MLton Hacker Guide

Stephen Weeks, Jeff Murphy (UB)

February 13, 2019

This document describes how to hack MLton, a whole-program optimizing compiler for the Standard ML programming language. The MLton homepage is `http://www.mlton.org/MLton/`. The document contains an overview of the source tree, a description of the programming style used in MLton, and delves into the bowels of the compiler and associated tools.

This document is very incomplete.

Contents

Chapter 1

The sources

This section is an overview of the sources to the compiler and all of the associated tools. Here is a brief description of each element of the root source directory. Throughout the rest of this document, we will use pathnames that are relative to the source directory.

`basis-library`
: The basis library implementation.

`benchmark`
: Code and tests used for benchmarking MLton, SML/NJ, and Moscow ML.

`bin`
: Scripts for type checking the basis library, making rpms, running MLton, and running regression tests.

`doc`
: Sources for the user guide, hacker guide, web site, announcements, README.

`include`
: Include files needed for compiling C files generated by MLton.

`lib`
: SML library code, which is used in `mlton`, `mlprof`, and `benchmark`. There are also many generally useful libraries.

`Makefile`
: To make everything. This is only used when building rpms.

`man`
: Manual pages for `mlton` and `mlprof`.

`mllex`
: Lexer generator, taken and slightly modified from SML/NJ.

`mlprof`
: Profiler.

`mlton`
: Compiler.

`mlyacc`
: Parser generator, taken and slightly modified from SML/NJ.

`regression`
: Regression tests, about 150 SML files that are used to test the compiler.

`runtime`
: Runtime system, which includes the garbage collector and C libraries used in the basis (including the GMP used for `IntInf`).

Chapter 2

The basis library

The basis library is implemented with about 12,000 lines of SML code. There is roughly one file for each signature and structure that the library specification defines. The files are grouped in directories in the same way that the corresponding modules are grouped in the basis library documentation. Here is an overview of the `basis-library` directory.

`arrays-and-vectors general integer io list posix real system text`
: SML code for basis library modules.

`basis.sml`
: Automatically constructed by `bin/check-basis`. Used to type check the basis libary under SML/NJ.

`bind-basis`
: A list of the files (in order) that define what is exported by the basis library.

`build-basis`
: A list of the files (in order) used to construct the basis library.

`Makefile`
: Only has a target to clean the directory.

`misc`
: SML code that didn't fit anywhere else. In particular, the `Primitive` structure.

`mlton`
: The `MLton` structure, which is not part of the standard basis library. For more details on what `MLton` provides, see the MLton User Guide.

`sml-nj`
: The `SMLofNJ` and `Unsafe` structures, which are not part of the standard basis library.

`top-level`
: Files describing the overloads, infixes, modules, types, and values that the basis library makes available to user programs.

2.0.1 How MLton builds the basis environment

The `forceBasisLibrary` function in `mlton/main/compile.sml` builds the basis environment that is used to compile user programs. Conceptually, the basis environment is constructed in two steps. First, all of the files in `build-basis` are concatenated together and evaluated to produce an environment E. Then, all of the files in `bind-basis` are concatenated and evaluated in environment E to produce a new environment E', which is the top-level environment. Another way to view it is that every user program is prefixed by the following.

```
local
  <concatenate files in build-basis>
in
  <concatenate files in bind-basis>
end
```

This view is not strictly accurate because some of the files are not SML (they use the `_prim`, `_ffi`, and `_overload` syntaxes) and because SML does not allow local functor or signature declarations. Here is a description of the basis files that are not SML.

`misc/primitive.sml`
: Defines the `Primitive` structure, which binds (via the `_prim` syntax) all of the primitives provided by the compiler that the basis library uses.

`mlton/syslog.sml`
: Defines constants and FFI routines used to implement `MLton.Syslog`.

`posix/primitive.sml`
: Defines the `PosixPrimitive` structrue, which binds the constants and FFI routines used to implement the `Posix` structure.

`top-level/overloads.sml`
: Defines the overloaded variables available at the top-level the `_overload` syntax: `_overload` x: ty `as` y_0 `and` y_1 `and ...`

2.0.2 Modifying the basis library

If you modify the basis library, you should first check that your modifications are type correct using the `bin/check-basis` script. Since this MLton does not have a proper typechecker, this script uses SML/NJ. First, it concatenates the files as described in Section 2.0.1 into one file, `basis.sml`. It also replaces the nonstandard syntax (`_prim`, etc.) and declares the toplevel types to match MLton's (necessary since SML/NJ uses 31 bits while MLton uses 32). It then feeds `basis.sml` to SML/NJ. If there are no type errors, a message like the following will appear.

```
stdIn:12213.1-12213.14 Error: operator is not a function [tycon mismatch]
  operator: unit
```

```
  in expression:
    () ()
```

This error message is intentionally introduced by `check-basis` at the end of `basis.sml` to make it clear that SML/NJ reached the end of `basis.sml` and has hence type checked the entire basis.

Once you have a basis library that type checks, you need to create a new version of MLton that uses this library. MLton preprocess the basis library to create a `world.mlton` file that contains the basis environment. The `world.mlton` file is stored in the `lib` directory and is loaded by `mlton` when compiling a user program (see the `bin/mlton` script). To build a new `world.mlton`, run `make world` from within the sources directory.

2.0.3 The `misc` directory

`cleaner.sig`
: Functions for register "cleaning" functions to be run at certain times, in particular at program exit. The `TextIO` module uses these cleaners to ensure that IO buffers are flushed upon exit.

`suffix.sml`
: Code that is (conceptually) concatenated on to the end of every user program. It just calls `OS.Process.exit`. The `forceBasisLibrary` function ensures that `suffix.sml` is elaborated in an environment where the basis library `OS` structure is available.

`top-level-handler.sml`
: This defines the top level exception handler that is installed (via a special compiler primitive) in the basis library, before any user code is run.

2.0.4 Dead-code elimination

In order to compile small programs rapidly and to cut down on executable size, `mlton` runs a pass of dead-code elimination (`mlton/core-ml/dead-code.sig`) to eliminate as much of the basis library as possible. The dead-code elimination algorithm used is not safe in general, and only works because the basis library implementation has special properties:

- it terminates
- it performs no I/O
- it doesn't side-effect top-level variables

The dead code elimination simply includes the minimal set of declarations from the basis so that their are no free variables in the user program (or basis). Hence, if you do something like the following in the basis, it will break.

```
val r = ref 13
val _ = r := 14
```

The dead code elimination will remove the `val _ = ...` binding.

Chapter 3

The runtime

3.1 Notes

There are multiple, possibly orthogonal issues. Limit checks and garbage collections are a little overloaded in their roles, because they also support preemptive thread switching and interrupt handling. Forcing frontier to be 0 and hitting a limit check (even a zero byte limit check) will invoke the GC, which will switch to the pending thread.

Recall that a limit check with bytes = 0 really means a check for LIMIT_SLOP bytes (currently LIMIT_SLOP = 512).

3.2 Bootstrap

When you compile your SML code, it is translated to machine code using one of several backends. For an in-depth description of how SML is compiled and optimized refer to [Lei13]. We will look at the C translation of a trivial SML program starting at the backend once all optimization phases have completed. The trivial SML program is a single statement: `val a = 2`

When reading this section of the guide, it will be useful to save the above statement as "test.sml" and then compile that using "mlton -keep g test.sml" so that you can refer to the intermediate files "test.0.c" "test.1.c" and "test.2.c"

Refer to Figure 3.1 for an overview of how the compiler emits C code given SML code, and how control flows through the bootstrap process of the emitted code.

The emitted C code bootstrap at the bottom of "test.0.c" looks like this:

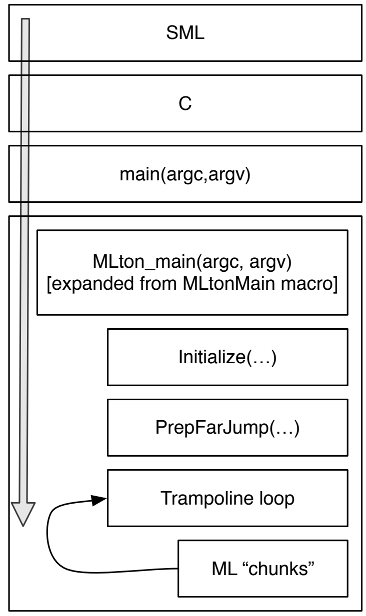

Figure 3.1: Runtime overview

```
MLtonMain (8, 0x7CB29B69, 136, TRUE, PROFILE_NONE, FALSE, 0, 218)
int main (int argc, char* argv[]) {
    return (MLton_main (argc, argv));
}
```

and contains a `main` routine that calls `MLton_main` which is created when the `MLtonMain` macro is expanded. `MLtonMain` is defined in `include/c-main.h` as a macro:

```
#define MLtonMain(al, mg, mfs, mmc, pk, ps, mc, ml)
```

and ultimately calls the routine `MLton_main (int argc, char* argv[])`
The parameters to the `MLtonMain` macro are:

al alignment width (`-align`)

mg a magic random number used for saving/restoring the world. This number is generated at compile time by `mlton/codegen/c-codegen/c-codegen.fun` and allows the application to save and restore its state (MLtonWorld)

mfs the maximum frame size

mmc whether or not the mutator marks cards. This is an optimization strategy used by the generational GC.

pk the kind of profiling to perform (compile time option)

ps whether stack profiling is enabled (`-profile-stack`)

mc the number of the first chunk to jump to

ml the function number in the chunk to jump to

The first six of these parameters are passed to `Initialize` (defined in `include/common-main.h`) while the final two (mc and ml) are passed to `PrepFarJump` (defined in `include/c-common.h`). `Initialize` sets variables in the `gcState` structure and then calls `MLton_init(argc, argv, &gcState)`.

`MLton_init` (`runtime/platform.c`) initializes the posix environment, the GC and processes the runtime command line arguments. Once `Initialize` completes, `MLton_main` continues and calls `PrepFarJump` to prepare to jump to the first chunk of the SML program. Alternatively, it will restore the saved world and restart from where the saved program left off. Finally, `MLton_main` goes into an infinite loop, jumping from chunk to chunk as the SML program executes.

Jumping between chunks is known as trampolining and this is done to avoid mapping highly recursive SML functions directly to C functions as this would exhaust the C stack (see §2.2.4 of [Lei13]). Trampolining involves selecting a chunk from the `cont` struct and then calling to that address (pointer). You will notice that, in our example above, `mc` is set to 0 and `ml` is set to 218. That means that `PrepFarJump` will select chunk 0 to execute and will set the next function within chunk zero to 218.

So walking through this, `PrepFarJump(0, 218)` will result in

```c
cont.nextChunk = (void *)Chunk0;
nextFun = 218; // note: unsynchronized global variable
```

`nextFun` is a global variable declared in `include/c-common.h`

The `Chunk0` symbol is declared in "test.0.c" via the `DeclareChunk (0)` line. This is a macro that expands to

```c
PRIVATE struct cont Chunk0(void);
```

The actual Chunk0 routine is defined in "test.2.c" via the line `Chunk (0)` which is another macro (defined in `include/c-chunk.h`) that expands to:

```

        DeclareChunk(0) {
                struct cont cont;
                Pointer frontier;
                uintptr_t l_nextFun = nextFun; // remember this is 218
                Pointer stackTop;
```

Note that a local copy of the nextFun variable is made. `DeclareChunk` is, you guessed it, a macro (defined in `include/c-common.h`) and results in the above expanding to:

```
PRIVATE struct cont Chunk0(void) {
                struct cont cont;
                Pointer frontier;
                uintptr_t l_nextFun = nextFun; // remember this is 218
                Pointer stackTop;
```

And so we finally have our `Chunk0` routine which is what we set `chunk.nextChunk` to above if you recall.

Given the above, the trampoline section of `MLton_main` (again, in `include/c-main.h`) will call

```
// equivalent in our example to cont = Chunk0();
cont=(*(struct cont(*)(void))cont.nextChunk)();
```

We will see, as we fully expand `Chunk0` how it ultimately returns a `cont` structure to allow us to trampoline to the next chunk. Also, we will see how each chunk routine is a large `switch` statement indexed by nextFun and so, architecturally, MLton aggregates SML functions into large C-functions where each SML function is one of the cases in the switch statement. This is how MLton minimizes the growth of the C-stack – recursive SML functions can be aggregated into the same C function and "call" each other by manipulating `nextFun` and `switch`ing between each other. `goto` is also used to switch between SML functions without incurring any C-stack growth.

Continuing on, we are now in the `Chunk0` routine which we see, from examining "test.2.c", continues past the `Chunk (0)` line as such:

```
Chunk (0)
        CPointer Q_0;
        CPointer Q_1;
        CPointer Q_2;
        .
        .
        .
ChunkSwitch (0)
case 5:
L_9:
        Push (-8);
        .
        .
        .
case 218:
        G(Word32, 0) = CPointer_lt (O(CPointer, GCState, 40), StackTop);
        BNZ (G(Word32, 0), L_8);
        G(Word64, 0) = CPointer_diff (O(CPointer, GCState, 1360), Frontier);
        G(Word32, 1) = WordU64_lt (G(Word64, 0), (Word64)(0x1090ull));
        BNZ (G(Word32, 1), L_8);
        goto L_2;
L_8:
        S(CPointer, 0) = 5;
        Push (8);
        FlushFrontier();
        FlushStackTop();
        GC_collect (GCState, (Word64)(0x1090ull), (Word32)(0x0ull));
        CacheFrontier();
        CacheStackTop();
        goto L_9;
EndChunk
```

Examining this routine, let's first look at the bottom `EndChunk` which is a macro (defined in `include/c-chunk.h`) and expands to:

```
                default:
                        /* interchunk return */
                        nextFun = l_nextFun;
                        cont.nextChunk = (void*)nextChunks[nextFun];
                        leaveChunk:
                                FlushFrontier();
                                FlushStackTop();
                                return cont;
                    } /* end switch (l_nextFun) */
                } /* end while (1) */
        } /* end chunk */
```

This results in `nextFun` (the global) being set to the next function in the switch statement to execute and then it sets `cont.nextChunk` to the next chunk (if we need to switch between C functions) and finally it flushes some registers and returns. Note that `nextChunks` is a list, indexed by SML function number, that lets us figure out which C-function contains the SML function we want to switch to. In our example, `nextChunks[218]` would contain a pointer to the C-function `Chunk0`. Since C-functions never call each other, but always `return` to the trampoline, the C-stack effectively does not grow while our program executes. If an SML function calls another SML function that is in the same C-function (Chunk0 in our case) we will not `return` but instead will "fall through" to the end of the `while` loop, taking us back to the top of the `switch` statement and then into the called SML function. In this way, SML functions can transition ("switch") between each other without any effect on the C-stack. Also note the label `leaveChunk` allows SML functions to jump out of the C function. The "end while (1)" refers to a while statement in the macro `ChunkSwitch` which we will now look at, before bringing this all together into a single C function.

Moving backwards, now, we look at the `CPointer` lines. These are variables that have been promoted to the top level by the AST pass of the compiler. Next we come to the `ChunkSwitch` statement. This is a macro (defined in `include/c-chunk.h`) that emits:

```
                CacheFrontier();
                CacheStackTop();
                while (1) {
                top:
                switch (l_nextFun) {
```

This fragment sets up the while loop that is referred to in the comment in `EndChunk` and we see the switch statement that keys on the value of `l_nextFun` which, if you remember, is 218 in our example. Now we can look at the entire function, with most of the larger macros (but not all of them) expanded. Notice that functions can jump directly to the top of the switch statement in order to move to a different SML function without incurring any C-stack cost.

```
PRIVATE struct cont Chunk0(void) {
    struct cont cont;
    Pointer frontier;
    uintptr_t l_nextFun = nextFun; // remember this is 218
    Pointer stackTop;

    CPointer Q_0;
    CPointer Q_1;
    CPointer Q_2;
        .
        .
        .
    CacheFrontier();
    CacheStackTop();
    while (1) {
       top:
       switch (l_nextFun) {
       case 5:
       L_9:
           Push (-8);
           .
           .
           .
       case 218:
           G(Word32, 0) = CPointer_lt (O(CPointer, GCState, 40),
              StackTop);
           BNZ (G(Word32, 0), L_8);
           G(Word64, 0) = CPointer_diff (O(CPointer, GCState, 1360),
              Frontier);
           G(Word32, 1) = WordU64_lt (G(Word64, 0), (Word64)(0x1090ull)
              );
           BNZ (G(Word32, 1), L_8);
           goto L_2;
       L_8:
           S(CPointer, 0) = 5;
           Push (8);
           FlushFrontier();
           FlushStackTop();
           GC_collect (GCState, (Word64)(0x1090ull), (Word32)(0x0ull));
```

```
37            CacheFrontier();
38            CacheStackTop();
39            goto L_9;
40        default:
41            /* interchunk return */
42            nextFun = l_nextFun;
43            cont.nextChunk = (void*)nextChunks[nextFun];
44            leaveChunk:
45                FlushFrontier();
46                FlushStackTop();
47                return cont;
48         } /* end switch (l_nextFun) */
49     } /* end while (1) */
50 } /* end chunk */
```

Some interesting things: note line 24 which is the piece of code that `MLton_main` ultimately executes the first time we enter the trampoline section after preparing the cont structure in `PrepFarJump`. Also, line 30 shows how SML functions that correspond to each case statement can jump around within the case statement itself – control is in no way linear in the emitted C code. We omitted the code at `L_2` for brevity, but you can look in "test.2.c" as a reference.

Also, note again line 16 which is the top of the switch statement. There is another macro, that isn't referenced in the example code we've shown here, that uses that label to switch back to the caller. The macro is `Return()` and it pops (but does not adjust the size of the stack) the caller off of the SML stack and then jumps to `top` in order to switch back ("return") to the calling function. The code for `Return()` follows. Note the setting of `l_nextFun`.

```
        l_nextFun = *(uintptr_t*)(StackTop - sizeof(void*));
        goto top;
```

SML functions themselves, once translated down to C, are essentially all pointer and memory manipulation. Let's look at the section of code for `case 218`.

```
1          case 218:
2              G(Word32, 0) = CPointer_lt (O(CPointer, GCState, 40),
                   StackTop);
3              BNZ (G(Word32, 0), L_8);
4              G(Word64, 0) = CPointer_diff (O(CPointer, GCState, 1360),
                   Frontier);
5              G(Word32, 1) = WordU64_lt (G(Word64, 0), (Word64)(0x1090ull)
                   );
6              BNZ (G(Word32, 1), L_8);
7              goto L_2;
8          L_8:
9              S(CPointer, 0) = 5;
10             Push (8);
11             FlushFrontier();
12             FlushStackTop();
13             GC_collect (GCState, (Word64)(0x1090ull), (Word32)(0x0ull));
14             CacheFrontier();
15             CacheStackTop();
16             goto L_9;
```

As we pointed out, once you get into the C code, it is all pointer and memory manipulation via a handful of macros. In the above code, we have four macros and two C-functions (that are inline-able and so should not affect the C-stack). The macros are defined in `include/c-chunk.h` and do the following:

G sets a global statically allocated variable for temporary use

O retrieves the value at a particular memory offset and casts it to a specified type/width

BNZ branch if not zero

For the `G` macro on line 2, we have the following expansion: `G(Word32, 0)` becomes `globalWord32[0]` which refers to a statically allocated array in "test.0.c" `PRIVATE Word32 globalWord32 [2]`. On the other side of the assignment we have `CPointer_lt (O(CPointer, GCState, 40), StackTop)` which is a C-function call and a macro that expands to:

```
G(Word32, 0) = CPointer_lt (O(CPointer, GCState, 40), StackTop);
// expands to:
globalWord32[0] = CPointer_lt ((*(CPointer*)((GCState) + (40))),
   StackTop);
```

which is comparing the value of the field that is 40 bytes into GCState with the value of StackTop. This offset is calculated by the compiler (see section 3.5) and should correspond to the fifth field (`GCStat.exnStack`) since our alignment was 8. So we are checking if exnStack < stackTop and if it is, globalWord32[0] is set to true (1). If it is true, we jump to L_8 because a GC is needed, otherwise we check (line 5) to see if there's at least 0x1090 bytes of space left before we hit the Frontier, and if that's not true we again need to GC, otherwise we can proceed to label L_2.

3.3 Object Header and Object Size Assertions

The MLton object header is 32 or 64 bits depending upon platform. If 64 bits, the upper 32 are zeros. The object header immediately precedes each object in memory. From object.h:

```
* 00        : 1
* 01 - 19   : type index bits, index into GC_state->objectTypes.
* 20 - 30   : counter bits, used by mark compact GC (initially 0)
*      31   : mark bit, used by mark compact GC (initially 0)
* 32 - 63   : 0wx00000000  (only w/ 64-bit header)
```

The object type index is used to make assertions about object sanity. For example, the GC_Thread structure has three members: two of size_t and one of objptr, for a total (on 32 bit platforms) of 12 bytes in the object. With the GC_header attached, the object consumes 16 bytes. If we were to add an additional 32 bit member field, we would need to update the runtime (e.g. thread.h) but if we stop there, the assertion in thread.c's sizeofThread would fail. This is because the compiler also needs to be taught that this object has changed size, and potentially layout. If we look in rep-type.fun (the GC_objectType comment in object.h leads us to rep-type.fun) we find that the thread object ("val thread = fn ()...") calculates the expected size of the object, and it is not derived directly from the runtime C structure, but is instead a hard coded calculation. Modifying this calculation ("val bytesObject =..") to include the hypothetical field we added, and also adding the type of the field, in the correct position, to the vector in the body of the function, will result in the assertion now passing.

3.4 Stacks

MLton uses green threads. These are managed entirely in user space by your application. Only one thread runs at a time and each thread has its own stack. The global state structure (`GC_state`) has fields that point to the stack of the currently running thread.

At startup (`init-world.c`) MLton creates a new thread and then switches to it. The act of switching copies a pointer to the thread's stacks into the `GC_state` structure. MLton's main loop (trampoline) operates entirely out of that structure, so this effectively runs the thread.

When another new thread is created (`new-object.c`), MLton allocates a new thread object (see `thread.h`) which has pointers to the newly allocated thread stack and exception stack. The running thread can then (optionally) "switch" to that thread which, as mentioned above, copies the new threads stack pointers into `GC_state` and then resumes the trampoline loop.

A thread stack is itself a block of heap allocate memory, the beginning of which corresponds to the `GC_stack` struct format. This struct tracks a few items related to garbage collection and also the total and used stack size. So the stack block itself might be 100 bytes, the struct only overlays the first 20-32 bytes (depending upon architecture) with the rest of the block being ostensibly unstructured.

However, the rest of the block is actually a sequence of stack frames. These frames are pushed onto the stack as needed based on what function is being called. When they are pushed, the stacktop is extended by a size corresponding to the frame being pushed. When the function returns, the stacktop is moved back down by the appropriate size, thereby reclaiming space on the stack for future function calls.

Since the compiler is able to derive what data will be stored in each frame, a frame will have a predetermined size and format. The start of each frame consists of a label indicating the type of frame, a pointer to an array of frame offsets, and the size of the frame. The offsets (relative to the base of the frame) are used to determine where in the frame each object pointer lives.

Since you can not determine exactly how many function calls may occur (or in what order) at compile time, the stack grows when necessary. This is handled by the garbage collector by making a copy of the current stack into a new, larger, memory area, and then updating the stack pointers in the thread structure.

3.4.1 Frames

A stack consists of a header and some number of frames. Each frame corresponds to a function in your program. Since frames consist of a known set of values (e.g. function local variables, calling function, return value) the frame size and layout is pre-calculated and stored in a frameLayout structure. That structure is stored in an array indexed by the function number (e.g. the "case" in the Chunk's switch statement). When you call a new function, the frame layout is extracted from the array and mapped to the next available chunk of stack memory and a Push() is performed to move the stack bottom forward. If no more stack memory is available, the GC will grow the stack to accommodate the new frame. When the function completes, the return value is placed into the frame, and a negative Push() is performed to move the stack bottom back down. The calling function number is loaded into the next function to execute and the trampoline loop is resumed at its top to allow the switch statement to move to the calling function.

How do we know, given a section of stack memory, which frame it contains? There is a list of GC_frameLayout structures (see test.0.c) that are indexed by function number. Since we know that the return address is at the top of every frame, we can locate it and use it as an index into the frameLayout structure.

3.4.2 New Thread

When a new thread is created, a new stack is also created for that thread. A pointer to the new stack is stored in the thread's GC_thread structure. When the application switches to that thread, it will call GC_switchToThread which then sets off a sequence of actions and functions in the C runtime.

GC_switchToThread (switch-thread.c) will save the current thread's stack usage and exception pointer. It will then call switchToThread (switch-thread.c).

switchToThread (switch-thread.c) will set the currentThread field in GC_state to the newly switched to thread and will then call setGCStateCurrentThreadAndStack (gc_state.c)

setGCStateCurrentThreadAndStack (gc_state.c) operates using the currentThread field to determine the stack details for the current thread (that we just switched to), i.e., bottom, top, limit, exception pointer. It sets corresponding fields in GC_state to those values to complete the thread switch. After some GC house keeping, the runtime is able to return back to the SML program. Since there is no 'program counter' as in assembly, the GC_switchToThread call is always followed by the Return macro. That macro will load a function number to execute and then jumps to the 'top' of the chunk's switch statement to then execute that function. But if the new stack is empty, where does that function number come from? What actually happens in the SML code is that newThread is called from copyThread and never called directly, so the new thread (that is being switched to) gets a copy (memcpy) of the old thread's stack. The frame at the top of that stack will contain the function pointer to the function the new thread is going to start at.

So switching to a new thread in SML follows the following steps:

curthr	SML	S(, X) = fn	setup function number at stackTop of curthr stack
curthr	SML	Push(X+wordsize)	push stackTop forward one frame
curthr	RT	GC_copyCurrentThread()	setup copy, enter/leave, etc
curthr	RT	copyThread()	make the copy of curthr, incl. its entire stack.
curthr	RT	newThread()	allocate new thread struct
curthr	RT	copyStack()	perform the stack copy using memcpy
curthr	SML	S(, X) = fn	setup function number that curthr will next execute on its stack, note that this does not change the function number that the new thread will execute because the copy, made above, is not altered
curthr	SML	Push(X+wordsize)	push stackTop forward one frame
curthr	SML	GC_switchToThread	switch to the new thread we created above. This will adjust the various GC_state pointers so that they, e.g., point to the new thread's stack bottom/top/etc.
newthr	SML	Return()	load the next function number from the top of the stack, jump to the top of the Chunk's switch statement.

3.5 Understanding the GCState

You will notice that some of the offsets your program uses are defined at the top of, eg, test.1.c

How are these calculated?

mlton/main/compile.fun search for "Set GC_state offsets and sizes"

The above function does a look up in a text file (build/lib/targets/self/constants) which is generated by a C program emitted by the "build" function in mlton/main/lookup-constant.fun

Chapter 4

MLton

This chapter describes the compiler proper, which is found in the `mlton` directory.

4.1 Sources

`ast`

Abstract syntax trees produced by the front end.

`atoms`

Common atomic pieces of syntax trees used throughout the compiler, like constants, primitives, variables, and types.

`backend`

The backend translates from the `Cps` IL to a machine independent IL called `Machine`. It decides data representations, stack frame layouts, and creates runtime system information like limit checks and bitmasks.

`call-main.sml`

A one-line file that is the last line of the compiler sources. It calls the main function.

`closure-convert`

The closure converter, which converts from `Sxml`, the higher-order simply-typed IL, to `Cps`, the first-order simply-typed IL.

`cm`

Support for SML/NJ-style compilation manager (CM) files.

`codegen`

Both the C and the native X86 code generator.

`control`

Compiler switches used throughout the rest of the compiler.

`core-ml`
: The implicitly typed IL that results from defunctorization. Contains a pass of dead code elimination for eliminating basis library code. Also contains the pass that replaces constants defined by `_prim` with their values.

`elaborate`
: The elaborator, which matches variable uses with bindings in the AST IL and defunctorizes to produce a `CoreML` program. It does not do type checking yet, but will someday.

`front-end`
: The lexer and parser, which turn files into ASTs.

`main`
: The two main structures in the compiler, one (`Main`) for handling all the command line switches and one (`Compile`) which is a high-level view of the the compiler passes, from front end to code generation.

`Makefile`
: To make the compiler.

`mlton.cm`
: An automatically generated file (`make mlton.cm`) that lists all of the files (in order) that make up the compiler.

`mlton.sml`
: An automatically generated file (`make mlton.sml`) that contains all of the compiler sources concatenated together.

`rcps`
: An experimental IL, similar to CPS, but with more expressive types for describing representations (hence the “r”). Not yet in use.

`sources.cm`
: For compiling with SML/NJ.

`ssa`
: Static-Single-Assignment form, the first-order simply-typed IL on which most optimization is performed. There are roughly 20 different optimization passes (some of which run several times).

`type-inference`
: The type inference pass, which translates from `CoreML` to `Xml`.

`xml`
: The `Xml` and `Sxml` intermediate languages. Also, the passes that monomorphise, do polvariance, and implement exceptions.

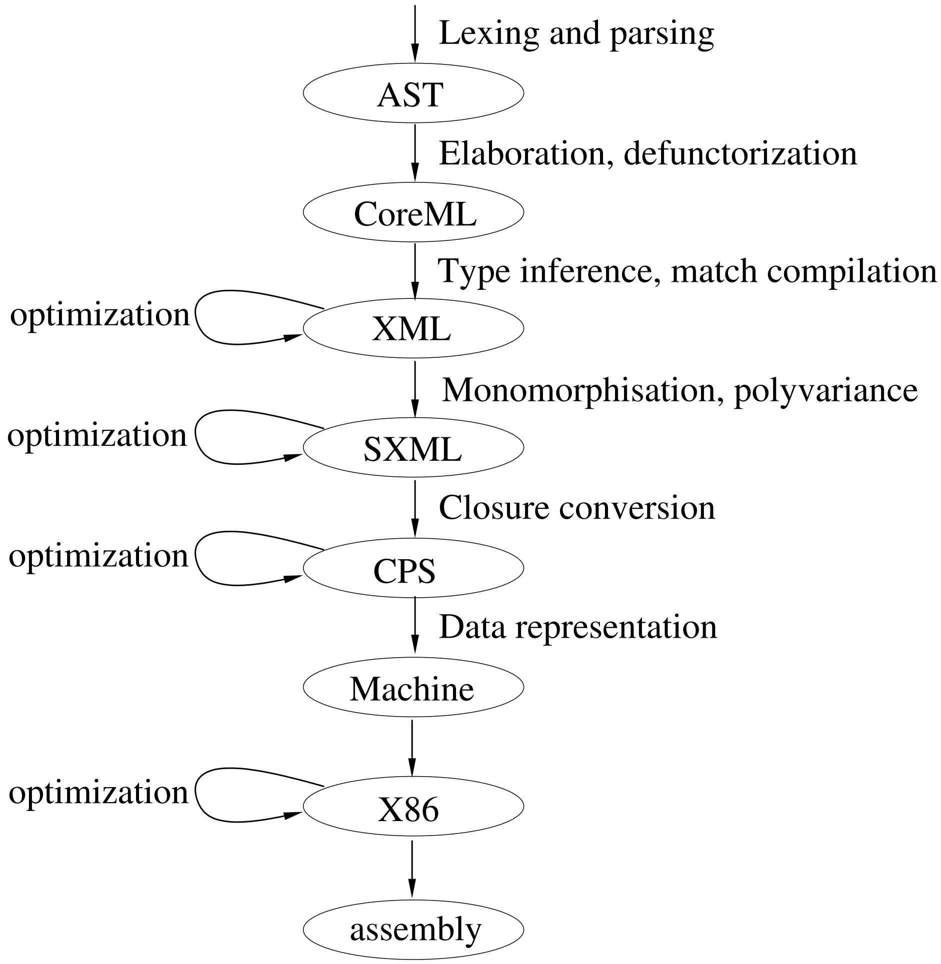

Figure 4.1: Compiler structure

4.2 Compiler Overview

Figure 4.1 shows the overall structure of the compiler. Intermediate languages (ILs) are shown in ovals. The names of compiler passes adorn arrows between ILs. In this section I give a brief description of each pass and a pointer to a later section that covers the pass in detail. Each IL also has a separate section devoted to it.

The front end (Chapter **??**) takes SML source code (a complete program) and performs lexing and parsing, producing an abstract syntax tree (Chapter **??**). The lexer is produced by ml-lex[**?**] and the parser is produced by ml-yacc[**?**]. The specifications for the lexer and parser were originally taken from SML/NJ109.32. The lexer is unchanged. I have substantially modified the actions in the grammar to produce my own version of abstract syntax trees (similar to, but different from SML/NJ).

Defunctorization (Chapter **??**), translates abstract syntax trees to a small implicitly typed core language, called Core ML (Chapter **??**). Its primary task is to eliminate all uses of the module system (signatures, structures, functors). It does this by applying all functors and flattening all structures, moving declarations to the top level. This phase also performs precedence parsing of infix expressions and patterns (the code to do this was taken from SML/NJ). Finally, it does some amount of "macro expansion", so that the core language is smaller.

Type inference (Chapter **??**) translates implicitly typed Core ML to an explicitly typed core language, XML (Chapter **??**), with explicit type abstraction and application. XML is based on the language "Core-XML" described in [Har]. Type inference consists of two passes. The first pass determines the binding sites of type variables that are not explicitly bound (section 4.6 of the Definition). The second pass is a pretty standard unification based Hindley-Milner type inference[**?**]. The type inference pass also performs overloading resolution and resolution of flexible record patterns. This pass also performs match compilation, by which I mean the translation of case statements with nested patterns to (nested) case statements with flat patterns.

Monomorphisation (Chapter **??**) translates XML to its simply-typed subset, called SXML (Chapter **??**), by duplicating all polymorphic functions and datatypes for each type at which they are instantiated. Monomorphisation is only possible because SML has "let-style" polymorphism, in which all uses of a polymorphic value are syntactically apparent (after functors are eliminated).

Chapter 5

Notes

This chapter contains random notes (usually old emails) on various subtle issues.

5.1 IntInf and Flattener

```
From: "Stephen T. Weeks" <sweeks@intertrust.com>
Date: Tue, 27 Jun 2000 18:52:19 -0700 (PDT)
To: MLton@research.nj.nec.com
Subject: safe for space ... and IntInf
```

Your mail also came at a fortunate time, as I was trying to track down a seg fault I was getting in the smith-normal-form regression test. For stress testing, I turned off all the cps simplify passes (except for poly equal) and ran the regressions. smith-normal-form failed with a seg fault when compiled normally, and failed with an assertion failure in `IntInf_do_neg` when compiled -g. The assertion failure was right at the beginning, checking that the frontier is in the expected place.

```
assert(frontier == (pointer)&bp->limbs[bp->card - 1]);
```

I'd been tracking this bug for a couple hours when I received your mail about the flattener. Do you see the connection? :-) As a reminder, here is the code for `bigNegate`

```
fun bigNegate (arg: bigInt): bigInt =
       if Prim.isSmall arg
             then let val argw = Prim.toWord arg
                  in if argw = badw
                        then negBad
                        else Prim.fromWord (Word.- (0w2, argw))
                  end
             else Prim.~ (arg, allocate (1 + bigSize arg))
```

The problem is, when the flattener is turned off, there is an allocation in between the call to allocate and the `Prim.~` call. The argument tuple allocation screws everything up. So, we are relying on the flattener for correctness of the IntInf implementation. Any ideas on how to improve the implementation to remove this reliance, or at least put an assert somewhere to avoid falling prey to this bug again?

Chapter 6

Todo

native backend vs x86 backend To unpackage debian, do

```
dpkg -x ../mlton_20010806-2_i386.deb .
```

Bibliography

[Har]

[Lei13] Brian Andrew Leibig. Masters project report: An LLVM back-end for MLton. `https://www.cs.rit.edu/\~mtf/student-resources/20124\_leibig\_msproject.pdf`, 2013.