# Advanced Project
# Advanced Compiler Construction

Martin Ekerå
martin.ekera@epfl.ch
076-200 11 83

Magnus Sköld
magnus.skold@epfl.ch
078-748 56 49

June 26, 2006

## 1 Copying GC

In order to improve the speed of memory allocation and garbage collection, we implemented a precise copying GC to replace the mark and sweep GC. In our implementation, we have used Cheney's algorithm which is non-recursive.

### 1.1 Pointer tagging

Since the copying GC is precise, it needs to be able to distingush between pointers and non-pointers. To accomplish this, it is necessary to modify both the VM and the compiler.

Since pointers are always multiples of 4, and thus never odd, we implement a simple tagging scheme in the compiler such that the integer literal $n$ is written as $2^n + 1 = T(n)$, ensuring that its representation is always odd and we use the last bit to distinguish between pointers and integer literals.

$$T(n) = (n << 1) + 1 \quad T^{-1}(n) = (n >> 1)$$

Furthermore, in the VM, the tagging scheme must be respected whenever aritmetric operations or input/output operations are performed; thus, almost all primitve operations need be modified and the VM will no longer be compatible with previous code output by older versions of the compiler.

For some of these operations, it is simple to maintain the tagging scheme, e.g. for the addition operations which becomes $x + y \to x + y - 1$. For more advanced operations, such as the modulus operation, we will have to shift $n$ to the right, perform the operation and then re-tag the result value, which incurs a non-negligible overhead on the VM.

### 1.2 Memory initialization

The heap was divided into two spaces; `FROM` space and `TO` space. We adopted the convention that `FROM` space is the half of memory to which the `next_free` pointer points, thus eliminating the need of introducing more variables.

### 1.3 Memory allocation

If the `next_free` pointer points to the first half of the heap, then we allocate blocks in the heap until we reach the middle. Otherwise, we allocate blocks until we reach the end of the heap. To allocate a block of size $n > 0$ bytes, we allocate L(n) bytes, where

$$L(n) = \begin{cases} 4 + n & \text{if } n \bmod 4 = 0 \\ 8 + n - (n \bmod 4) & \text{otherwise} \end{cases}$$

and use the first 4 bytes to store the length of the block in tagged format, i.e. if the length in bytes is $l$ we store $T(l)$ in the first block.

If the current layout of the `FROM` space does not permit an allocation of $L(n)$ bytes, garbage collection is initialized, see section 1.4, and the allocation reattempted. Should the allocation still fail, an error message is printed and the VM is terminated.

### 1.4 Garbage collection

Using the method described earlier, see 1.2, we first determine which part of the heap is `FROM` space and which part is `TO` space. We initialize the `nf` and `scan` pointers to point to the first block in `TO` space.

Next, we iterate over all registers except (`R0` which is identically 0, `R28` also known as the LK register and R31 which contains the code pointer). If a register does not contain a pointer, it is ignored.

Otherwise, we subtract the pointer by 4 and follow the pointer to the length field of the allocated block. Next, we check if the value in the length field is even or odd.

If it is even, then the block has already been copied and the length field contains its new address in TO space. We then update the value of the register and proceed with the next register.

Otherwise, we move the block to the position denoted by nf in TO, the value of the length field of the block in TO space is set to nf, and then nf is advanced by the size of the block. We update the value of the register to hold the address to the block in TO space.

After having examined a register, we let the scan pointer sweep from its current position up to nf. For each 4 byte block examined that is a pointer we repeat the process described previously, i.e. if the block referred to has not already been copied, it is copied. Then, we update the pointer examined so that it points to the block's new address in TO space.

Finally, when all register have been processed, we let next_free = nf and return.

## 1.5 Problems

In general, we experienced few problems with the implementation of the algorithm, but the potential of a small errors raging havoc is quite large, considering that even a small error that manages to sneak into the code has the potential of generating completely unexpected behaviour.

One problem is that if the user performs pointer arithmetrics (which is possible using the addition or subtraction, for example) so that the pointer no longer points to the beginning of a block but still remains inside it, then the GC could cause unexpected behaviour and/or crash the program.

Our implementation of the mark and sweep garbage collector does support such pointers - at least to a limited extent - but we have decided not to support this kind of pointers for the copying GC due to extra overhead that such support would introduce.

## 2 JIT compilation

In order to make the JIT compiler compatible with the copying GC, we decided to let it operate on assembler code with tagged integer literals. The JIT compilation was implemented by leveraging functionality from the GNU Lightning library.

To activate JIT compilation, the flag -b must be specified on the command line when executing minivm. This flag is incompatible with the flags -t indicating threaded mode, for obvious reasons, and -i since we do not support disassembling in JIT mode.

When JIT compilation has been activated, initialization of the VM proceeds as normal, with the exception of label conversion, see 2.1 and entry into the main execution loop, see section 2.2.

### 2.1 Labels

To support CMOV-style jump instructions, we introduced two new opcodes: LLAB for loading labels and ANCH for anchoring labels. Next, we modified alignloader.c and parser.c amongst other files to load the miniscm assembler into memory whilst effecuating the following transformation on-the-fly

```
   LINT R1 label           LLAB R1 label
   LINT R2 20              LINT R2 20
   CMOV R31 R1 R0          CMOV R31 R1 R0
   HALT               →    HALT
label:                     ANCH label
   PINT R2                 LINT R1 label
   CMOV R31 R2 R0          CMOV R31 R2 R0
```

after which the label label is translated to an ASM code PC, in this case 24.

### 2.2 Initialization

Instead of calling exec() or t_exec() when the initialization of the VM is complete, the function init_JIT() is called. It initializes a code buffer of size sizeof(jit_insn) * code_size bytes that will hold the JIT-compiled representation of the program. Futhermore, it defines a new function prog() that will represent the entire program, once JIT compilation is complete.

The user of the VM must define a large enough code size using command line arguments. It is quite difficult to calculate how large a code size will be required; a single ASM instruction often require a number of JIT instructions to be emitted. A conservative approximation could be made, or one could set out with a small code size and expand it progressively; however, we have not implemented such functionality.

Instead, before processing an instruction, we verify that there is enough room for at least 25 primitive function to be output. According to the GNU Lightning manual, 4096 bytes sufficient for between 100 and 400 instructions. Therefore we require that 1024 bytes of space is available before processing the next instruction. If there is less space available, we print an error message and exit the VM gracefully.

## 2.3 Conversion ASM $\longrightarrow$ JIT

Next, we let `R[31]` iterate over the entire program, translating each opcode into corresponding GNU lightning JIT instructions. Almost all functions need deal with integer literals, which are tagged, and thus must maintain the tagging scheme. This introduces some overhead in the program, as many shift instructions are emitted to handle tagging.

The code conversion is straightforward, with the exception of the `LLAB`, `ANCH` and `CMOV` instructions, and instructions which need call other methods such as `ALOC` or `PINT`.

## 2.4 The `LLAB` instruction

When an LLAB instruction is read, we check if the label referred to has been used previously by looking up the label PC in a special hashtable. If it was not previously used, we add a record in the hashtable for the label, storing its ASM PC in the hashtable.

Next, we emit a `MOVI` JIT instruction, moving the current code PC as given by `jit_forward()` into register `R1`, and we store a pointer to this JIT instruction in the label's record in the hashtable.

## 2.5 The `ANCH` instruction

When an LLAB instruction is read, we check if the label used has been used previously by looking up the label PC in a special hashtable. If it was not previously used, we add a record in the hashtable for the label, storing its PC in the JIT code, as given by `(_jit.x.pc)`.

## 2.6 Label patching

Once the entire program has been processed, we iterate over all labels contained in the label hashtable. For each label, we iterate over the list of `MOVI` instructions having it as target. All of these JIT `MOVI` instructions are then patched, making them move the JIT code PC to which to jump into the register instead of the their own PC (as they were originally setup to do).

## 2.7 The `CMOV` instruction

Moving a value from some register into register `R31` is the standard method of performing jumps in the ASM code. We need to handle this case separately, and translate it into a `jit_jmpr()` instruction. Given the instruction

```
CMOV R[a] R[b] R[c]
```

then if `R[c] = 0` and if $\texttt{a} = 31$, we simply jump to the value of `R[a]`. This register will contain the JIT PC of the jump's target instruction (grâce à label patching, see 2.6). The translation to JIT code when $\texttt{a} \neq 31$ is obvious.

N.B. We have decided not to update `R31` when a jump operation takes place, since the `R31` register no longer corresponds to the ASM PC. We could update it after the evaluation, so that it always corresponds to the JIT PC, but this would incurr performance penalties. Instead, `R31` is a register like any other, execept that one can not perform a "traditional" `CMOV` operation with `R31` as target.

This restriction should not pose any problems, as `R31` is only used for `CMOV`-style jumps at present.

## 2.8 `DIV`, `MOD` and division by zero

The instructions `DIV` and `MOD` test if the denominator is zero and if so prints an error message and exits the VM gracefully.

## 2.9 Instructions that call functions

Instructions such as `ALOC`, `PINT`, etc. which call other function are implemented using the jit_prepare(), jit_pusharg() and jit_finish() instructions provided by GNU Lightning.

## 2.10 Executing the JIT code

Once the entire program has been JIT compiled, we perform label patching as described in 2.6, flush the code buffer, call prog() and cross our fingers.

## 2.11 Problems

The implementation of the JIT compiler resulted in a considerable increase in our caffeine consumption, as minute code errors were given gigantesque proportions and often ended up crashing the VM. Furthermore, the GNU Lightning library is slightly unstable, and its documentation could well do with some revisions. Nevertheless, we had fun writing the JIT and taking on the challenges that it laid out for us.

## 2.12 Possible Improvements

We currently exclusively use registers in the R vector. Performance could be improved by moving a few frequently used registers such as R0 and R28 into JIT registers. We have not had the time to implement this feature, although it should be rather easy.

Currently, we do not support logging in the VM, but this could be done easily; if the log level is set to debug, we would simply call the write_log() function with the appropriate arguments after the execution of each instruction.

Furthermore, the disassembler does not function in conjunction with the JIT compiler. It is possible to write a disassembler "hook" for the JIT compiler, but we have not had time to support disassembling.

The instructions LOAD and STOR do not check if the address is a code pointer or a heap pointer. We have not had the time to implement this feature, although it should be rather easy. It should be pointed out however, that such a check would incur a penalty on the VM's performance.

As was mentioned above, register R31 is not updated after the evaluation of each instruction to reflect the ASM or JIT PCs. We have decided against this because of the overhead it incurs, see section 2.7.

Another strange problem that we experienced was that the VM crashed in JIT mode every time it called an external function that printed a character string followed by a single line break character "\n" (ASCII code 10).

We tried a number of different approaches in our attempts to correct the error. It is possible that the flush operation that is performed when "\n" is written to the output stream is responsible for the VM crashing, but if so we did not manage to find out exactly why.

Finally, we resorted to printing "\n \b", i.e. we terminated the string with a space and a backspace character, which is ugly but the VM behaves reliably.

## 2.13 PPC G4

We developed the JIT compiler on a PowerBook G4 and have not had time to test the compiler under Intel or SPARC processors.