Source-Level Debugging

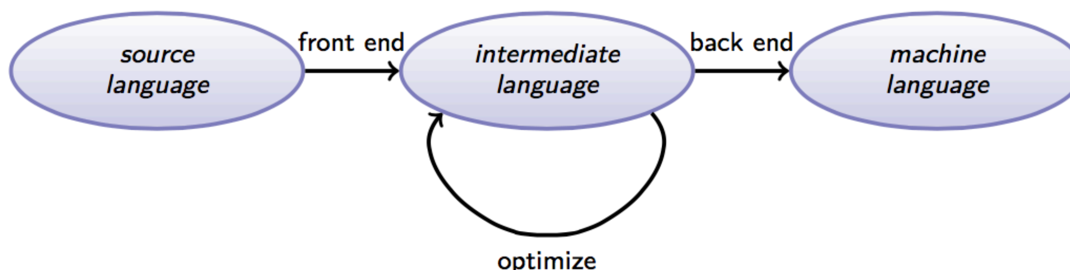
Vedant Raiththa
Mentored by: Matthew Fluet

Introduction

MLton is an open-source, whole-program, optimizing compiler for the Standard ML programming language. Standard ML is a strict, statically typed, functional programming language with type inference, abstract data types, a sophisticated module system, garbage collection, and many other features. As a high-level language with advanced programming-language features, Standard ML is a challenge to implement efficiently. MLton uses whole-program compilation to provide both advanced programming-language features and superior performance. Whole-program compilation gives the compiler maximal information about the program being compiled, which, in turn, allows the compiler to eliminate or mitigate the performance cost of using advanced programming-language features. Furthermore, whole-program compilation simplifies the compiler itself, making a whole-program compiler a good pedagogical platform and an excellent research framework in which to rapidly implement and evaluate new analyses, optimizations, and runtime-system improvements.
Debugging is a fact of programming life. Unfortunately, MLton provides little to no source-level debugging support. This project aims to add basic to intermediate source-level debugging support to the MLton compiler. MLton already supports source-level profiling, which can be used to attribute bytes allocated or time spent in source functions. It should be relatively straightforward to leverage this source-level information into basic source-level debugging support, with the ability to set/unset breakpoints and step through declarations and functions. It may be possible to also provide intermediate source-level debugging support, with the ability to inspect in-scope variables of basic types (e.g., types compatible with MLton's foreign function interface).

Background

A source code, which is usually a high-level program, goes through various stages before it is translated to low-level native assembly code (machine language).



MLton uses a number of intermediate languages (IL) from the input source program, which is SML, to low-level assembly code. The intermediate languages are listed in the order in which they are translated [15].

1. Abstract Syntax Tree (AST) [16]: This is produced by front end. It parses each token and creates a tree representation of all the instructions. Hence the name Abstract Syntax Tree.
2. CoreML [17]: CoreML is a polymorphic, higher-order language. It is explicitly typed and has nested patterns. DeadCode [24] optimization is applied in this pass.
3. XML [18]: XML is polymorphic, higher-order, with flat patterns. Each XML expression is annotated with its type. Polymorphic generalization is made explicit through type variables annotating val and fun declarations. Polymorphic instantiation is made explicit by specifying type arguments at variable references. Optimization applied in this pass are XMLSimplifyTypes [25] and XMLShrink [26]
4. SXML [19]: SXML is simple typed version of XML.
5. SSA [20]: SSA is a first order intermediate language. This is the primary language used for major optimizations. It is named after Static Single Assignment form. The key characteristic of this form is that each variable is defined before it used and it is assigned exactly once. This is also the pass in which the basic blocks are formed. Every basic block is marked with *Enter/Leave* statements, which will play a key role in generating debug information.
6. SSA2 [21]: This is a slight variant of SSA. The prime difference being that it includes mutable fields in objects and makes the vector type constructor.
7. RSSA [22]: This is an IL that makes representation explicit. This contains the actual source information in data structures which is referenced in further stages
8. Machine [23]: Not to be confused with Machine level language, this is the last IL which transfers the resultant code along with context to codegen for further processing.

The codegens then convert the program into low-level assembly code that can be executed natively by the operating system.

MLton has a built in profiling infrastructure, which keeps tracks of bytes allocated, execution counts, or time spent in each function [3]. MLton introduces several Enter and Leave statements to keep track of the function calls. These Enter/Leave are acknowledged, but ignored by most of the compiler. Just after the RSSA pass, before translating to Machine IL [7], the profiler uses these Enter/Leave statements to determine the scope of each point in the program. The profiler then removes these Enter/Leave statements and adds profiling information based on the selected mode of profile. There is information such as the source function name, lexical nesting and file position as well which is encoded into with the Enter/Leave statements. The profiler then uses this information at run time to calculate and give appropriate analysis.
Below are intermediate files generated from a simple program.
Command used:

```
      $ mlton -codegen llvm -profile label -keep g -keep o -prefer-abs-paths true -keep ssa
-keep rssa -keep machine simple_add.sml
```

Source code (simple_add.sml)

```
datatype num = zero | Succ of num
datatype bool = True | False

fun add (n1, n2) =
    case n1 of
         zero => n2
       | Succ m1 => Succ (add (m1, n2))

val one = Succ zero
val two = add (one, one)
val four = add (two, two)
```

Excerpt from simple_add.ssa

```
fun add_0 (x_75: num_0, x_74: num_0): {raises = None, returns = Some (num_0)} = L_76 ()
  L_76 ()
    Enter add /Users/vedant/Desktop/lectures/project-
mtf/my_examples/simple_add_profile_label/simple_add.sml: 4
    case x_74 of
      zero_0 => L_78 | Succ_0 => L_77
  L_78 ()
    Leave add /Users/vedant/Desktop/lectures/project-
mtf/my_examples/simple_add_profile_label/simple_add.sml: 4
    return x_75
  L_77 (x_76: num_0)
    add_0 (x_75, x_76) NonTail {cont = L_79, handler = Dead}
  L_79 (x_77: num_0)
    x_78: num_0 = Succ_0 (x_77)
    Leave add /Users/vedant/Desktop/lectures/project-
mtf/my_examples/simple_add_profile_label/simple_add.sml: 4
    return x_78
```

Excerpt from simple_add.rssa

```
fun add_0 (x_6: Objptr (opt_7), x_5: Objptr (opt_7)): {raises = None,
                                                       returns = Some (Objptr (opt_7))} =
L_13 ()
  L_13 () Jump =
    ProfileLabel MLtonProfile9
    x_7: Word32 = CPointer_lt (StackLimit, StackTop)
    switch {test = x_7, default = None, cases = ((0x0, L_15), (0x1, L_14))}
  L_15 () Jump =
    ProfileLabel MLtonProfile6
    switch {test = Cast (x_5, Bits64),
           default = Some L_16,
           cases = ((0x1, L_17))}
  L_16 () Jump =
    ProfileLabel MLtonProfile4
    x_8: Objptr (opt_7) = OP (x_5, 0): Objptr (opt_7)
    add_0 (x_6, x_8) NonTail {cont = L_18, handler = Dead}
  L_18 (x_9: Objptr (opt_7)) Cont {handler = Dead} =
    ProfileLabel MLtonProfile3
    x_10: Word32 = CPointer_lt (Limit, Frontier)
    switch {test = x_10, default = None, cases = ((0x0, L_20), (0x1, L_19))}
  L_20 () Jump =
    ProfileLabel MLtonProfile0
    x_11: Objptr (opt_7) = Object {header = 0xF, size = 16}
    OP (x_11, 0): Objptr (opt_7) = x_9
    return (x_11)
  L_19 () Jump =
    ProfileLabel MLtonProfile2
    CCall {args = (<GCState>, 0x0: Word64, 0x0: Word32),
          func = {args = (GCState, Word64, Word32),
                 convention = cdecl,
                 kind = Runtime {bytesNeeded = None,
                               ensuresBytesFree = true,
                               mayGC = true,
                               maySwitchThreads = false,
                               modifiesFrontier = true,
                               readsStackTop = true,
                               writesStackTop = true},
                 prototype = {args = (CPointer, Word64, Int32), res = None},
                 return = Bits0,
                 symbolScope = private,
                 target = GC_collect},
          return = Some L_21}
  L_21 () CReturn {func = {args = (GCState, Word64, Word32),
                         convention = cdecl,
                         kind = Runtime {bytesNeeded = None,
                                       ensuresBytesFree = true,
                                       mayGC = true,
                                       maySwitchThreads = false,
                                       modifiesFrontier = true,
                                       readsStackTop = true,
                                       writesStackTop = true},
                         prototype = {args = (CPointer, Word64, Int32),
                                     res = None},
                         return = Bits0,
                         symbolScope = private,
                         target = GC_collect}} =
    ProfileLabel MLtonProfile1
    L_20 ()
```

Excerpt from simple_add.machine

```
ProfileInfo:
{frameSources = (3, 3, 3, 3, 4, 3, 2, 2, 2),
 labels = ({label = MLtonProfile22, sourceSeqsIndex = 3},
          {label = MLtonProfile21, sourceSeqsIndex = 3},
          {label = MLtonProfile20, sourceSeqsIndex = 3},
          {label = MLtonProfile19, sourceSeqsIndex = 3},
          {label = MLtonProfile18, sourceSeqsIndex = 3},
          {label = MLtonProfile17, sourceSeqsIndex = 3},
          {label = MLtonProfile16, sourceSeqsIndex = 3},
          {label = MLtonProfile15, sourceSeqsIndex = 3},
          {label = MLtonProfile14, sourceSeqsIndex = 3},
          {label = MLtonProfile13, sourceSeqsIndex = 3},
          {label = MLtonProfile12, sourceSeqsIndex = 3},
          {label = MLtonProfile11, sourceSeqsIndex = 3},
          {label = MLtonProfile10, sourceSeqsIndex = 4},
          {label = MLtonProfile9, sourceSeqsIndex = 2},
          {label = MLtonProfile8, sourceSeqsIndex = 2},
          {label = MLtonProfile7, sourceSeqsIndex = 2},
          {label = MLtonProfile6, sourceSeqsIndex = 2},
          {label = MLtonProfile5, sourceSeqsIndex = 2},
          {label = MLtonProfile4, sourceSeqsIndex = 2},
          {label = MLtonProfile3, sourceSeqsIndex = 2},
          {label = MLtonProfile2, sourceSeqsIndex = 2},
          {label = MLtonProfile1, sourceSeqsIndex = 2},
          {label = MLtonProfile0, sourceSeqsIndex = 2}),
 names = (<unknown>, <gc>, <main>, add       simple_add.sml: 4),
 sourceSeqs = ((0), (1), (3), (2), ()),
 sources = ({nameIndex = 0, successorsIndex = 4},
           {nameIndex = 1, successorsIndex = 4},
           {nameIndex = 2, successorsIndex = 2},
           {nameIndex = 3, successorsIndex = 2})}
L_13: {kind = Jump,
       live = (SP(8): Objptr (opt_7), SP(0): Objptr (opt_7)),
       raises = None,
       returns = Some (SP(0): Objptr (opt_7))}
    ProfileLabel MLtonProfile9
    RW32(0): Word32  = CPointer_lt (OQ (<GCState>, 24): CPointer, <StackTop>)
    switch {test = RW32(0): Word32,
            default = None,
            cases = ((0x0, L_15), (0x1, L_14))}
add_0: {kind = Func,
        live = (SP(8): Objptr (opt_7), SP(0): Objptr (opt_7)),
        raises = None,
        returns = Some (SP(0): Objptr (opt_7))}
    Goto L_13
```

Excerpt from simple_add.1.ll

```
L_76:
        ; ProfileLabel MLtonProfile115
        ; ProfileLabel (add /Users/vedant/Desktop/lectures/project-
mtf/my_examples/simple_add_profile_label/simple_add.sml: 4)
        ; RW32(0): Word32  = CPointer_lt (OQ (<GCState>, 24): CPointer, <StackTop>)
        %r135 = bitcast %struct.GC_state* @gcState to %Pointer
        %r136 = getelementptr inbounds %Pointer %r135, i32 24
        %r137 = bitcast %Pointer %r136 to %CPointer*
        %r138 = load %CPointer* %r137
        %r139 = load %Pointer* %stackTop
        %r141 = icmp ult %Pointer %r138, %r139
        %r140 = zext i1 %r141 to %Word32
        store %Word32 %r140, %Word32* %regW32_0
        ; switch {test = RW32(0): Word32, default = None, cases = ((0x0, L_203), (0x1,
L_202))}
        %r142 = load %Word32* %regW32_0
        %r143 = trunc %Word32 %r142 to i1
        br i1 %r143, label %L_202, label %L_203

add_0:
        ; Goto L_76
        br label %L_76
```

The LLVM [28] is a compiler infrastructure project. It started as a project called Low Level Virtual Machine that was developed to research related with dynamic compilation techniques for programming languages. It has greatly evolved now having much more functionality and different objectives, however it still uses its old name. LLVM now incorporates various tools and acts as an intermediate layer where it is used as an intermediate language. LLVM is written in C++ and is used for compiling, linking and run-time optimization for various programming languages.

LLVM is written in C++ and is designed for compile-time, link-time, run-time, and "idle-time" optimization of programs written in arbitrary programming languages.
LLDB [9] is a textual debugger that is robust and high-performance. It is developed using the modular pattern, reusing components that highly leverage existing libraries in the larger LLVM Project, such as the Clang expression parser and LLVM disassembler.
Behind the scenes, to leverage existing infrastructure of clang architecture, LLDB converts debug information into clang types. This creates a level of abstraction for LLDB and harnesses the latest C, C++, Objective C and Objective C++ language features and runtimes in expressions without having to maintain or re-implement the functionality provided by them. This also helps in leveraging the compiler to take care of all ABI details when making functions calls for expressions or when disassembling instructions and extracting instruction details

Some interesting facts about LLDB [8], that makes it a good choice for MLton are:

- It supports a variety of basic debugging features such as reading DWARF, supporting step, next, finish, backtraces, etc.
- Debug information from object files is extracted incrementally by the Debug symbol file parsers. It currently supports Mach-O & DWARF symbol tables.
- Debugger plug-ins implement the host and target specific functions required to debug.
- It provides disassembly plug-ins for various architectures.
- It provides a framework API to the library
- It provides a command line debugger

LLDB also acts as a shell interpreter for Python functions [14]. So, using python commands to inspect the debugging environment at run time can use the LLDB API's directly from the command line.

Extensive documentation has been provided on the following links, which helps to gain a better insight into the workings of LLDB and Debug information that is injected into the LLVM code:

- http://llvm.org/docs/SourceLevelDebugging.html
- http://llvm.org/docs/LangRef.html
- http://llvm.org/docs/doxygen/html/classllvm_1_1Metadata.html
- http://llvm.org/docs/doxygen/html/namespacellvm_1_1dwarf.html
- http://llvm.org/test-doxygen/api/Dwarf_8h.html
- http://www.ibm.com/developerworks/aix/library/au-dwarf-debug-format/

Analysis

LLVM is one of the flavor by which we could generate an assembly file from source file (SML). Keeping this in mind, it would be a good idea to inject the debug information in the LLVM code, which could later be used by the debugger (LLDB) to process.

LLVM

Debugging information used by LLVM has been abstracted in such a way that the compiler can use the debugging information without knowing anything about the debugging information. Especially, the use of metadata as the base class avoids duplication of the debugging information through out the program. Also, the debugging information related to some dead code which will be removed in the global dead code elimination pass is also removed while removing the code associated with it.
Most of the debugging information, like, type definitions, functions, variables, source position, etc.), is inserted by the language in the front-end phase of compilation by lexer and parser, typically, in the form of LLVM metadata. Using the standards for representing debugging information like, Stabs or DWARF, the debug information is designed to be independent from the target debugger. A debugger uses these abstract objects to form stack traces, show information about local variables, etc.
These Debug information descriptors are child nodes of the base class Metadata.
LLVM uses several intrinsic functions (name prefixed with "llvm.dbg") to provide debug information at various points in generated code.

LLDB

LLDB is an open-source debugger that has a Read-Eval-Print-Loop, along with C++ and Python plugins. Once you are inside LLDB shell, there are couple of things you could do. You could set the target executable, if you haven't already done so while entering the shell using

```
(lldb) set target simple_add
```

There is also a help command that prints a list of command you could use. LLDB also supports auto-completion feature that is activated by pressing the tab key. The LLDB shell also has its own history, so you could access the commands previously used. Common commands that are useful are `print, run, step, continue, exit, breakpoint set, breakpoint list, breakpoint delete`.
While the program is being executed if you give the interrupt signal (^C), it will pause the program execution and point to the nearest breakpoint location. This breakpoint need not be active.
Some commands also have shortcuts to activate them like `br s` is for `breakpoint set`. You can also create variables on the go in the REPL environment.
LLDB uses a generic pass to decode the information that represents type information, function, variables, namespaces, etc. This allows the use of arbitrary source-language semantics and type-systems, as long as there is a module written for the target debugger to interpret the information. However, the LLDB assumes some basic minimal information related to the source-level language being debugged. Some common features that are assumed by the LLDB are source files, and program objects.

MLton

When generating C code from SML, as C stack is not structured to handle deeply nested recursive functions. SML, being a functional language, relies heavily on recursion. So, to address this

shortcoming, MLton uses a code module that acts as a trampoline for SML stack. This code basically creates an array of function pointers with a lookup table used to reference each function. Now, the execution consists of an infinite while loop that indirectly calls the function and handles the control flow manually. This ingenious approach however has some caveats. Since, C does not have any knowledge of the workings of the code, we are not able to leverage most of the C complier features, like optimizations, call stack and debugging information.

Hypothesis

As there is little control given by SML to C, we can't use the debugging functionalities that are implemented by the default compiler. Consequently, we can't directly put hooks in LLVM code for LLDB. Generally, most of the debugging information is inserted in front-end, we will need to insert similar information like metadata ourselves, and persist it till the end. This is a non-trivial task considering the complexity of the SML architecture.
Furthermore, the SML stack is different from C stack. So, we will need to teach the debugger to walk the SML stack.

Solution Plan

1. Familiarize with LLVM and the metadata generated
    a. Familiarize with syntax and architecture
    b. Figure out bare minimum needed data needed for LLDB to run
2. Inject hard coded code inside LLVM generated file for LLDB to work with.
3. Teach MLton to generate debug information
    a. Implement a Debug mode for MLton.
4. Ability to insert breakpoint at the correct location.
    a. Co-relate the code generated by LLVM (usable by LLDB) to source code.

Synthesis

Investigation

As the documentation on LLVM commands for LLDB is very limited, a lot of work is based on educated conjecture. I have used the code generated by clang for C as a guideline for injecting debug information inside the code generated for SML.
I started off by generating the debug information with a C program containing a simple return statement. For the debug information to be included, Clang has a command line switch with various options. I used the following command to generate the LLVM file for the basic C program.

```
$ clang –g -emit-llvm –S -o simple.ll simple.c
```

'-g' here tells the compiler to generate complete debug information associated with the program. '-S' like in GCC says emit textual assembly rather than assembled binary. '-emit-llvm' tells the compiler to emit the LLVM code instead native assembly file. Please note here that, Apple doesn't work off of released versions of LLVM/Clang and since MLton requires LLVM tools, I am using a released version of Clang (Clang 3.6), which is installed separately. (I have used homebrew to manage packages on OS X). So, once I generated the LLVM code which contained debug information, I started removing some information to determine which is the bare essential information which is required in order for LLDB to process it properly.

Armed with this information, I proceeded to insert this information inside MLton generated LLVM code. I excluded the basis library and wrote a small factorial program to simplify the LLVM code generated.
Every line in LLVM code generated is not a valid breakpoint because the code generated in LLVM doesn't have one to one correspondence with the source code, as there were several operations performed behind the scene by MLton to provide several features to the programmers. In the intermediate passes of MLton, most of the source information like variable names, etc. is lost. Furthermore, due to optimizations such as dead code elimination, code in-lining, etc., the code would be simplified in some areas and complicated in some areas. Hence, co relating each instruction with a source location posed as a quite complex problem. So, I was faced with 4 viable solutions for this problem:

1. Avoid all optimizations while debugging
2. Insert a no operation instruction at corresponding points in the code
3. Insert the breakpoint marker on each statement in a basic block.
4. Insert a dummy instruction which LLVM could not optimize away

Implementation

Now, using my findings, I had to integrate this functionality in MLton. I started by creating a command line switch called ProfileDebug and I had the system treat it the same as ProfileLabel, so, I could take advantages of ProfileLabel yet not introduce and side effects for my code. The ProfileDebug switch is also used to set the environmental flags needed for its proper function. For example, LLDB needs absolute file paths for debugging. So, we force the compiler to use absolute paths. On the other hand, the LLVM tools like optimizer and linker require the intermediate files generated in the same directory to persist the debug information. So, we set a flag in MLton to store the intermediate files in the same directory. By default, they are kept in an arbitrary hidden directory.
Major parts of the changes were made to LLVM-Codegen. The way MLton is modeled it follows various chunking strategies. More details about that can be found here [11]. This is important because, to reduce execution time, MLton creates multiple source files for different chunks. So, we needed to treat each chunk as an independent unit while generating debugging information.
I have created a structure for representing the metadata information to be generated for the source code. It generates all the debug information as it processes the code. It keeps all the information in memory as key-value pair with some additional information. After the chunk is processed completely, it dumps the accumulated information at the end of the file and clears it. The datatype used for representing debugging information was designed after the LLVM Metadata structure [12] for consistency and defining constraints for usage.

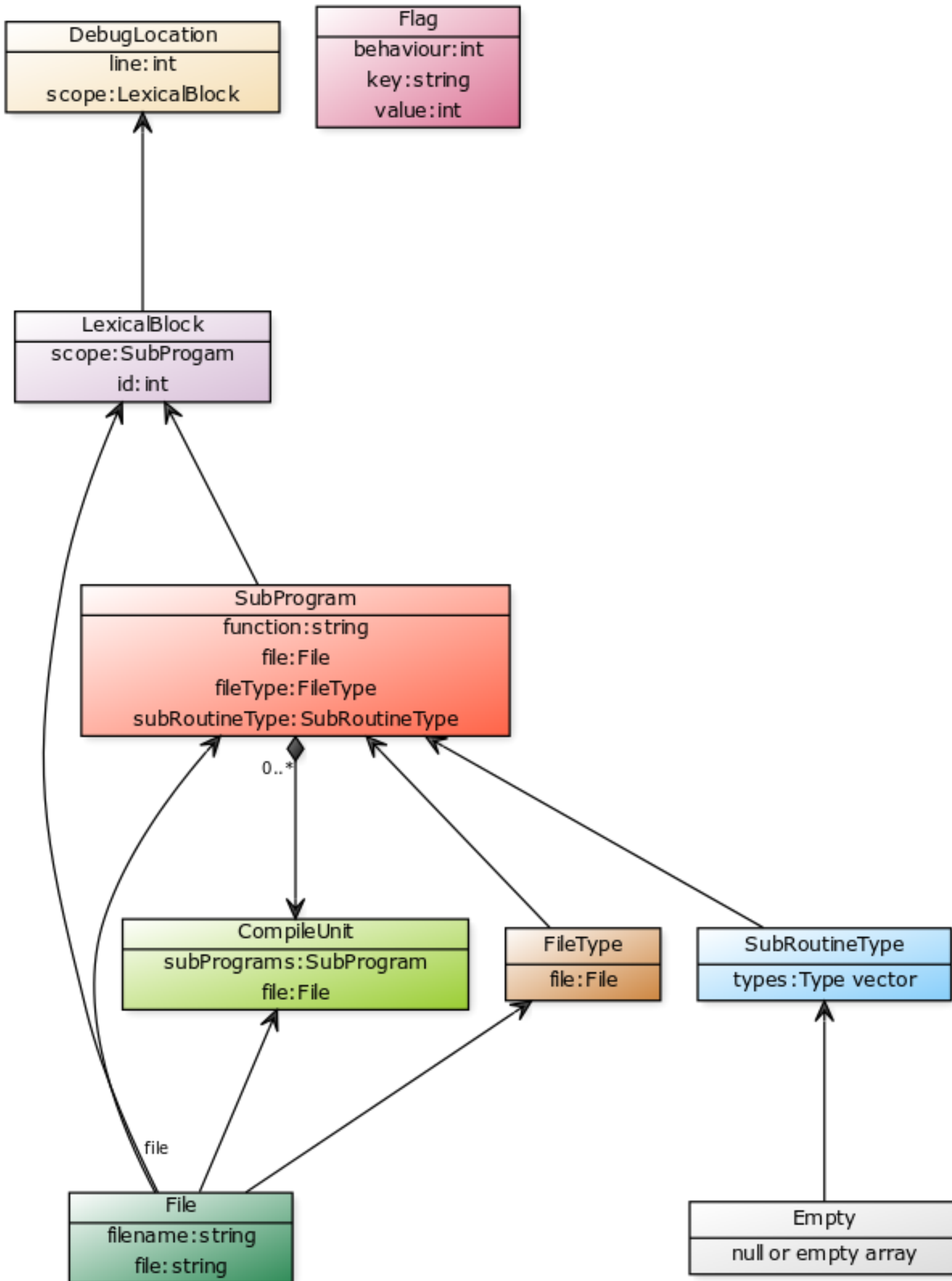


Figure 1: Class Diagram of underlying data structure

The Class diagram represents the structure used by me to store the metadata information in code.

For extracting source information from ProfileLabels, we have to extract the label index from them. This label index points us to the corresponding index in sourceSeqs vector, which in turn returns a vector of indexes in source vector. These source indexes are mapped to sourceInfos that contains the source location information like filepath, line number, etc. This source information is used for create debug information and generate the metadata related to it.
A NOP instruction maintains the current program state but advances the clock cycle and program counter. This instruction is ideal to put a breakpoint at. Now, LLVM doesn't have any NOP instruction. So, I had to introduce a dummy instruction similar to such an operation. A dummy instruction does not change the state of the program, however it performs some actions that does not have any side effect, but performs some instruction that consumes some trivial amount of clock cycles. So after some considerations, I chose the following dummy instruction, as it was the simplest, cost effective and reliable.

```
store i32 1, i32* @dbgState
```

The above instruction performs a write operation to an external global variable. This external global variable is not used by anybody else, so it doesn't cause any side effects. Being a write to an external variable, LLVM can't optimize it.
I inserted this instruction at each basic block's entry points. Since, MLton already has a framework which introduced a label every time it Enters/Leaves a function for profiling purpose, I piggybacked onto these labels and inserted this instruction following these labels. This did the trick and I got a working model that could be used by LLDB.
However, I found an issue here. As per the structure of information available in MLton, when there are various nested immediate function calls, the source locations are stacked together in a vector. So, based on this, when I generated multiple 'store' instructions, LLVM optimized it and only considered the last instruction, as that would be the only one that would matter. To address this issue, I created an external array instead of just a single variable, and updated sequential locations in the array in different store instructions. I had to ensure that the size of the array was big enough that a contiguous chunk of store statements did not exceed it. In other words, I had to ensure there weren't any store statements updating the same variable inside one block of store statements.

```
%tmp = getelementptr inbounds [256 x i32]* @dbgState, i32 0, i32 0
store i32 1, i32* @dbgState
```

This approach works, however, it has a performance cost associated with it. Performance matrices are mentioned in the results section.

Shortcomings

The datatypes designed only have minimum attributes needed for functioning. They are relatively incomplete as there is little documentation available on the LLVM classes.
It runs into issues, when multiple LLVM files come into picture.
LLDB provide couple of ways to set breakpoints. One is by specifying the file name and line number and another is by specifying function name. Since, MLton doesn't explicitly use functions, refer trampoline in Analysis: MLton section, we can't take advantage of the later feature.

Results

The LLDB now is able to extract and use the debug information provided by the executable file that was generated using the updated codegen.

The user can set and unset breakpoints in the source code, using the source file and the corresponding line number, within LLDB using LLDB's commands. The user can also perform step operations in the executable as well.
The debug mode can be activated by using a command line switch '–profile debug'. This will allow you to set breakpoints at the function entry points. If you compile using '–profile-branch true' and '–profile-val true', you can set breakpoints at each non-trivial declaration (val statement) and branches (if/case branch).

You can set this breakpoint by using the filename and line on which the function is defined.
For example,

```
(lldb) breakpoint set –f simple_add.sml –l 4
```

This feature will help the developers to trace the program execution. It is particularly useful when you want to test if and when a function is executed as well as for debugging infinite recursion issues.
Also while the program is running inside the LLDB shell, you could give the interrupt signal (^C) that will pause the program execution at the nearest breakpoint.

For testing the code, I have used several scenarios:

- Simple basic code without including any libraries
- Simple basic code including the basis library
- Relatively complex code
- Relatively complex code using different chunkify modes

Excerpt from output on terminal

```
(lldb) breakpoint set -f simple_add.sml -l 4
Breakpoint 1: where = simple_add` + 85 at simple_add.sml:4,
address = 0x0000000100000f35
(lldb) run
Process 76299 launched:
'~/Desktop/projects/simple_add/simple_add.exe' (x86_64)
Process 76299 stopped
* thread #1: tid = 0x2e16af, 0x0000000100001148
simple_add at simple_add.sml:4, queue = 'com.apple.main-thread',
stop reason = breakpoint 1.1
    frame #0: 0x0000000100001148 simple_add at simple_add.sml:4
   1     datatype num = zero | Succ of num
   2     datatype bool = True | False
   3
-> 4     fun add (n1, n2) =
   5         case n1 of
   6              zero => n2
   7            | Succ m1 => Succ (add (m1, n2))
(lldb) continue
Process 76299 resuming
```

I discovered that creating and using 256 different variables were better in terms of performance. I have listed some stats below, for an arbitrarily complex program (mandelbrot).

	No debug info	Basic profile info	Basic profile info & profile-branch	Basic profile info & profile-branch & profile-val
Array of variables	13.13	38.97	165.15	292.17
Different variables	13.13	40.31	93.53	235.30

This abnormality can be attributed to LLVM's optimizer as the assembly file generated differs at several places.

Future Scope

This project could be extended to include other debugging functionalities like:

- Printing a stack trace at the current point in the program
- Inspect a variable which is in scope
- Evaluate simple expressions in the environment in which the program is halted. Also, we could give it ability to change the environment based on the evaluation

References

1. https://wiki.cs.rit.edu/bin/view/Main/Mtf:Source-levelDebuggingForMLton
2. http://mlton.org/
3. http://mlton.org/HowProfilingWorks
4. http://mlton.org/Profiling
5. http://llvm.org/docs/SourceLevelDebugging.html
6. http://lldb.llvm.org/
7. http://mlton.org/Machine
8. http://lldb.llvm.org/features.html
9. http://lldb.llvm.org/index.html
10. http://llvm.org/docs/LangRef.html
11. http://mlton.org/Chunkify
12. http://llvm.org/docs/doxygen/html/classllvm_1_1Metadata.html
13. https://github.com/MLton/mlton
14. http://lldb.llvm.org/python-reference.html
15. http://mlton.org/IntermediateLanguage
16. http://mlton.org/AST
17. http://mlton.org/CoreML
18. http://mlton.org/XML
19. http://mlton.org/SXML
20. http://mlton.org/SSA
21. http://mlton.org/SSA2
22. http://mlton.org/RSSA
23. http://mlton.org/Machine

24. http://mlton.org/DeadCode
25. http://mlton.org/XMLSimplifyTypes
26. http://mlton.org/XMLShrink
27. https://www.objc.io/issues/19-debugging/lldb-debugging/
28. https://en.wikipedia.org/wiki/LLVM