Minischeme Project

Michel Schinz
Advanced compiler construction, 2008-02-22

The project

What you get:

1. an interpreter and a compiler for minischeme, written in Scala,
2. a virtual machine, written in C.

What you have to do:

1. two non-graded “warm-up” exercises,
2. add a garbage collector to the virtual machine,
3. add support for closures to the compiler,
4. optimise tail calls in the compiler,
5. an advanced project of your choice.

The minischeme language

The minischeme language

Minischeme is a dialect of Scheme, itself a dialect of Lisp. Its main characteristics are:

- it is “dynamically typed”,
- it has few side effects (exceptions: arrays, input/output),
- it is functional: functions are first-class values,
- it is very simple, with only four keywords (`define`, `let`, `lambda` and `if`),
- memory is freed automatically.

Syntax

`(define` *name* *expr*`)`

Global value definition, binding the value of *expr* to the *name*. Only valid at the top level.

Global values are visible in the whole program, but are initialised in the order in which they are written.

`(let ((`$name_1$ $expr_1$`) …)` $body_1$ … $body_m$`)`

Local value(s) definition: $name_1$ is bound to the value of $expr_1$, $name_2$ to the value of $expr_2$, etc. while $body_1$ … is evaluated. The value of the whole expression is the value of $body_m$.

Note: the names $name_{1 \ldots n}$ are only visible in $body_{1 \ldots m}$, not in $expr_{1 \ldots n}$

Syntax

`(lambda (`$name_1$ ...`)` $body_1$ ...`)`

Anonymous function, with parameters $name_1$ ... $name_n$ and body $body_1$ ... $body_m$.

`(if` $expr_{cond}$ $expr_{then}$ $expr_{else}$`)`

Conditional: evaluate $expr_{else}$ iff $expr_{cond}$ evaluates to 0, otherwise evaluate $expr_{then}$.

`(`$expr_{fun}$ $expr_1$ ...`)`

Function application: call $expr_{fun}$ with $expr_1$ … $expr_n$ as arguments.

`1 2 3` ...

Integer constants.

Code Example

Function to compute x^y on integers (*y* must be positive):

```
(define pow
  (lambda (x y)
    (if (= 0 y)
        1
        (if (= 0 (% y 2))
            (let ((z (pow x (/ y 2))))
              (* z z))
            (* x (pow x (- y 1)))))))
```

Let as syntactic sugar

Notice that `let` can be considered as syntactic sugar, as it is completely equivalent to the immediate application of an anonymous function:

```
(let ((name1 expr1)
      (name2 expr2)
      ...)
  body1
  body2
  ...)
```

⇔

```
((lambda (name1 name2 ...)
   body1
   body2
   ...)
 expr1
 expr2
 ...)
```

Example:

```
(let ((x 40)
      (y 2))
  (+ x y))
```

⇔

```
((lambda (x y)
   (+ x y))
 40
 2)
```

Primitives

Minischeme is equipped with the following primitives, most of which correspond directly to one VM instruction:

- Arithmetic primitives: `+`, `-`, `*`, `/`, `%`
- Logical primitives: `<`, `<=`, `=`
- Vector primitives: `vector`, `vector-ref`, `vector-set!`
- Input/ouput primitives: `read-char`, `print-char`

Primitives are invoked using the syntax of function application, for example: `(* 6 (+ 4 3))`

However, *primitives are not functions*. In particular, primitives cannot be manipulated as values, while functions can.

Eta-exapansion

Since primitives cannot be manipulated as values, the following definition should in principle not be accepted:

```
(define plus +)
```

However, the minischeme compiler performs a transformation known as **eta-expansion** to transform the above code into the following, legal one:

```
(define plus (lambda (x1 x2) (+ x1 x2)))
```

In summary, the aim of eta-expansion is that whenever the programmer tries to use a primitive as a value, that primitive is replaced by an equivalent anonymous function. This guarantees that primitives are never used as values.

Vectors

Minischeme provides three primitives to work with vectors (a.k.a. arrays):

- `(vector` e_1 … e_n`)` creates a vector of *n* elements, initialised with the values of e_1 … e_n.
- `(vector-ref` *v* *n*`)` returns the n^{th} element of *v*. Indexing is 0-based.
- `(vector-set!` *v* *n* *e*`)` sets the n^{th} element of *v* to the value of *e*.

Notice that `vector` accepts a variable number of expressions. Since minischeme does not provide the concept of functions with a variable number of parameters, it is the only primitive that cannot be eta-expanded.

Representing pairs

Pairs can easily be represented using vectors:

```
;; construct a pair
(define cons
  (lambda (f s)
    (vector f s)))
;; get first component
(define car (lambda (p) (vector-ref p 0)))
;; get second component
(define cdr (lambda (p) (vector-ref p 1)))
```

Note: the names `cons`, `car` and `cdr` are historical.

Representing lists

Lists can easily be represented using pairs: the first component of the pair contains the head of the list, while the second component contains its tail – another list. The empty list is represented by a special value called `nil`.

This representation of lists by pairs is used in most functional languages: Scheme, Haskell, OCaml, Scala, etc.

For example, the list 1,2,3,4 can be constructed by the following code:

```
(cons 1 (cons 2 (cons 3 (cons 4 nil))))
```

and its second element can be accessed by the following code, where *lst* represents the list:

```
(car (cdr lst))
```

The minivm virtual machine

The minivm virtual machine

Minivm is a virtual machine designed for this project. Its main characteristics are:

- it is register-based: there are 32 general-purpose registers $R_0 \ldots R_{31}$, and a program counter,
- it is very simple, with only 16 instructions,
- it accepts textual assembly code as input.

The design goals were:

- to have a simple, easy to implement machine,
- to have it resemble a real processor, to make the compiler realistic.

However, this machine is definitely not an ideal target for a Scheme compiler!

Instruction set

Minivm instruction set can be categorised as follows:

- Arithmetic: `ADD`, `SUB`, `MUL`, `DIV`, `MOD`
- Control: `ISLT`, `ISLE`, `ISEQ`, `JMPZ`, `HALT`
- Memory: `ALOC`, `LOAD`, `STOR`, `LINT`
- Input/output: `RCHR`, `PCHR`

Arithmetic instructions

`ADD` R_a R_b R_c	$R_a \leftarrow R_b + R_c$
`SUB` R_a R_b R_c	$R_a \leftarrow R_b - R_c$
`MUL` R_a R_b R_c	$R_a \leftarrow R_b * R_c$
`DIV` R_a R_b R_c	$R_a \leftarrow R_b / R_c$
`MOD` R_a R_b R_c	$R_a \leftarrow R_b \bmod R_c$

Control instructions

`ISLT` R_a R_b R_c	$R_a \leftarrow R_b < R_c$ [false: 0, true: 1]
`ISLE` R_a R_b R_c	$R_a \leftarrow R_b \leq R_c$ [false: 0, true: 1]
`ISEQ` R_a R_b R_c	$R_a \leftarrow R_b = R_c$ [false: 0, true: 1]
`JMPZ` R_a R_b	if $R_b = 0$ then `PC` $\leftarrow R_a$
`HALT`	halt virtual machine

Memory instructions

`LINT` R C	$R \leftarrow C$
`LOAD` R_a R_b R_c	$R_a \leftarrow \texttt{Mem}[R_b + w * R_c]$
`STOR` R_a R_b R_c	$\texttt{Mem}[R_b + w * R_c] \leftarrow R_a$
`ALOC` R_a R_b	$R_a \leftarrow$ new block of R_b words

w is the word size in bytes of the host architecture: 4 on 32 bits architectures, 8 on 64 bits architectures.

I/O instructions

`RCHR` *R*	*R* ← read character from input
`PCHR` *R*	print `char`(*R*) on output

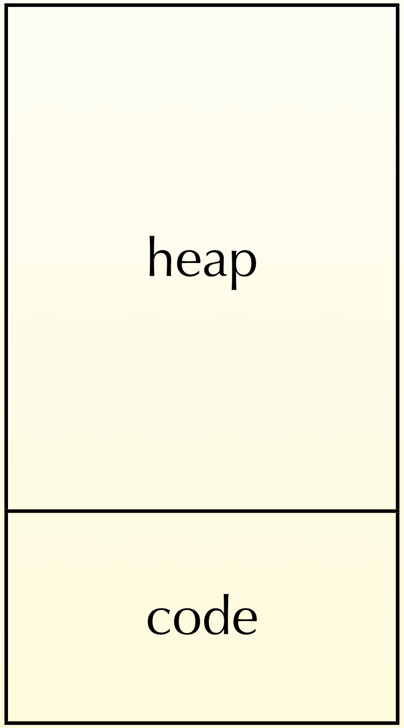
Memory model

The memory of minivm is split in two parts:

1. the bottom one contains the code,
2. the top one contains the heap.

Heap memory can be allocated using the `ALOC` instruction.

There is no instruction to free heap memory. Therefore, it is either never freed, or freed implicitly by a garbage collector or similar mechanism.



Implementation

You will be given a C implementation of minivm, with the following limitations:

- heap memory is never freed, and the VM exits when all available memory has been used,
- not as efficient as it could be.

Part of your job will be to improve it!

Implementation overview

The implementation is composed of the following three main modules (C files):

- **loader**: parses textual assembly files and calls functions in the engine module to emit the corresponding instructions,
- **engine**: produces the representation of the program in memory, based on instructions from the loader, and execute it later,
- **memory**: allocates memory used to store the program and the data used by it.

The minischeme interpreter and compiler

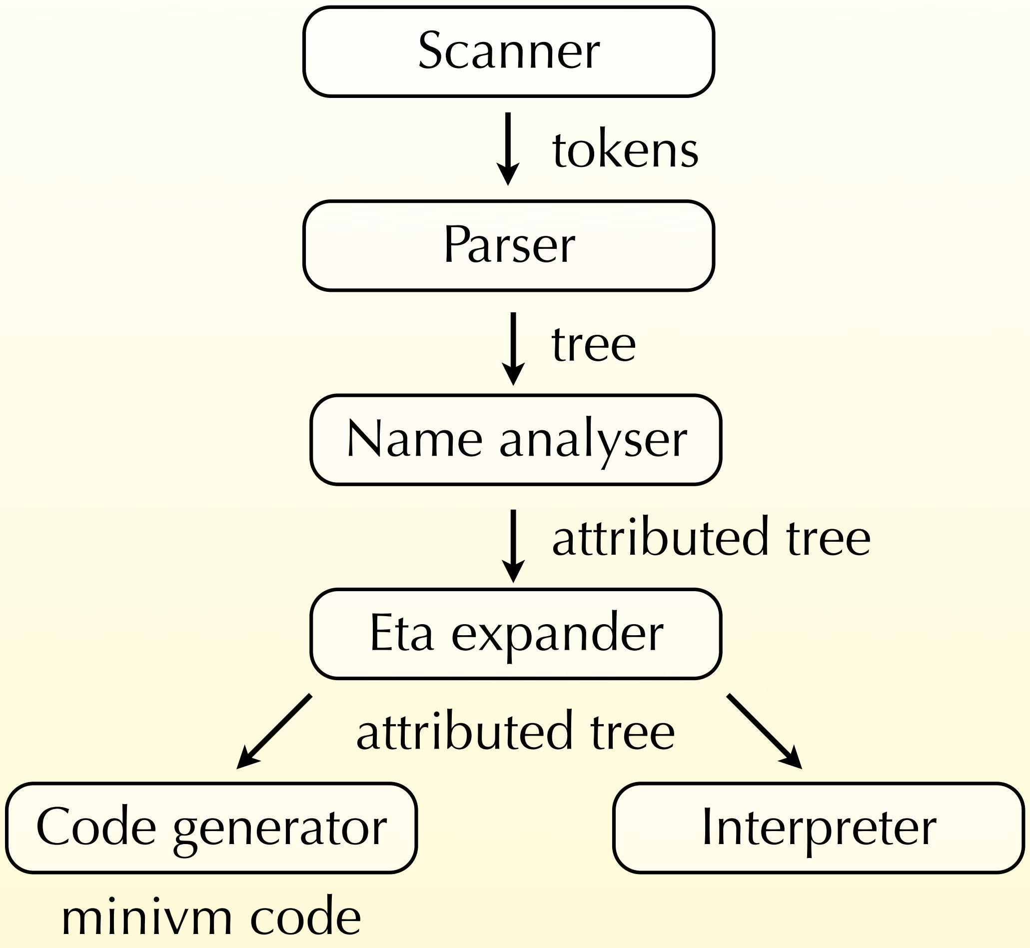
Interpreter and compiler

You will be given a Scala implementation of a minischeme interpreter and compiler. The interpreter implements the full language, but the compiler has the following limitations:

- anonymous functions cannot refer to values defined in an enclosing scope – unless they are global,
- no code is produced to perform dynamic checks, which means that most type errors or incorrect array indexing result in a VM crash (!),
- the produced code is not very good.

Your job will be to remove some of these limitations later.

Compiler organisation



Register usage

The compiler assigns specific roles to the following registers:

- `R0` – holds the constant 0,
- `R29` – holds the return address (`LK`),
- `R30` – points to the current stack frame (`FP`),
- `R31` – points to the global variables area (`GP`), containing all global values.

Notice that these conventions are in no way enforced by the VM itself!

Calling conventions

Function arguments are passed in registers $R_1 \ldots R_{28}$.

Functions with more than 28 arguments are not supported. They could be supported by passing some of the arguments on the stack, though.

The return value is put in R_1.

Registers R_0, R_{30} and R_{31} are callee-saved, $R_1 \ldots R_{29}$ are caller-saved.

Stack

Stack frames are allocated from the heap, and a pointer to the stack frame of the currently-executing function is stored in `R30` (a.k.a. the frame pointer `FP`).

The stack frame of a function *f* contains:

- the frame pointer of the function that called *f*,
- the return address,
- the arguments passed to *f*, which are saved on the stack at function entry,
- all the local variables of *f*.

Characters and strings

The minischeme compiler defines syntactic sugar for characters and strings.

A character constant is written `#\`*c* and is translated to the ASCII code of *c*. For example, `#\A` is translated to 65.

A string constant is written `"`*string*`"` and is translated to a vector. The first component of that vector contains the length of the string, while the next ones contain its characters encoded as above. For example, `"HELLO"` is translated to `(vector 5 72 69 76 76 79)`.

Code example

```
fact: LINT R2 else
      JMPZ R2 R1
      LINT R2 3
      ALOC R2 R2
      STOR R30 R2 R0
      LINT R3 1
      STOR R29 R2 R3
      LINT R3 2
      STOR R1 R2 R3
      ADD  R30 R2 R0
      LINT R2 1
      SUB  R1 R1 R2
      LINT R29 ret
      LINT R2 fact
      JMPZ R2 R0
ret:  LINT R3 2
      LOAD R2 R30 R3
      MUL  R1 R1 R2
      LINT R3 1
      LOAD R2 R30 R3
      LOAD R30 R30 R0
      JMPZ R2 R0
else: LINT R1 1
      JMPZ R29 R0
```

Code example

```
fact: LINT R2 else
      JMPZ R2 R1
      LINT R2 3
      ALOC R2 R2
      STOR R30 R2 R0
      LINT R3 1
      STOR R29 R2 R3
      LINT R3 2
      STOR R1 R2 R3
      ADD  R30 R2 R0
      LINT R2 1
      SUB  R1 R1 R2
      LINT R29 ret
      LINT R2 fact
      JMPZ R2 R0
ret:  LINT R3 2
      LOAD R2 R30 R3
      MUL  R1 R1 R2
      LINT R3 1
      LOAD R2 R30 R3
      LOAD R30 R30 R0
      JMPZ R2 R0
else: LINT R1 1
      JMPZ R29 R0
```

allocate,
initialise and
link frame

Code example

```
fact: LINT R2 else
      JMPZ R2 R1
```

allocate,
initialise and
link frame

```
      LINT R2 3
      ALOC R2 R2
      STOR R30 R2 R0
      LINT R3 1
      STOR R29 R2 R3
      LINT R3 2
      STOR R1 R2 R3
      ADD  R30 R2 R0
```

perform
recursive
call

```
      LINT R2 1
      SUB  R1 R1 R2
      LINT R29 ret
      LINT R2 fact
      JMPZ R2 R0
```

```
ret:  LINT R3 2
      LOAD R2 R30 R3
      MUL  R1 R1 R2
      LINT R3 1
      LOAD R2 R30 R3
      LOAD R30 R30 R0
      JMPZ R2 R0
else: LINT R1 1
      JMPZ R29 R0
```

Code example

allocate, initialise and link frame

perform recursive call

```
fact: LINT R2 else
      JMPZ R2 R1
      LINT R2 3
      ALOC R2 R2
      STOR R30 R2 R0
      LINT R3 1
      STOR R29 R2 R3
      LINT R3 2
      STOR R1 R2 R3
      ADD  R30 R2 R0
      LINT R2 1
      SUB  R1 R1 R2
      LINT R29 ret
      LINT R2 fact
      JMPZ R2 R0
```

compute result

```
ret:  LINT R3 2
      LOAD R2 R30 R3
      MUL  R1 R1 R2
      LINT R3 1
      LOAD R2 R30 R3
      LOAD R30 R30 R0
      JMPZ R2 R0
else: LINT R1 1
      JMPZ R29 R0
```

Code example

```
fact: LINT R2 else    ret:  LINT R3 2
      JMPZ R2 R1            LOAD R2 R30 R3
      LINT R2 3             MUL  R1 R1 R2
      ALOC R2 R2            LINT R3 1
      STOR R30 R2 R0        LOAD R2 R30 R3
      LINT R3 1             LOAD R30 R30 R0
      STOR R29 R2 R3        JMPZ R2 R0
      LINT R3 2       else: LINT R1 1
      STOR R1 R2 R3         JMPZ R29 R0
      ADD  R30 R2 R0
      LINT R2 1
      SUB  R1 R1 R2
      LINT R29 ret
      LINT R2 fact
      JMPZ R2 R0
```

allocate, initialise and link frame

perform recursive call

compute result

unlink frame and return