatternIdenInsertionOrder.push_back(N->getIDs()->toString());
  this->PrevPatternmatch.reset(new LongIdentifierPatternmatch(
      N->getLocation(), this->ConstnameTypes,
          N->getIDs()->toString()));
  this->PrevType.reset();
}

void TypecheckVisitor::visit(TypePattern *N) {
  this->doVisit(N->getPattern());
  auto Match = this->PrevPatternmatch;
  this->doVisit(N->getType());
  if (!this->PrevType->isType())
    errorExit(N->getLocation(), "expected type");
  Match->matchType(this->PrevType);
```

```
  this->PrevPatternmatch = Match;
  // this->PrevType is set correctly at this point.
  // this->PrevType can be used to verify that this node has been
      visited.
}

void TypecheckVisitor::visit(ApplyPattern *N) {
  int Count = 0;
  std::shared_ptr<Patternmatch> First;
  for (auto C : *N) {
    this->doVisit(C);
    switch (Count) {
    case 0:
      First = this->PrevPatternmatch;
      break;
    case 1:
      this->PrevPatternmatch.reset(new ApplyPatternmatch(
          N->getLocation(), C->getLocation(), First,
              this->PrevPatternmatch));
      break;
    default:
      errorExit(C->getLocation(), "invalid argument [1]");
    }
    ++Count;
  }
  assert(!!this->PrevPatternmatch);
  this->PrevType.reset();
}

void TypecheckVisitor::visit(ListPattern *N) {
  auto Pats = std::make_shared<ListPatternmatch>(N->getLocation());
  for (auto P : *N->getPatterns()) {
    this->doVisit(P);
    Pats->append(this->PrevPatternmatch);
  }
  this->PrevPatternmatch = Pats;
  this->PrevType.reset();
}

void TypecheckVisitor::visit(TuplePattern *N) {
  auto Pats =
      std::make_shared<TuplePatternmatch>(N->getLocation());
  for (auto P : *N->getPatterns()) {
    this->doVisit(P);
    Pats->append(this->PrevPatternmatch);
  }
  this->PrevPatternmatch = Pats;
  this->PrevType.reset();
}

void TypecheckVisitor::visit(SeqDeclaration *N) {
  for (auto C : *N)
    this->doVisit(C);
}

void TypecheckVisitor::addDefaults() {
  // Add default type declarations:
  this->TypenameTypes.insert("int", this->IntTypedef,
      this->InsertionScope);
  this->TypenameTypes.insert("bool", this->BoolTypedef,
      this->InsertionScope);
  this->TypenameTypes.insert("unit", this->UnitTypedef,
```

```
        this->InsertionScope);
    auto ListDatatype = getListTypedef();
    this->TypenameTypes.insert("list", ListDatatype,
        this->InsertionScope);
    auto RefDatatype = getRefTypedef();
    this->TypenameTypes.insert("ref", RefDatatype,
        this->InsertionScope);
    auto ArrayDatatype = getArrayTypedef();
    this->TypenameTypes.insert("array", ArrayDatatype,
        this->InsertionScope);

    this->ConstnameTypes.insert("true", this->BoolTypedef,
        this->InsertionScope);
    this->ConstnameTypes.insert("false", this->BoolTypedef,
        this->InsertionScope);

    auto ConsOf = std::make_shared<ProductTypedef>();
    ConsOf->append(
        std::make_shared<VariableTypedef>("'a", this->TypenamePrefix
            + "'a"));
    auto ConsList = getListTypedef();
    ConsList->setArgument({0, 0}, std::make_shared<VariableTypedef>(
                                    "'a", this->TypenamePrefix +
                                        "'a"));
    ConsOf->append(ConsList);

    this->ConstnameTypes.insert(
        "::",
            std::make_shared<DatatypeInstanceTypedef>(*ListDatatype,
            ConsOf),
        this->InsertionScope);
    this->ConstnameTypes.insert(
        "nil",
            std::make_shared<DatatypeInstanceTypedef>(*ListDatatype),
        this->InsertionScope);

    // Add default function declarations:
    {
      auto PrintType = std::make_shared<FunctionTypedef>();
      PrintType->append(
          std::make_shared<VariableTypedef>("'a",
              this->TypenamePrefix + "'a"));
      PrintType->append(this->UnitTypedef);
      this->ValnameTypes.insert("print", PrintType,
          this->InsertionScope);
    }
    {
      auto PlusType = std::make_shared<FunctionTypedef>();
      auto Tup = std::make_shared<ProductTypedef>();
      Tup->append(this->IntTypedef);
      Tup->append(this->IntTypedef);
      PlusType->append(Tup);
      PlusType->append(this->IntTypedef);
      this->ValnameTypes.insert("+", PlusType, this->InsertionScope);
    }
    {
      auto Type = std::make_shared<FunctionTypedef>();
      auto Tup = std::make_shared<ProductTypedef>();
      Tup->append(this->IntTypedef);
      Tup->append(this->IntTypedef);
      Type->append(Tup);
      Type->append(this->IntTypedef);
```

```
    this->ValnameTypes.insert("-", Type, this->InsertionScope);
  }
  {
    auto Type = std::make_shared<FunctionTypedef>();
    auto Tup = std::make_shared<ProductTypedef>();
    Tup->append(this->IntTypedef);
    Tup->append(this->IntTypedef);
    Type->append(Tup);
    Type->append(this->IntTypedef);
    this->ValnameTypes.insert("*", Type, this->InsertionScope);
  }
  {
    auto Type = std::make_shared<FunctionTypedef>();
    auto Tup = std::make_shared<ProductTypedef>();
    Tup->append(this->IntTypedef);
    Tup->append(this->IntTypedef);
    Type->append(Tup);
    Type->append(this->IntTypedef);
    this->ValnameTypes.insert("div", Type, this->InsertionScope);
  }
  {
    auto Type = std::make_shared<FunctionTypedef>();
    auto Tup = std::make_shared<ProductTypedef>();
    Tup->append(this->IntTypedef);
    Tup->append(this->IntTypedef);
    Type->append(Tup);
    Type->append(this->IntTypedef);
    this->ValnameTypes.insert("mod", Type, this->InsertionScope);
  }
  {
    auto Type = std::make_shared<FunctionTypedef>();
    auto Tup = std::make_shared<ProductTypedef>();
    auto Ref = getRefTypedef();
    Ref->setArgument({0, 0}, std::make_shared<VariableTypedef>(
                                 "'a", this->TypenamePrefix +
                                    "'a"));
    Tup->append(Ref);
    Tup->append(
        std::make_shared<VariableTypedef>("'a",
            this->TypenamePrefix + "'a"));
    Type->append(Tup);
    Type->append(this->UnitTypedef);
    this->ValnameTypes.insert(":=", Type, this->InsertionScope);
  }
  {
    auto Type = std::make_shared<FunctionTypedef>();
    Type->append(this->IntTypedef);
    Type->append(this->IntTypedef);
    this->ValnameTypes.insert("~", Type, this->InsertionScope);
  }
  {
    auto Type = std::make_shared<FunctionTypedef>();
    Type->append(this->BoolTypedef);
    Type->append(this->BoolTypedef);
    this->ValnameTypes.insert("not", Type, this->InsertionScope);
  }
  {
    auto Type = std::make_shared<FunctionTypedef>();
    auto RefInstance = std::make_shared<DatatypeInstanceTypedef>(
        *getRefTypedef(),
        std::make_shared<VariableTypedef>("'a",
            this->TypenamePrefix + "'a"));
```

```
    RefInstance->setArgument({0, 0},
        std::make_shared<VariableTypedef>(
                                         "'a",
                                             this->TypenamePrefix
                                             + "'a"));
    Type->append(
        std::make_shared<VariableTypedef>("'a",
            this->TypenamePrefix + "'a"));
    Type->append(RefInstance);
    this->ValnameTypes.insert("ref", Type, this->InsertionScope);
  }
  {
    auto Type = std::make_shared<FunctionTypedef>();
    auto Tup = std::make_shared<ProductTypedef>();
    auto ArrayInstance = std::make_shared<DatatypeInstanceTypedef>(
        *getArrayTypedef(),
        std::make_shared<VariableTypedef>("'a",
            this->TypenamePrefix + "'a"));
    ArrayInstance->setArgument({0, 0},
        std::make_shared<VariableTypedef>(
                                           "'a",
                                               this->TypenamePrefix
                                               + "'a"));
    Tup->append(this->IntTypedef);
    Tup->append(
        std::make_shared<VariableTypedef>("'a",
            this->TypenamePrefix + "'a"));
    Type->append(Tup);
    Type->append(ArrayInstance);
    this->ValnameTypes.insert("Array.array", Type,
        this->InsertionScope);
  }
  {
    auto Type = std::make_shared<FunctionTypedef>();
    auto Tup = std::make_shared<ProductTypedef>();
    auto Ary = getArrayTypedef();
    Ary->setArgument({0, 0}, std::make_shared<VariableTypedef>(
                                "'a", this->TypenamePrefix +
                                    "'a"));

    Tup->append(Ary);
    Tup->append(this->IntTypedef);
    Tup->append(
        std::make_shared<VariableTypedef>("'a",
            this->TypenamePrefix + "'a"));
    Type->append(Tup);
    Type->append(this->UnitTypedef);
    this->ValnameTypes.insert("Array.update", Type,
        this->InsertionScope);
  }
  {
    auto Ary = getArrayTypedef();
    this->ValnameTypes.insert("Array.empty", Ary,
        this->InsertionScope);
  }
  {
    auto Type = std::make_shared<FunctionTypedef>();
    auto Tup = std::make_shared<ProductTypedef>();
    auto Ary = getArrayTypedef();
    Ary->setArgument({0, 0}, std::make_shared<VariableTypedef>(
                                "'a", this->TypenamePrefix +
                                    "'a"));
```

```
    Tup->append(Ary);
    Tup->append(this->IntTypedef);
    Type->append(Tup);
    Type->append(
        std::make_shared<VariableTypedef>("'a",
            this->TypenamePrefix + "'a"));
    this->ValnameTypes.insert("Array.get", Type,
        this->InsertionScope);
  }
  {
    auto Type = std::make_shared<FunctionTypedef>();
    auto Ary = getArrayTypedef();
    Ary->setArgument({0, 0}, std::make_shared<VariableTypedef>(
                                "'a", this->TypenamePrefix +
                                    "'a"));
    Type->append(Ary);
    Type->append(this->IntTypedef);
    this->ValnameTypes.insert("Array.length", Type,
        this->InsertionScope);
  }
  {
    auto Type = std::make_shared<FunctionTypedef>();
    auto RefInstance = std::make_shared<DatatypeInstanceTypedef>(
        *getRefTypedef(),
        std::make_shared<VariableTypedef>("'a",
            this->TypenamePrefix + "'a"));
    RefInstance->setArgument({0, 0},
        std::make_shared<VariableTypedef>(
                                            "'a",
                                                this->TypenamePrefix
                                                + "'a"));
    Type->append(RefInstance);
    Type->append(
        std::make_shared<VariableTypedef>("'a",
            this->TypenamePrefix + "'a"));
    this->ValnameTypes.insert("!", Type, this->InsertionScope);
  }
  {
    auto Type = std::make_shared<FunctionTypedef>();
    auto Tup = std::make_shared<ProductTypedef>();
    Tup->append(this->IntTypedef);
    Tup->append(this->IntTypedef);
    Type->append(Tup);
    Type->append(this->BoolTypedef);
    this->ValnameTypes.insert("=", Type, this->InsertionScope);
  }
  {
    auto Type = std::make_shared<FunctionTypedef>();
    auto Tup = std::make_shared<ProductTypedef>();
    Tup->append(this->IntTypedef);
    Tup->append(this->IntTypedef);
    Type->append(Tup);
    Type->append(this->BoolTypedef);
    this->ValnameTypes.insert("<>", Type, this->InsertionScope);
  }
  {
    auto Type = std::make_shared<FunctionTypedef>();
    auto Tup = std::make_shared<ProductTypedef>();
    Tup->append(this->IntTypedef);
    Tup->append(this->IntTypedef);
    Type->append(Tup);
```

```
    Type->append(this->BoolTypedef);
    this->ValnameTypes.insert("<", Type, this->InsertionScope);
  }
  {
    auto Type = std::make_shared<FunctionTypedef>();
    auto Tup = std::make_shared<ProductTypedef>();
    Tup->append(this->IntTypedef);
    Tup->append(this->IntTypedef);
    Type->append(Tup);
    Type->append(this->BoolTypedef);
    this->ValnameTypes.insert(">", Type, this->InsertionScope);
  }
  {
    auto Type = std::make_shared<FunctionTypedef>();
    auto Tup = std::make_shared<ProductTypedef>();
    Tup->append(this->IntTypedef);
    Tup->append(this->IntTypedef);
    Type->append(Tup);
    Type->append(this->BoolTypedef);
    this->ValnameTypes.insert("<=", Type, this->InsertionScope);
  }
  {
    auto Type = std::make_shared<FunctionTypedef>();
    auto Tup = std::make_shared<ProductTypedef>();
    Tup->append(this->IntTypedef);
    Tup->append(this->IntTypedef);
    Type->append(Tup);
    Type->append(this->BoolTypedef);
    this->ValnameTypes.insert(">=", Type, this->InsertionScope);
  }
}

void TypecheckVisitor::visit(Root *N) {
  this->pushFuncVals();
  N->setFunctionVals(this->peekFuncVals());
  this->enterScope();

  this->addDefaults();

  this->enterScope();
  for (auto C : *N)
    this->doVisit(C);

  this->leaveScope();
  this->leaveScope();
  this->popFuncVals();
}

void TypecheckVisitor::visit(ValDeclaration *N) {
  bool PrevHasApply = this->ExprHasValnameApply;

  this->doVisit(N->getDest());
  auto DestType = this->getPrevPatternType();
  if (!DestType)
    errorExit(N->getDest()->getLocation(),
              "pattern missing type specification");
  auto Match = this->PrevPatternmatch;
  auto NameTypes = Match->getNameTypes();

  for (auto &S : this->PatternIdenInsertionOrder) {
    assert(NameTypes.count(S));
    auto Def = NameTypes[S];
```

```
    if (Def->getKind() == Typedef::SimpleBuildin)
      this->peekFuncVals()->addVal(false);
    else
      this->peekFuncVals()->addVal(true);
  }
  this->PatternIdenInsertionOrder.clear();

  this->ExprHasValnameApply = false;
  this->doVisit(N->getSource());
  auto SourceType = this->PrevType;

  if (this->cannotBeAssignWith(DestType,
      N->getSource()->getLocation(),
                                   SourceType,
                                       this->ExprHasValnameApply,
                                       false))
    errorExit(N->getLocation(),
              "pattern and expression have conflicting types");

  this->ValnameTypes.mergeWith(NameTypes, 0);
  this->ConstnameTypes.erase(NameTypes, 0);

  this->ExprHasValnameApply = PrevHasApply;
}

void TypecheckVisitor::visit(NonfixDeclaration *N) { // Nop.
}

void TypecheckVisitor::visit(InfixDeclaration *N) { // Nop.
}

void TypecheckVisitor::visit(InfixRDeclaration *N) { // Nop.
}

static Typemap::MapType mergeParamNameTypes(SourceLocation L,
                                            const std::string
                                                &FunName,
                                            std::vector<Typemap::MapType>
                                                &V) {
  Typemap::MapType Ret;
  for (auto &&M : V) {
    for (auto &&P : M) {
      if (Ret.count(P.first) || P.first == FunName)
        errorExit(L, "multiple declarations of identifier '" +
            P.first + "'");
      Ret[P.first] = P.second;
    }
  }
  return Ret;
}

void TypecheckVisitor::visit(FunPatternDeclaration *N) {
  bool PrevHasApply = this->ExprHasValnameApply;
  bool PrevFunParam = this->AddVariableTypeToTypenames;

  auto IDParams = N->getIDParams();
  auto TypePat = IDParams->asTypePattern();
  if (!TypePat)
    errorExit(N->getLocation(), "missing function return type");

  auto ReturnType = TypePat->getType();
  auto TypeLHS = TypePat->getPattern();
```

```
auto Apply = TypeLHS->asApplyPattern();
if (!Apply)
  errorExit(TypeLHS->getLocation(), "function missing parameter");
auto FunName = Apply->front()->asLongIdentifierPattern();
if (!FunName || FunName->getIDs()->size() != 1)
  errorExit(N->getLocation(), "invalid function identifier");
auto FunID = FunName->getIDs()->toString();

auto FunType = std::make_shared<FunctionTypedef>();
std::vector<Typemap::MapType> ParamNameTypes;
for (size_t I = 1; I < Apply->size(); ++I) {
  this->AddVariableTypeToTypenames = true;
  this->doVisit((*Apply)[I]);
  this->PrevType = this->getPrevPatternType();
  if (!this->PrevType)
    errorExit((*Apply)[I]->getLocation(), "missing type specification");
  if (this->PrevType->getKind() == Typedef::SimpleBuildin)
    (*Apply)[I]->setHasSimpleType();
  FunType->append(this->PrevType);
  ParamNameTypes.push_back(this->PrevPatternmatch->getNameTypes());
}

auto PTypes = mergeParamNameTypes(N->getLocation(), FunID, ParamNameTypes);
for (auto &S : this->PatternIdenInsertionOrder) {
  assert(PTypes.count(S));
  auto Def = PTypes[S];
  if (Def->getKind() == Typedef::SimpleBuildin)
    this->peekFuncVals()->addVal(false);
  else
    this->peekFuncVals()->addVal(true);
}
this->PatternIdenInsertionOrder.clear();

this->AddVariableTypeToTypenames = true;
this->ExprHasValnameApply = false;
this->doVisit(ReturnType);
auto RetTT = this->PrevType;
FunType->append(RetTT);

this->ValnameTypes.insert(FunID, FunType, 1);
this->ConstnameTypes.erase(FunID, 1);

this->enterScope();
this->ValnameTypes.mergeWith(PTypes, 0);
this->ConstnameTypes.erase(PTypes, 0);
this->doVisit(N->getDef());
this->leaveScope();

if (this->cannotBeAssignWith(this->PrevType, ReturnType->getLocation(), RetTT,
                             this->ExprHasValnameApply, false, true))
  errorExit(N->getLocation(),
            "conflicting return type and function expression");

this->PrevType.reset(FunType->clone());
this->PrevFunName = std::move(FunID);
this->ExprHasValnameApply = PrevHasApply;
this->AddVariableTypeToTypenames = PrevFunParam;
```

```
}

void TypecheckVisitor::visit(FunDeclaration *N) {
  std::string FunName;
  std::shared_ptr<Typedef> FunType;

  this->pushFuncVals();
  N->setFunctionVals(this->peekFuncVals());
  this->enterScope();

  for (auto &&F : *N) {
    this->doVisit(F);
    if (!FunType) {
      FunName = this->PrevFunName;
      FunType = this->PrevType;
    } else {
      if (this->PrevFunName != FunName)
        errorExit(N->getLocation(), "conflicting function case identifiers");
      if (*this->PrevType != *FunType)
        errorExit(N->getLocation(), "conflicting function case types");
    }
  }
  FunType->unmarkInScope();
  this->leaveScope();
  this->popFuncVals();
  this->peekFuncVals()->addVal(true);
}

void TypecheckVisitor::visit(DatatypeBareInstanceDeclaration *N) {
  auto Def = this->CurrDatatypeTypedef;
  this->ConstnameTypes.insert(
      *N->getID()->getID(),
      std::make_shared<DatatypeInstanceTypedef>(*Def, nullptr),
      this->InsertionScope);
  this->ValnameTypes.erase(*N->getID()->getID(),
      this->InsertionScope);
}

void TypecheckVisitor::visit(DatatypeTypedInstanceDeclaration *N) {
  auto Def = this->CurrDatatypeTypedef;

  this->doVisit(N->getType());
  auto ParamType = this->PrevType;
  if (!ParamType->isType())
    errorExit(N->getType()->getLocation(), "not a type");
  if (!ParamType->isTypeComplete())
    errorExit(N->getType()->getLocation(), "incomplete type");

  this->ConstnameTypes.insert(
      *N->getID()->getID(),
      std::make_shared<DatatypeInstanceTypedef>(*Def, ParamType),
      this->InsertionScope);
  this->ValnameTypes.erase(*N->getID()->getID(),
      this->InsertionScope);
}

static bool vectorContains(const std::vector<std::string> V,
                           const std::string S) {
  for (auto &&E : V)
    if (E == S)
```

```
      return true;
  return false;
}

void TypecheckVisitor::visit(BareDatatypeDeclaration *N) {
  this->enterScope();

  std::vector<std::string> Ps;
  auto Typedef =
      std::make_shared<DatatypeTypedef>(*N->getID()->getID(), Ps);

  ++this->InsertionScope;

  this->CurrDatatypeTypedef = Typedef;
  this->TypenameTypes.insert(*N->getID()->getID(), Typedef,
                             this->InsertionScope);

  this->doVisit(N->getDef());

  --this->InsertionScope;

  this->leaveScope();
}

void TypecheckVisitor::visit(TypedDatatypeDeclaration *N) {
  std::vector<std::string> Paramnames;
  this->enterScope();
  for (auto &&P : *N->getTypeParams()) {
    const std::string &Name = *P->getID()->getID();
    if (vectorContains(Paramnames, Name))
      errorExit(P->getLocation(),
                "multiple uses of variable type name in type
                    parameter list");
    Paramnames.push_back(this->TypenamePrefix + Name);
    this->TypenameTypes.insert(Name,
        std::make_shared<VariableTypedef>(
                                          Name,
                                              this->TypenamePrefix
                                              + Name),
                               this->InsertionScope);
  }
  auto Typedef =
      std::make_shared<DatatypeTypedef>(*N->getID()->getID(),
                                        std::move(Paramnames));

  ++this->InsertionScope;

  this->CurrDatatypeTypedef = Typedef;
  this->TypenameTypes.insert(*N->getID()->getID(), Typedef,
                             this->InsertionScope);

  this->doVisit(N->getDef());

  --this->InsertionScope;
  this->leaveScope();
}

void TypecheckVisitor::visit(LongIdentifierType *N) {
  auto T = TypenameTypes.get(N->getIDs()->toString());
  if (!T)
    errorExit(N->getLocation(), "unknown type " +
        N->getIDs()->toString());
```

```
  this->PrevType = T;
}

void TypecheckVisitor::visit(VariableType *N) {
  VariableTypedef *T;
  std::shared_ptr<Typedef> ScopeVar1 =
      this->TypenameTypes.get(*N->getID()->getID());
  if (ScopeVar1) {
    assert(ScopeVar1->getKind() == Typedef::Variable);
    auto ScopeVar = static_cast<VariableTypedef
        *>(ScopeVar1.get());
    T = new VariableTypedef(ScopeVar->getUserTypename(),
                            ScopeVar->getInternTypename());
    T->setIsInScope(true);
    this->PrevType.reset(T);
  } else {
    T = new VariableTypedef(*N->getID()->getID(),
                            this->TypenamePrefix +
                                *N->getID()->getID());
    auto TT = std::shared_ptr<VariableTypedef>(T);
    if (this->AddVariableTypeToTypenames) {
      TT->setIsInScope(true);
      this->TypenameTypes.insert(*N->getID()->getID(), TT, 0);
    }
    this->PrevType = TT;
  }
}

void TypecheckVisitor::visit(SeqType *N) {
  for (auto C : *N)
    this->doVisit(C);
}

void TypecheckVisitor::visit(SeqVariableType *N) {
  fatalExit("Typecheckvisitor should not vist SeqVariableType");
}

void TypecheckVisitor::visit(TupleType *N) {
  auto Tup = std::make_shared<TupleTypedef>();
  for (auto &&C : *N->getTypes()) {
    this->doVisit(C);
    Tup->append(this->PrevType);
  }
  this->PrevType = Tup;
}

void TypecheckVisitor::visit(ApplyType *N) {
  std::shared_ptr<Typedef> Typ;
  std::shared_ptr<Typedef> First;

  for (auto &&C : *N) {
    this->doVisit(C);
    auto Next = this->PrevType;
    if (Typ)
      Next->setArgument(C->getLocation(), Typ);
    else
      First = Next;
    Typ = Next;
  }
  if (First->isTypeMissingArgument())
    errorExit(N->getLocation(), "missing type argument 1");

```

```
  this->PrevType = Typ;
}

void TypecheckVisitor::visit(ProductType *N) {
  auto T = std::make_shared<ProductTypedef>();
  for (auto C : *N) {
    this->doVisit(C);
    T->append(this->PrevType);
  }
  this->PrevType = T;
}

void TypecheckVisitor::visit(FunctionType *N) {
  auto T = std::make_shared<FunctionTypedef>();
  for (auto C : *N) {
    this->doVisit(C);
    T->append(this->PrevType);
  }
  this->PrevType = T;
}
```

Listing 79: lib/Typecheck/Patternmatch.h

```
#ifndef LLVM_TOOLS_SSML_LIB_TYPECHECK_PATTERNMATCH_H
#define LLVM_TOOLS_SSML_LIB_TYPECHECK_PATTERNMATCH_H

#include "Typemap.h"

#include "ssml/Common/SourceLocation.h"

#include <string>
#include <memory>

namespace ssml {
namespace typecheck {
class Typedef;
class DatatypeInstanceTypedef;
class DatatypeTypedef;
} // End namespace typecheck.
} // End namespace ssml.

namespace ssml {
namespace typecheck {
class Patternmatch {
protected:
  std::shared_ptr<Typedef> PatternType = nullptr;
  SourceLocation Location;
  Typemap::MapType NameTypes;
  bool IsTypeMatched = false;
  bool IsNameTypesReturned = false;

protected:
  Patternmatch(SourceLocation);
  void appendNameTypes(Typemap::MapType);

public:
  virtual ~Patternmatch() = 0;
  virtual void matchType(std::shared_ptr<Typedef>);
  virtual Typemap::MapType getNameTypes();
  std::shared_ptr<Typedef> getPatternType() { return
      this->PatternType; }
```

```
  virtual std::shared_ptr<Typedef> getPreeMatchPatternType();
};
} // End namespace typecheck.
} // End namespace ssml.

namespace ssml {
namespace typecheck {
class WildcardPatternmatch : public Patternmatch {
public:
  WildcardPatternmatch(SourceLocation);
  void matchType(std::shared_ptr<Typedef>) override;
};
} // End namespace typecheck.
} // End namespace ssml.

namespace ssml {
namespace typecheck {
class LongIdentifierPatternmatch : public Patternmatch {
private:
  const Typemap &ConstnameTypes;
  std::string ID;

public:
  LongIdentifierPatternmatch(SourceLocation, const Typemap
      &ConstnameTypes,
                             std::string &&ID);
  void matchType(std::shared_ptr<Typedef>) override;
  std::shared_ptr<Typedef> getPreeMatchPatternType() override;
};
} // End namespace typecheck.
} // End namespace ssml.

namespace ssml {
namespace typecheck {
class TuplePatternmatch : public Patternmatch {
protected:
  std::vector<std::shared_ptr<Patternmatch>> Matches;

public:
  TuplePatternmatch(SourceLocation);
  virtual void append(std::shared_ptr<Patternmatch>);
  void matchType(std::shared_ptr<Typedef>) override;
  Typemap::MapType getNameTypes() override;
  std::shared_ptr<Typedef> getPreeMatchPatternType() override;
};
} // End namespace typecheck.
} // End namespace ssml.

namespace ssml {
namespace typecheck {
class ListPatternmatch : public TuplePatternmatch {
public:
  ListPatternmatch(SourceLocation);
  void append(std::shared_ptr<Patternmatch>) override;
  void matchType(std::shared_ptr<Typedef>) override;
  std::shared_ptr<Typedef> getPreeMatchPatternType() override;
};
} // End namespace typecheck.
} // End namespace ssml.

namespace ssml {
namespace typecheck {
```

```
class LiteralPatternmatch : public Patternmatch {
private:
  std::shared_ptr<Typedef> LiteralType;

public:
  LiteralPatternmatch(SourceLocation, std::shared_ptr<Typedef>
      LiteralType);
  void matchType(std::shared_ptr<Typedef>) override;
  std::shared_ptr<Typedef> getPreeMatchPatternType() override;
};
} // End namespace typecheck.
} // End namespace ssml.

namespace ssml {
namespace typecheck {
class ApplyPatternmatch : public Patternmatch {
private:
  std::shared_ptr<Patternmatch> Left;
  std::shared_ptr<Patternmatch> Right;
  SourceLocation LeftLocation;
  SourceLocation RightLocation;

private:
  std::shared_ptr<Typedef> match(
      const std::vector<std::pair<std::string,
          std::shared_ptr<Typedef>>> &V,
      std::shared_ptr<Typedef> Match);
  std::shared_ptr<Typedef> simpleBuildinMatch(
      const std::vector<std::pair<std::string,
          std::shared_ptr<Typedef>>> &V,
      std::shared_ptr<Typedef> Match);
  std::shared_ptr<Typedef> productMatch(
      const std::vector<std::pair<std::string,
          std::shared_ptr<Typedef>>> &V,
      std::shared_ptr<Typedef> Match);
  std::shared_ptr<Typedef> functionMatch(
      const std::vector<std::pair<std::string,
          std::shared_ptr<Typedef>>> &V,
      std::shared_ptr<Typedef> Match);
  std::shared_ptr<Typedef> datatypeMatch(
      const std::vector<std::pair<std::string,
          std::shared_ptr<Typedef>>> &V,
      std::shared_ptr<Typedef> Match);
  std::shared_ptr<Typedef> variableMatch(
      const std::vector<std::pair<std::string,
          std::shared_ptr<Typedef>>> &V,
      std::shared_ptr<Typedef> Match);

public:
  ApplyPatternmatch(SourceLocation LeftLocation, SourceLocation
      RightLocation,
                    std::shared_ptr<Patternmatch> LHS,
                    std::shared_ptr<Patternmatch> RHS);
  void matchType(std::shared_ptr<Typedef>) override;
  std::shared_ptr<Typedef> getPreeMatchPatternType() override;
  Typemap::MapType getNameTypes() override;
};
} // End namespace typecheck.
} // End namespace ssml.

#endif // LLVM_TOOLS_SSML_LIB_TYPECHECK_PATTERNMATCH_H
```

Listing 80: lib/Typecheck/TypecheckVisitor.h

```
#ifndef LLVM_TOOLS_SSML_LIB_TYPECHECK_TYPECHECKVISITOR_H
#define LLVM_TOOLS_SSML_LIB_TYPECHECK_TYPECHECKVISITOR_H

#include "Typemap.h"
#include "Typedef.h"

#include "ssml/AST/FunctionVals.h"
#include "ssml/AST/Visitor.h"

namespace ssml {
namespace ast {
class Node;
class Pattern;
} // End namespace ast.
} // End namespace ssml.

namespace ssml {
namespace typecheck {
class Patternmatch;
class Typedef;
} // End namespace typecheck.
} // End namespace ssml.

namespace ssml {
namespace typecheck {
class TypecheckVisitor : public ssml::ast::Visitor {
private:
  Typemap TypenameTypes;
  Typemap ValnameTypes;
  Typemap ConstnameTypes;
  size_t InsertionScope = 0;
  /**
   * TypenamePrefix is used to make sure user defined datatypes are distinct
   * when declared in different scopes, even though the name is the same. E.g.:
   * datatype A = ...
   * val x:A = ...
   * let
   *   datatype A = ...
   *   val y:A = x (* Error type of x != type of y *)
   * in
   *   ...
   * end
   */
  std::string TypenamePrefix;

  std::shared_ptr<Patternmatch> PrevPatternmatch;
  std::shared_ptr<DatatypeTypedef> CurrDatatypeTypedef;
  std::shared_ptr<Typedef> PrevType;

  std::shared_ptr<Typedef> IntTypedef =
      std::make_shared<SimpleBuildinTypedef>("int");
  std::shared_ptr<Typedef> BoolTypedef =
      std::make_shared<SimpleBuildinTypedef>("bool");
  std::shared_ptr<Typedef> UnitTypedef =
      std::make_shared<ProductTypedef>();

  std::string PrevFunName;

  bool ApplyExprWasVal = false;
```

```
  bool ExprHasValnameApply = false;

  bool AddVariableTypeToTypenames = false;

  std::vector<std::string> PatternIdenInsertionOrder;

  std::vector<std::shared_ptr<ssml::ast::FunctionVals>>
      FuncValsStack;

private:
  void enterScope();
  void leaveScope();
  void addDefaults();
  void doVisit(std::shared_ptr<ssml::ast::Node>);
  void doVisit(ssml::ast::Node *);
  bool cannotBeAssignWith(std::shared_ptr<Typedef> DestType,
                          SourceLocation SourceL,
                          std::shared_ptr<Typedef> SourceType,
                          bool ExprHasValApplication, bool
                              ScopedVarIsVar,
                          bool IsRet = false);
  std::shared_ptr<Typedef> getPrevPatternType();
  void pushFuncVals() {
    this->FuncValsStack.push_back(std::shared_ptr<ssml::ast::FunctionVals>(
        new ssml::ast::FunctionVals()));
  }
  void popFuncVals() { this->FuncValsStack.pop_back(); }
  std::shared_ptr<ssml::ast::FunctionVals> peekFuncVals() {
    return this->FuncValsStack.back();
  }

public:
  void visit(ssml::ast::IntLiteral *) override;

  void visit(ssml::ast::ShortIdentifier *) override;
  void visit(ssml::ast::LongIdentifier *) override;
  void visit(ssml::ast::SeqShortIdentifier *) override;
  void visit(ssml::ast::SeqLongIdentifier *) override;

  void visit(ssml::ast::Match *) override;
  void visit(ssml::ast::SeqMatch *) override;

  void visit(ssml::ast::SeqExpression *) override;
  void visit(ssml::ast::LiteralExpression *) override;
  void visit(ssml::ast::LongIdentifierExpression *) override;
  void visit(ssml::ast::TupleExpression *) override;
  void visit(ssml::ast::ListExpression *) override;
  void visit(ssml::ast::ApplyExpression *) override;
  void visit(ssml::ast::OrElseExpression *) override;
  void visit(ssml::ast::AndAlsoExpression *) override;
  void visit(ssml::ast::LetExpression *) override;
  void visit(ssml::ast::IfExpression *) override;
  void visit(ssml::ast::WhileExpression *) override;
  void visit(ssml::ast::LambdaExpression *) override;

  void visit(ssml::ast::SeqPattern *) override;
  void visit(ssml::ast::LiteralPattern *) override;
  void visit(ssml::ast::WildcardPattern *) override;
  void visit(ssml::ast::LongIdentifierPattern *) override;
  void visit(ssml::ast::TypePattern *) override;
  void visit(ssml::ast::ApplyPattern *) override;
  void visit(ssml::ast::ListPattern *) override;
```

```
  void visit(ssml::ast::TuplePattern *) override;

  void visit(ssml::ast::SeqDeclaration *) override;
  void visit(ssml::ast::Root *) override;
  void visit(ssml::ast::ValDeclaration *) override;
  void visit(ssml::ast::NonfixDeclaration *) override;
  void visit(ssml::ast::InfixDeclaration *) override;
  void visit(ssml::ast::InfixRDeclaration *) override;
  void visit(ssml::ast::FunPatternDeclaration *) override;
  void visit(ssml::ast::FunDeclaration *) override;
  void visit(ssml::ast::DatatypeBareInstanceDeclaration *)
      override;
  void visit(ssml::ast::DatatypeTypedInstanceDeclaration *)
      override;
  void visit(ssml::ast::BareDatatypeDeclaration *) override;
  void visit(ssml::ast::TypedDatatypeDeclaration *) override;

  void visit(ssml::ast::LongIdentifierType *) override;
  void visit(ssml::ast::VariableType *) override;
  void visit(ssml::ast::SeqType *) override;
  void visit(ssml::ast::SeqVariableType *) override;
  void visit(ssml::ast::TupleType *) override;
  void visit(ssml::ast::ApplyType *) override;
  void visit(ssml::ast::ProductType *) override;
  void visit(ssml::ast::FunctionType *) override;
};
} // End namespace typecheck.
} // End namespace ssml.

#endif // LLVM_TOOLS_SSML_LIB_TYPECHECK_TYPECHECKVISITOR_H
```

Listing 81: lib/Typecheck/Typedef.cpp

```
#include "Typedef.h"

#include "ssml/Common/ErrorMessages.h"
#include "ssml/Common/FatalExit.h"

#include "llvm/ADT/StringRef.h"

#include "llvm/Support/raw_ostream.h"

#include <map>

using namespace ssml;
using namespace ssml::typecheck;

static std::shared_ptr<Typedef>
    &vectorGetRef(Typedef::VartypeVector &V,
                                                            const std::string
                                                                &S) {
  for (auto &&P : V) {
    if (P.first == S) {
      assert(!!P.second);
      return P.second;
    }
  }
  fatalExit("cannot return ref!");
  return V[0].second;
}

```

```
static std::shared_ptr<Typedef> vectorGet(const Typedef::VartypeVector &V,
                                          const std::string &S) {
  for (auto &&P : V) {
    if (P.first == S) {
      assert(!!P.second);
      return P.second;
    }
  }
  return nullptr;
}

static std::shared_ptr<Typedef> vectorGet(const Typedef::VartypeVector *V,
                                          const std::string &S) {
  return vectorGet(*V, S);
}

Typedef::~Typedef() = default;

bool Typedef::canBeAssignWith(const Typedef &Y, const VartypeVector *V,
                              bool ScopedVarIsVar) const {
  return !this->cannotBeAssignWith(Y, V, ScopedVarIsVar);
}

bool Typedef::cannotBeAssignWith(const Typedef &Y, const VartypeVector *V,
                                 bool ScopedVarIsVar) const {
  return !this->canBeAssignWith(Y, V, ScopedVarIsVar);
}

bool Typedef::isTypeMissingArgument() const { return false; }

bool Typedef::operator==(const Typedef &Y) const { return !(*this != Y); }

bool Typedef::operator!=(const Typedef &Y) const { return !(*this == Y); }

bool Typedef::isType() const { return true; }

std::shared_ptr<Typedef>
Typedef::funApplication(SourceLocation L, std::shared_ptr<Typedef> T) const {
  errorExit(L, "cannot apply argument here 1");
  return nullptr;
}

SimpleBuildinTypedef::SimpleBuildinTypedef(std::string N)
    : Typename(std::move(N)) {}

SimpleBuildinTypedef *SimpleBuildinTypedef::clone() const {
  return new SimpleBuildinTypedef(*this);
}

void SimpleBuildinTypedef::unmarkInScope() {}

void SimpleBuildinTypedef::setArgument(SourceLocation L,
                                       std::shared_ptr<Typedef> A) {
  errorExit(L, "type argument(s) applied to simple build in type
```

```
      '" +
                    this->Typename + "'");
}

std::shared_ptr<Typedef>
SimpleBuildinTypedef::withoutVariableTypedefs(const VartypeVector
    &) {
  return std::shared_ptr<Typedef>(this->clone());
}

Typedef::VartypeVector
    SimpleBuildinTypedef::getVariableTypes(SourceLocation,
                                           std::shared_ptr<Typedef>)
                                               const {
  return {};
}

static const Typedef &getVarTypedef(const Typedef &X,
                                    const Typedef::VartypeVector
                                        *V) {
  if (!V || X.getKind() != Typedef::Variable)
    return X;
  auto &Var = static_cast<const VariableTypedef &>(X);
  auto Ret = vectorGet(V, Var.getInternTypename());
  if (!Ret)
    return X;
  return *Ret;
}

bool SimpleBuildinTypedef::canBeAssignWith(const Typedef &X,
                                           const
                                               Typedef::VartypeVector
                                               *V,
                                           bool ScopedVarIsVar)
                                               const {
  auto &Y = getVarTypedef(X, V);
  if (this->getKind() != Y.getKind())
    return false;
  auto YY = static_cast<const SimpleBuildinTypedef &>(Y);
  return this->Typename == YY.Typename;
}

bool SimpleBuildinTypedef::hasVariable() const { return false; }

bool SimpleBuildinTypedef::operator==(const Typedef &Y) const {
  return this->canBeAssignWith(Y, nullptr, false);
}

bool SimpleBuildinTypedef::isTypeComplete() const { return true; }

Typedef::Kind SimpleBuildinTypedef::getKind() const {
  return Typedef::SimpleBuildin;
}

#if 0
ProductTypedef::ProductTypedef(const ProductTypedef &Y) {
  for (auto &&T : Y.Typedefs)
    this->Typedefs->push_back(std::shared_ptr<Typedef>(T->clone()));
}
#endif

ProductTypedef *ProductTypedef::clone() const {
```

```
  return new ProductTypedef(*this);
}

void ProductTypedef::setArgument(SourceLocation L,
    std::shared_ptr<Typedef>) {
  errorExit(L, "tuple does not take type argument(s)");
}

void ProductTypedef::unmarkInScope() {
  for (auto T : this->Typedefs)
    T->unmarkInScope();
}

Typedef::VartypeVector
ProductTypedef::getVariableTypes(SourceLocation L,
                                 std::shared_ptr<Typedef> Def)
                                     const {
  std::vector<std::pair<std::string, std::shared_ptr<Typedef>>>
      Ret;

#if 0
  if (Def->getKind() != this->getKind())
    errorExit(L, "unexpected argument(s)");
#else
  if (Def->getKind() != this->getKind())
    return {};
#endif

  auto PTypes = static_cast<ProductTypedef *>(Def.get())->Typedefs;
  auto Size = this->Typedefs.size();
  if (Size != PTypes.size())
    errorExit(L, "unexpected number of type arguments 1");

  for (size_t I = 0; I < Size; ++I) {
    auto &T = this->Typedefs[I];

    assert(!!T);
    assert(!!PTypes[I]);
    auto M = T->getVariableTypes(L, PTypes[I]);
    for (auto &&Def : M) {
      auto Elem = vectorGet(Ret, Def.first);
      Typedef::VartypeVector VTypes;
      if (Elem)
        VTypes = Elem->getVariableTypes(L, Def.second);
      if (Elem && Def.second->cannotBeAssignWith(*Elem, &VTypes,
          false))
        errorExit(L, "contracicting types for type variable [1]");
      if (!Elem)
        Ret.push_back(Def);
      else
        vectorGetRef(Ret, Def.first) =
            Elem->withoutVariableTypedefs(VTypes);
    }
  }

  return Ret;
}

void ProductTypedef::append(std::shared_ptr<Typedef> T) {
  this->Typedefs.push_back(T);
}

```

```
std::shared_ptr<Typedef>
ProductTypedef::withoutVariableTypedefs(const VartypeVector &V) {
  auto Ret = std::make_shared<ProductTypedef>();
  for (auto &&T : this->Typedefs)
    Ret->append(T->withoutVariableTypedefs(V));
  return Ret;
}

static bool vecCanBeAssignWith(const std::vector<std::shared_ptr<Typedef>> &LHS,
                               const std::vector<std::shared_ptr<Typedef>> &RHS,
                               const Typedef::VartypeVector *V,
                               bool ScopedVarIsVar) {
  auto Size = LHS.size();
  if (Size != RHS.size())
    return false;
  for (size_t I = 0; I < Size; ++I)
    if (LHS[I]->cannotBeAssignWith(*RHS[I], V, ScopedVarIsVar))
      return false;
  return true;
}

bool ProductTypedef::canBeAssignWith(const Typedef &X,
                                     const Typedef::VartypeVector *V,
                                     bool ScopedVarIsVar) const {
  auto &Y = getVarTypedef(X, V);
  if (this->getKind() != Y.getKind())
    return false;
  auto YY = static_cast<const ProductTypedef &>(Y);
  return vecCanBeAssignWith(this->Typedefs, YY.Typedefs, V, ScopedVarIsVar);
}

bool ProductTypedef::hasVariable() const {
  for (auto &&T : this->Typedefs) {
    if (T->hasVariable())
      return true;
  }
  return false;
}

static bool vecEquals(const std::vector<std::shared_ptr<Typedef>> &LHS,
                      const std::vector<std::shared_ptr<Typedef>> &RHS) {
  auto Size = LHS.size();
  if (Size != RHS.size())
    return false;
  for (size_t I = 0; I < Size; ++I)
    if (*LHS[I] != *RHS[I])
      return false;
  return true;
}

bool ProductTypedef::operator==(const Typedef &Y) const {
  if (this->getKind() != Y.getKind())
    return false;
  auto YY = static_cast<const ProductTypedef &>(Y);
  return vecEquals(this->Typedefs, YY.Typedefs);
```

```
}

bool ProductTypedef::isTypeComplete() const {
  for (auto &&T : this->Typedefs)
    if (!T->isTypeComplete())
      return false;
  return true;
}

Typedef::Kind ProductTypedef::getKind() const { return Typedef::Product; }

#if 0
FunctionTypedef::FunctionTypedef(const FunctionTypedef &Y) {
  for (auto &&T : Y.Typedefs)
    this->Typedefs->push_back(std::shared_ptr<Typedef>(T->clone()));
}
#endif

FunctionTypedef *FunctionTypedef::clone() const {
  return new FunctionTypedef(*this);
}

void FunctionTypedef::append(std::shared_ptr<Typedef> T) {
  if (this->Typedefs.size() < 2) {
    this->Typedefs.push_back(T);
  } else if (this->HasCreatedAppendFunc) {
    assert(this->Typedefs.back()->getKind() == Typedef::Function);
    auto Back = static_cast<ProductTypedef *>(this->Typedefs.back().get());
    Back->append(T);
  } else {
    this->HasCreatedAppendFunc = true;
    auto AppendFunc = std::make_shared<FunctionTypedef>();
    auto First = this->Typedefs.back();
    this->Typedefs.back() = AppendFunc;
    AppendFunc->append(First);
    AppendFunc->append(T);
  }
}

Typedef::Kind FunctionTypedef::getKind() const { return Typedef::Function; }

bool FunctionTypedef::canBeAssignWith(const Typedef &X,
                                      const Typedef::VartypeVector *V,
                                      bool ScopedVarIsVar) const {
  auto &Y = getVarTypedef(X, V);
  if (this->getKind() != Y.getKind())
    return false;
  auto YY = static_cast<const ProductTypedef &>(Y);

  std::map<std::string, std::shared_ptr<const Typedef>> VarTypenameMap;

  auto &LHS = this->getTypedefs();
  auto &RHS = YY.getTypedefs();
  auto Size = LHS.size();

  if (Size != RHS.size())
    return false;
```

```

  for (size_t I = 0; I < Size; ++I) {
    std::shared_ptr<const Typedef> RightType;

    if (RHS[I]->getKind() == Typedef::Variable) {
      auto R = static_cast<VariableTypedef *>(RHS[I].get());
      if (!VarTypenameMap.count(R->getUserTypename())) {
        VarTypenameMap[R->getUserTypename()] = LHS[I];
        continue;
      } else {
        RightType = VarTypenameMap[R->getUserTypename()];
      }
    } else {
      RightType = RHS[I];
    }

    if (LHS[I]->cannotBeAssignWith(*RightType, V, ScopedVarIsVar))
      return false;
  }

  return true;
}

std::shared_ptr<Typedef>
FunctionTypedef::funApplication(SourceLocation L,
                                std::shared_ptr<Typedef> T) const {
  auto &Orig = this->getTypedefs();
  if (Orig.size() != 2)
    fatalExit("FunctionTypedef with wrong number of typedefs");

  auto VarTypes = Orig.front()->getVariableTypes(L, T);
  for (auto &&V : VarTypes) {
    auto T = vectorGet(this->ApplicationVarTypes, V.first);
    if (T && *T != *V.second)
      errorExit(L, "argument has unexpected type [1]");
    assert(!!V.second);
    if (!T)
      this->ApplicationVarTypes.push_back(V);
  }
  if (Orig.front()->cannotBeAssignWith(*T,
      &this->ApplicationVarTypes, false) &&
      T->cannotBeAssignWith(*Orig.front(),
          &this->ApplicationVarTypes, false))
    errorExit(L, "argument has unexpected type [2]");

  std::shared_ptr<Typedef> Ret =
      std::shared_ptr<Typedef>(Orig.back()->clone());
  return Ret->withoutVariableTypedefs(this->ApplicationVarTypes);
}

std::shared_ptr<Typedef>
FunctionTypedef::withoutVariableTypedefs(const VartypeVector &V) {
  auto Ret = std::make_shared<FunctionTypedef>();
  for (auto &&T : this->Typedefs)
    Ret->append(T->withoutVariableTypedefs(V));
  return Ret;
}

#if 0
TupleTypedef::TupleTypedef(const FunctionTypedef &Y) {
  for (auto &&T : Y.Typedefs)
    this->Typedefs->push_back(std::shared_ptr<Typedef>(T->clone()));
```

```
}
#endif

TupleTypedef *TupleTypedef::clone() const { return new
    TupleTypedef(*this); }

Typedef::Kind TupleTypedef::getKind() const { return
    Typedef::Tuple; }

void TupleTypedef::append(std::shared_ptr<Typedef> T) {
  this->Typedefs.push_back(T);
}

void TupleTypedef::setArgument(SourceLocation L,
    std::shared_ptr<Typedef>) {
  errorExit(L, "cannot apply type argument here");
}

void TupleTypedef::unmarkInScope() {
  for (auto T : this->Typedefs)
    T->unmarkInScope();
}

Typedef::VartypeVector
TupleTypedef::getVariableTypes(SourceLocation L,
                               std::shared_ptr<Typedef> Def) const
                                   {
  fatalExit("get variable types from tuple type");
  return {};
}

bool TupleTypedef::canBeAssignWith(const Typedef &X,
                                   const Typedef::VartypeVector *V,
                                   bool ScopedVarIsVar) const {
  auto &Y = getVarTypedef(X, V);
  if (this->getKind() != Y.getKind())
    return false;
  auto YY = static_cast<const TupleTypedef &>(Y);
  return vecCanBeAssignWith(this->Typedefs, YY.Typedefs, V,
      ScopedVarIsVar);
}

bool TupleTypedef::hasVariable() const {
  for (auto &&T : this->Typedefs) {
    if (T->hasVariable())
      return true;
  }
  return false;
}

bool TupleTypedef::operator==(const Typedef &Y) const {
  if (this->getKind() != Y.getKind())
    return false;
  auto YY = static_cast<const TupleTypedef &>(Y);
  return vecEquals(this->Typedefs, YY.Typedefs);
}

bool TupleTypedef::isTypeComplete() const {
  for (auto &&T : this->Typedefs)
    if (!T->isTypeComplete())
      return false;
  return true;
```

```
}

std::shared_ptr<Typedef>
TupleTypedef::withoutVariableTypedefs(const VartypeVector &V) {
  auto Ret = std::make_shared<TupleTypedef>();
  for (auto &&T : this->Typedefs)
    Ret->append(T->withoutVariableTypedefs(V));
  return Ret;
}

bool TupleTypedef::isType() const { return false; }

#if 0
DatatypeTypedef::DatatypeTypedef(const DatatypeTypedef &Y)
    : Typename(Y.Typename), IsArgumentSet(Y.IsArgumentSet) {
  for (auto &&P : Y.Params)
    Params.push_back(P.first,
        std::shared_ptr<Typedef>(P.second->clone())});
}
#endif

DatatypeTypedef::DatatypeTypedef(std::string Name,
                                 std::vector<std::string>
                                     Paramnames)
    : Typename(std::move(Name)) {
  for (auto &&N : Paramnames)
    Params.push_back({std::move(N), nullptr});
}

DatatypeTypedef::DatatypeTypedef(
    std::string Name,
    const std::vector<std::pair<std::string,
        std::shared_ptr<Typedef>>>
        &Paramnames)
    : Typename(std::move(Name)) {
  for (auto &&N : Paramnames)
    Params.push_back({N.first, nullptr});
}

DatatypeTypedef *DatatypeTypedef::clone() const {
  return new DatatypeTypedef(*this);
}

Typedef::Kind DatatypeTypedef::getKind() const { return
    Typedef::Datatype; }

void DatatypeTypedef::setArgument(SourceLocation L,
                                  std::shared_ptr<Typedef> T) {
  this->IsArgumentSet = true;

  auto Size = this->Params.size();
  if (Size == 0)
    errorExit(L, "type does not take type arguments");

  if (Size == 1) {
    if (T->getKind() == Typedef::Tuple) {
      auto TT = static_cast<TupleTypedef *>(T.get());
      if (TT->getTypedefs().size() != 1)
        errorExit(L, "unexpected number of type arguments 2");
      this->Params.back().second = TT->getTypedefs().front();
    } else {
      this->Params.back().second = T;
```

```
    }
    return;
  }

  if (T->getKind() != Typedef::Tuple)
    errorExit(L, "expected more type argument(s)");
  auto Types = static_cast<TupleTypedef *>(T.get())->getTypedefs();

  if (Types.size() != Size)
    errorExit(L, "unexpected number of type arguments 3");

  for (size_t I = 0; I < Size; ++I)
    this->Params[I].second = Types[I];
}

void DatatypeTypedef::unmarkInScope() {
  for (auto &P : this->Params) {
    P.second->unmarkInScope();
  }
}

std::shared_ptr<Typedef>
DatatypeTypedef::withoutVariableTypedefs(const VartypeVector &V) {
  auto Ret = std::make_shared<DatatypeTypedef>(this->Typename, this->Params);
  for (auto &&P : this->Params)
    if (P.second)
      Ret->setVariableType(P.first,
          P.second->withoutVariableTypedefs(V));
  return Ret;
}

void DatatypeTypedef::setVariableType(const std::string &Name,
                                      std::shared_ptr<Typedef> T) {
  size_t Idx = 0;
  size_t Size = this->Params.size();
  for (Idx = 0; Idx < Size; ++Idx)
    if (Name == this->Params[Idx].first)
      break;
  if (Idx >= Size)
    fatalExit("type variable " + Name + " expected as datatype parameter");
  this->Params[Idx].second = T;
}

Typedef::VartypeVector
DatatypeTypedef::getVariableTypes(SourceLocation L,
                                  std::shared_ptr<Typedef> T) const {
  std::vector<std::pair<std::string, std::shared_ptr<Typedef>>> Ret;

  const DatatypeTypedef *Def = nullptr;
  if (T->getKind() == Typedef::DatatypeInstance)
    Def = &static_cast<const DatatypeInstanceTypedef *>(T.get())
               ->getDatatypedef();
  else if (T->getKind() == Typedef::Datatype)
    Def = static_cast<const DatatypeTypedef *>(T.get());
#if 0
  else
    errorExit(L, "unexpected type");
#else
```

```
  else
    return {};
#endif

  auto Size = this->Params.size();
  if (Size != Def->Params.size())
    errorExit(L, "unexpected number of arguments");

  for (size_t I = 0; I < Size; ++I) {
    auto &T = this->Params[I].second;
#if 0
    if (T->getKind() != Typedef::Variable && Def->Params[I].second &&
        T->cannotBeAssignWith(*Def->Params[I].second, nullptr, false))
      errorExit(L, "invalid argument [2]");
#endif

    if (!T)
      continue;
    if (!Def->Params[I].second)
      continue;

    auto M = T->getVariableTypes(L, Def->Params[I].second);
    for (auto &&Def : M) {
      auto Elem = vectorGet(Ret, Def.first);
      if (Elem && Elem->cannotBeAssignWith(*Def.second, nullptr, false))
        errorExit(L, "contracicting types for type variable [2]");
      if (!Elem && Def.second)
        Ret.push_back(Def);
    }
  }

  return Ret;
}

bool DatatypeTypedef::canBeAssignWith(const Typedef &X,
                                      const Typedef::VartypeVector *V,
                                      bool ScopedVarIsVar) const {
  auto &Y = getVarTypedef(X, V);

  if (Y.getKind() == Typedef::DatatypeInstance) {
    auto &YY = static_cast<const DatatypeInstanceTypedef &>(Y);
    return this->canBeAssignWith(YY.getDatatypedef(), V, ScopedVarIsVar);
  } else if (Y.getKind() != Typedef::Datatype) {
    return false;
  }

  auto &YY = static_cast<const DatatypeTypedef &>(Y);
  auto Size = this->Params.size();
  if (Size != YY.Params.size())
    return false;
  for (size_t I = 0; I < Size; ++I) {
#if 0
    if (!this->Params[I].second)
      fatalExit("datatype typedef parameter was expected to be set!");
#endif
    if (!this->Params[I].second || !YY.Params[I].second)
```

```
      continue;
    if (YY.Params[I].second->getKind() == Typedef::Variable) {
      auto Var = static_cast<VariableTypedef *>(YY.Params[I].second.get());
      if (!Var->isInScope() || ScopedVarIsVar) {
        continue;
      }
    }
    if (this->Params[I].second->cannotBeAssignWith(*YY.Params[I].second, V,
                                                    ScopedVarIsVar))
      return false;
  }
  return this->Typename == YY.Typename;
}

bool DatatypeTypedef::hasVariable() const {
  for (auto &&P : this->Params) {
    if (!P.second || P.second->hasVariable())
      return true;
  }
  return false;
}

bool DatatypeTypedef::operator==(const Typedef &Y) const {
  if (Y.getKind() == Typedef::DatatypeInstance) {
    auto &YY = static_cast<const DatatypeInstanceTypedef &>(Y);
    return *this == YY.getDatatypedef();
  } else if (Y.getKind() != Typedef::Datatype) {
    return false;
  }

  auto &YY = static_cast<const DatatypeTypedef &>(Y);
  auto Size = this->Params.size();
  if (Size != YY.Params.size())
    return false;
  for (size_t I = 0; I < Size; ++I) {
    if (!this->Params[I].second)
      fatalExit("datatype typedef parameter was expected to be set!");
    if (!YY.Params[I].second)
      continue;
    else if (*this->Params[I].second != *YY.Params[I].second)
      return false;
  }
  return this->Typename == YY.Typename;
}

bool DatatypeTypedef::isTypeComplete() const {
  if (!this->Params.size())
    return true;

  if (!this->IsArgumentSet)
    return false;

  for (auto &&P : this->Params)
    if (!P.second || !P.second->isTypeComplete())
      return false;
  return true;
}

```

```
bool DatatypeTypedef::isTypeMissingArgument() const {
  return this->Params.size() && !this->IsArgumentSet;
}

VariableTypedef *VariableTypedef::clone() const {
  return new VariableTypedef(*this);
}

VariableTypedef::VariableTypedef(std::string UserName, std::string InternName)
    : UserTypename(UserName), InternTypename(InternName) {}

Typedef::Kind VariableTypedef::getKind() const { return Typedef::Variable; }

void VariableTypedef::setArgument(SourceLocation L, std::shared_ptr<Typedef>) {
  errorExit(L, "variable type does not take type argument(s)");
}

void VariableTypedef::unmarkInScope() { this->IsInScope = false; }

Typedef::VartypeVector
VariableTypedef::getVariableTypes(SourceLocation L,
                                  std::shared_ptr<Typedef> T) const {
  return {{this->InternTypename, T}};
}

bool VariableTypedef::canBeAssignWith(const Typedef &X,
                                      const Typedef::VartypeVector *V,
                                      bool ScopedVarIsVar) const {
  auto &Y = getVarTypedef(X, V);

  if (this->getKind() != Y.getKind())
    return false;
  auto YY = static_cast<const VariableTypedef &>(Y);
  return this->InternTypename == YY.InternTypename;
}

bool VariableTypedef::hasVariable() const {
  return this->IsInScope ? false : true;
}

std::shared_ptr<Typedef>
VariableTypedef::withoutVariableTypedefs(const VartypeVector &V) {
  auto T = vectorGet(V, this->InternTypename);
  if (!T)
    return std::shared_ptr<Typedef>(this->clone());
  return T;
}

bool VariableTypedef::operator==(const Typedef &Y) const {
  return this->canBeAssignWith(Y, nullptr, false);
}

bool VariableTypedef::isTypeComplete() const { return true; }

#if 0
DatatypeInstanceTypedef::DatatypeInstanceTypedef(
    const DatatypeInstanceTypedef &Y)
```

```
    : Datatypedef(Y.Datatypedef),
      ArgumentTypedef(Y.ArgumentTYpedef->clone()),
      Argument(Y.Argument->clone()) {}
#endif

DatatypeInstanceTypedef::DatatypeInstanceTypedef(
    const DatatypeTypedef &Datatype, std::shared_ptr<Typedef> Arg)
    : Datatypedef(Datatype), ArgumentTypedef(Arg) {}

DatatypeInstanceTypedef *DatatypeInstanceTypedef::clone() const {
  return new DatatypeInstanceTypedef(*this);
}

Typedef::Kind DatatypeInstanceTypedef::getKind() const {
  return Typedef::DatatypeInstance;
}

namespace dummy_namespace {
size_t vectorCount(
    const std::vector<std::pair<std::string,
        std::shared_ptr<Typedef>>> &V,
    const std::string &S) {
  size_t Ret = 0;
  for (auto &&P : V)
    if (P.first == S)
      ++Ret;
  return Ret;
}
} // End dummy_namespace namespace.

namespace dummy_namespace {
bool noDuplicates(
    const std::vector<std::pair<std::string,
        std::shared_ptr<Typedef>>> &V) {
  for (auto &&P : V) {
    if (vectorCount(V, P.first) > 1)
      return false;
  }
  return true;
}
} // End dummy_namespace namespace.

void DatatypeInstanceTypedef::setArgument(SourceLocation L,
                                          std::shared_ptr<Typedef>
                                              T) {
  if (!this->ArgumentTypedef)
    errorExit(L, "does not take arguments");

  this->Argument.reset(T->clone());

  auto VarTypes = this->ArgumentTypedef->getVariableTypes(L, T);

  if (T->cannotBeAssignWith(*this->ArgumentTypedef, &VarTypes,
      false))
    errorExit(L, "argument type mismatch");

  assert(dummy_namespace::noDuplicates(VarTypes));
  for (auto &&P : VarTypes)
    this->Datatypedef.setVariableType(P.first, P.second);
}
void DatatypeInstanceTypedef::unmarkInScope() {
  this->Datatypedef.unmarkInScope();
```

```
}

std::shared_ptr<Typedef>
DatatypeInstanceTypedef::funApplication(SourceLocation L,
                                        std::shared_ptr<Typedef>
                                            T) const {
  if (this->Argument)
    errorExit(L, "cannot apply argument here 2");
  auto Ret = std::shared_ptr<Typedef>(this->clone());
  Ret->setArgument(L, T);
  return Ret;
}

Typedef::VartypeVector
DatatypeInstanceTypedef::getVariableTypes(SourceLocation L,
                                          std::shared_ptr<Typedef>
                                              T) const {
  return this->Datatypedef.getVariableTypes(L, T);
}

bool DatatypeInstanceTypedef::canBeAssignWith(const Typedef &X,
                                              const
                                                  Typedef::VartypeVector
                                                  *V,
                                              bool ScopedVarIsVar)
                                                  const {
  auto &Y = getVarTypedef(X, V);
  if (Y.getKind() == Typedef::Datatype) {
    auto YY = static_cast<const DatatypeTypedef &>(Y);
    return this->Datatypedef.canBeAssignWith(YY, V,
        ScopedVarIsVar);
  } else if (Y.getKind() == Typedef::DatatypeInstance) {
    auto YY = static_cast<const DatatypeInstanceTypedef &>(Y);
    return this->Datatypedef.canBeAssignWith(YY.Datatypedef, V,
        ScopedVarIsVar);
  }
  return false;
}

bool DatatypeInstanceTypedef::hasVariable() const {
  return this->Datatypedef.hasVariable();
}

bool DatatypeInstanceTypedef::operator==(const Typedef &Y) const {
  if (Y.getKind() == Typedef::Datatype) {
    auto YY = static_cast<const DatatypeTypedef &>(Y);
    return YY == *this;
  } else if (Y.getKind() == Typedef::DatatypeInstance) {
    auto YY = static_cast<const DatatypeInstanceTypedef &>(Y);
    return this->Datatypedef == YY.Datatypedef;
  }
  return false;
}

bool DatatypeInstanceTypedef::isTypeComplete() const {
  if (this->ArgumentTypedef && !this->Argument)
    return false;
  if (this->ArgumentTypedef && !this->Argument->isTypeComplete())
    return false;
  auto &Params = this->Datatypedef.getParams();
  for (auto &&P : Params) {
    if (!P.second || !P.second->isTypeComplete())
```

```
      return false;
  }
  return true;
}

bool DatatypeInstanceTypedef::isTypeMissingArgument() const {
  if (!this->ArgumentTypedef)
    return false;
  if (this->Argument)
    return false;
  return true;
}

std::shared_ptr<Typedef>
DatatypeInstanceTypedef::withoutVariableTypedefs(const
    VartypeVector &V) {
  std::shared_ptr<DatatypeInstanceTypedef> Ret;
  if (this->ArgumentTypedef)
    Ret = std::shared_ptr<DatatypeInstanceTypedef>(new
        DatatypeInstanceTypedef(
        static_cast<const DatatypeTypedef &>(
            *this->Datatypedef.withoutVariableTypedefs(V)),
        std::shared_ptr<Typedef>(this->ArgumentTypedef->clone())));
  else
    Ret = std::shared_ptr<DatatypeInstanceTypedef>(new
        DatatypeInstanceTypedef(
        static_cast<const DatatypeTypedef &>(
            *this->Datatypedef.withoutVariableTypedefs(V)),
        std::shared_ptr<Typedef>(nullptr)));

  if (this->Argument)
    Ret->Argument.reset(this->Argument->clone());

  return Ret;
}

std::shared_ptr<DatatypeTypedef> ssml::typecheck::getListTypedef()
    {
  return std::make_shared<DatatypeTypedef>("list",
                                           std::vector<std::string>{"'a"});
}

std::shared_ptr<DatatypeTypedef> ssml::typecheck::getRefTypedef() {
  return std::make_shared<DatatypeTypedef>("ref",
                                           std::vector<std::string>{"'a"});
}

std::shared_ptr<DatatypeTypedef>
    ssml::typecheck::getArrayTypedef() {
  return std::make_shared<DatatypeTypedef>("array",
                                           std::vector<std::string>{"'a"});
}
```

Listing 82: lib/Typecheck/Patternmatch.cpp

```
#include "Patternmatch.h"
#include "Typedef.h"

#include "ssml/Common/FatalExit.h"
#include "ssml/Common/ErrorMessages.h"

```

```
#include "llvm/Support/raw_ostream.h"

#include <vector>

using namespace ssml::typecheck;

Patternmatch::Patternmatch(SourceLocation L) : Location(L) {}

Patternmatch::~Patternmatch() = default;

void Patternmatch::matchType(std::shared_ptr<Typedef> T) {
  if (this->IsTypeMatched && *T != *this->PatternType)
    errorExit(this->Location, "conflicting pattern type specifications");
  this->IsTypeMatched = true;
  this->PatternType = T;
}

std::shared_ptr<Typedef> Patternmatch::getPreeMatchPatternType() {
  if (this->IsTypeMatched)
    return this->PatternType;
  return nullptr;
}

Typemap::MapType Patternmatch::getNameTypes() {
  auto Pree = this->getPreeMatchPatternType();
  if (!Pree || !Pree->isTypeComplete())
    errorExit(this->Location, "missing type declaration");
  if (this->IsNameTypesReturned)
    fatalExit("Patternmatch Name types already returned");
  this->IsNameTypesReturned = true;
  return std::move(this->NameTypes);
}

void Patternmatch::appendNameTypes(Typemap::MapType M) {
  for (auto &&T : M) {
    if (this->NameTypes.count(T.first))
      errorExit(this->Location, "multiple declarations of same name");
    this->NameTypes[T.first] = T.second;
  }
}

WildcardPatternmatch::WildcardPatternmatch(SourceLocation L)
    : Patternmatch(L) {}

void WildcardPatternmatch::matchType(std::shared_ptr<Typedef> T) {
  this->Patternmatch::matchType(T);
}

LongIdentifierPatternmatch::LongIdentifierPatternmatch(SourceLocation L,
                                                       const Typemap &M,
                                                       std::string &&ID)
    : Patternmatch(L), ConstnameTypes(M), ID(std::move(ID)) {}

void LongIdentifierPatternmatch::matchType(std::shared_ptr<Typedef> T) {
```

```
  this->Patternmatch::matchType(T);
  auto CT = this->ConstnameTypes.get(this->ID);
  if (!CT) {
    this->NameTypes[std::move(this->ID)] = T;
  } else {
    if (!CT->isType())
      fatalExit("unexpected value: does not have a type");
    if (CT->isTypeMissingArgument())
      errorExit(this->Location, "incomplete value, missing argument");
    if (T->cannotBeAssignWith(*CT, nullptr, false))
      errorExit(this->Location, "pattern does not match type");
  }
}

std::shared_ptr<Typedef> LongIdentifierPatternmatch::getPreeMatchPatternType() {
  if (this->IsTypeMatched)
    return this->PatternType;

  auto CT = this->ConstnameTypes.get(this->ID);
  if (!CT)
    return nullptr;
  return CT;
}

TuplePatternmatch::TuplePatternmatch(SourceLocation L) : Patternmatch(L) {}

void TuplePatternmatch::append(std::shared_ptr<Patternmatch> M) {
  Matches.push_back(M);
}

void TuplePatternmatch::matchType(std::shared_ptr<Typedef> T) {
  this->Patternmatch::matchType(T);
  if (T->getKind() != Typedef::Product)
    errorExit(this->Location, "pattern does not match type");
  auto &&Defs = static_cast<ProductTypedef *>(T.get())->getTypedefs();
  auto Size = this->Matches.size();
  if (Defs.size() != Size)
    errorExit(this->Location, "unexpected number of tuple elements");
  for (size_t I = 0; I < Size; ++I)
    this->Matches[I]->matchType(Defs[I]);
}

std::shared_ptr<Typedef> TuplePatternmatch::getPreeMatchPatternType() {
  if (this->IsTypeMatched)
    return this->PatternType;

  auto Ret = std::make_shared<ProductTypedef>();
  for (auto &&M : this->Matches) {
    auto T = M->getPreeMatchPatternType();
    if (!T)
      return nullptr;
    Ret->append(T);
  }
  return Ret;
}

```

```
Typemap::MapType TuplePatternmatch::getNameTypes() {
  for (auto &&M : this->Matches)
    this->appendNameTypes(M->getNameTypes());
  return this->Patternmatch::getNameTypes();
}

ListPatternmatch::ListPatternmatch(SourceLocation L) :
    TuplePatternmatch(L) {}

void ListPatternmatch::matchType(std::shared_ptr<Typedef> T) {
  this->Patternmatch::matchType(T);

  if (T->cannotBeAssignWith(*getListTypedef(), nullptr, false))
    errorExit(this->Location, "expected 'list' type specificatin");
  assert(T->getKind() == Typedef::Datatype);

  auto List = static_cast<DatatypeTypedef &>(*T.get());
  auto Type = List.getParams().front().second;
  for (auto &&M : this->Matches)
    M->matchType(Type);
}

void ListPatternmatch::append(std::shared_ptr<Patternmatch> M) {
  auto T = M->getPreeMatchPatternType();
  if (this->Matches.size()) {
    auto Type = this->Matches.front()->getPreeMatchPatternType();
    if (Type->cannotBeAssignWith(*T, nullptr, false) &&
        T->cannotBeAssignWith(*Type, nullptr, false))
      errorExit(this->Location, "contradicting pattern types in
          list pattern");
  }
  this->TuplePatternmatch::append(M);
}

std::shared_ptr<Typedef>
    ListPatternmatch::getPreeMatchPatternType() {
  if (this->IsTypeMatched)
    return this->PatternType;

  auto Ret = getListTypedef();
  if (!this->Matches.size())
    return Ret;

  std::shared_ptr<Typedef> ListType;
  for (auto &&M : this->Matches) {
    auto T = M->getPreeMatchPatternType();
    if (!T)
      return nullptr;
    if (!ListType)
      ListType = T;
    else if (ListType->cannotBeAssignWith(*T, nullptr, false) &&
             T->cannotBeAssignWith(*ListType, nullptr, false))
      return nullptr;
    if (!T->isTypeComplete())
      ListType = T;
  }
  Ret->setArgument(this->Location, ListType);
  return Ret;
}

LiteralPatternmatch::LiteralPatternmatch(SourceLocation L,
                                         std::shared_ptr<Typedef>
```

```
                                            T)
    : Patternmatch(L), LiteralType(T) {}

void LiteralPatternmatch::matchType(std::shared_ptr<Typedef> T) {
  this->Patternmatch::matchType(T);
  if (T->cannotBeAssignWith(*this->LiteralType, nullptr, false))
    errorExit(this->Location, "pattern does not match");
}

std::shared_ptr<Typedef> LiteralPatternmatch::getPreeMatchPatternType() {
  return this->LiteralType;
}

ApplyPatternmatch::ApplyPatternmatch(SourceLocation L, SourceLocation R,
                                     std::shared_ptr<Patternmatch> LHS,
                                     std::shared_ptr<Patternmatch> RHS)
    : Patternmatch(L), Left(LHS), Right(RHS), RightLocation(R) {}

static std::shared_ptr<Typedef> vectorGetVariableTypedef(
    const std::vector<std::pair<std::string, std::shared_ptr<Typedef>>> &V,
    const std::string &VarName) {
  for (auto &P : V)
    if (VarName == P.first)
      return P.second;
  ssml::fatalExit("didnt find variable type name " + VarName + " in vector");
  return nullptr;
}

std::shared_ptr<Typedef> ApplyPatternmatch::match(
    const std::vector<std::pair<std::string, std::shared_ptr<Typedef>>> &V,
    std::shared_ptr<Typedef> Match) {
  switch (Match->getKind()) {
  case Typedef::SimpleBuildin:
    return this->simpleBuildinMatch(V, Match);
  case Typedef::Product:
    return this->productMatch(V, Match);
  case Typedef::Function:
    return this->functionMatch(V, Match);
  case Typedef::Tuple:
    fatalExit("unexpected tuple type discovered as type parameter");
    break;
  case Typedef::Datatype:
    return this->datatypeMatch(V, Match);
  case Typedef::Variable:
    return this->variableMatch(V, Match);
  case Typedef::DatatypeInstance:
    fatalExit("unexpected datatype instance discovered as type parameter");
  }
  return nullptr;
}

std::shared_ptr<Typedef> ApplyPatternmatch::simpleBuildinMatch(
    const std::vector<std::pair<std::string,
```

```
        std::shared_ptr<Typedef>>> &V,
    std::shared_ptr<Typedef> Match) {
  return Match;
}

std::shared_ptr<Typedef> ApplyPatternmatch::productMatch(
    const std::vector<std::pair<std::string,
        std::shared_ptr<Typedef>>> &V,
    std::shared_ptr<Typedef> Match) {
  auto &Prod = static_cast<ProductTypedef &>(*Match.get());
  auto Ret = std::make_shared<ProductTypedef>();
  auto Defs = Prod.getTypedefs();
  for (auto &&D : Defs)
    Ret->append(this->match(V, D));
  return Ret;
}

std::shared_ptr<Typedef> ApplyPatternmatch::functionMatch(
    const std::vector<std::pair<std::string,
        std::shared_ptr<Typedef>>> &V,
    std::shared_ptr<Typedef> Match) {
  auto &Fun = static_cast<FunctionTypedef &>(*Match.get());
  auto Ret = std::make_shared<FunctionTypedef>();
  auto Defs = Fun.getTypedefs();
  for (auto &&D : Defs)
    Ret->append(this->match(V, D));
  return Ret;
}

std::shared_ptr<Typedef> ApplyPatternmatch::datatypeMatch(
    const std::vector<std::pair<std::string,
        std::shared_ptr<Typedef>>> &V,
    std::shared_ptr<Typedef> Match) {
  auto &Datatype = static_cast<DatatypeTypedef &>(*Match.get());
  auto &Params = Datatype.getParams();
  auto Ret = std::shared_ptr<DatatypeTypedef>(
      new DatatypeTypedef(Datatype.getUserTypename(),
          Datatype.getParams()));
  auto Tup = std::make_shared<TupleTypedef>();
  for (auto &&P : Params) {
    if (!P.second)
      fatalExit("unexpected unused datatype parameter");
    Tup->append(this->match(V, P.second));
  }
  if (Params.size())
    Ret->setArgument(this->RightLocation, Tup);
  return Ret;
}

std::shared_ptr<Typedef> ApplyPatternmatch::variableMatch(
    const std::vector<std::pair<std::string,
        std::shared_ptr<Typedef>>> &V,
    std::shared_ptr<Typedef> Match) {
  auto &VarName =
      static_cast<VariableTypedef
          *>(Match.get())->getInternTypename();
  auto Def = vectorGetVariableTypedef(V, VarName);
  return Def;
}

void ApplyPatternmatch::matchType(std::shared_ptr<Typedef> T) {
  this->Patternmatch::matchType(T);
```

```
  auto L = this->Left->getPreeMatchPatternType();
  if (!L)
    errorExit(this->Location, "unknown type name");
  if (T->cannotBeAssignWith(*L, nullptr, false))
    errorExit(this->Location, "pattern does not match");

  if (L->getKind() != Typedef::DatatypeInstance)
    errorExit(this->Location, "does not take argument");
  auto LHS = static_cast<DatatypeInstanceTypedef *>(L.get());

  auto Arg = LHS->getArgumentTypedef();
  if (!Arg)
    errorExit(this->Location, "does not take argument");

  if (!T->getKind() == Typedef::Datatype)
    fatalExit("expected match type to be datatype typedef");
  auto TT = static_cast<DatatypeTypedef *>(T.get());

  auto ArgDef = this->match(TT->getParams(),
      LHS->getArgumentTypedef());
  LHS->setArgument(this->RightLocation, ArgDef);
  if (T->cannotBeAssignWith(*LHS, nullptr, false))
    errorExit(this->Location, "pattern does not match");

  this->Right->matchType(ArgDef);
}

Typemap::MapType ApplyPatternmatch::getNameTypes() {
  this->appendNameTypes(this->Right->getNameTypes());
  return this->Patternmatch::getNameTypes();
}

std::shared_ptr<Typedef>
    ApplyPatternmatch::getPreeMatchPatternType() {
  if (this->IsTypeMatched)
    return this->PatternType;

  assert(!!this->Left);
  auto Type = this->Left->getPreeMatchPatternType();
  if (!Type)
    return nullptr;

  auto Ret =
      std::shared_ptr<Typedef>(Type->clone());
  auto R = this->Right->getPreeMatchPatternType();
  if (R)
    Ret->setArgument(this->RightLocation, R);
  return Ret;
}
```

Listing 83: lib/Typecheck/Typemap.cpp

```
#include "Typemap.h"

#include "Typedef.h"

using namespace ssml::typecheck;

void Typemap::insert(std::string K, ValueType V, size_t Scope) {
  Maps[Maps.size() - 1 - Scope][std::move(K)] = V;
}
```

```

auto Typemap::get(const std::string &K) const -> ValueType {
  for (auto It = Maps.rbegin(), End = Maps.rend(); It != End; ++It)
    if (It->count(K))
      return ValueType(It->find(K)->second->clone());
  return nullptr;
}

void Typemap::mergeWith(MapType M, size_t Scope) {
  for (auto &&T : M)
    this->insert(std::move(T.first), T.second, Scope);
}

void Typemap::enterScope() {
  Maps.push_back(MapType());
}

void Typemap::leaveScope() {
  Maps.pop_back();
}
```