The Abstract Machine Mach

gla@postech

1 Introduction

The Mach abstract machine is a virtual machine written in ML, designed to execute code written in a simple assembly language. We will begin with a description of the memory layout of Mach, then describe the instructions available in Mach. Finally we will detail the ML interface functions for Mach. Please refer to `mach.mli` and `mach.ml` for more details.

2 Description of the Virtual Machine

Assembly code is a sequential list of instructions to be executed by a processor. The execution of an assembly program begins at a designated start label and proceeds linearly. Typical instructions include arithmetic operations, moves of values from one memory location to another, conditional tests of values, and jumps to new locations within the same program.

The machine can be in one of the following three states:

```
(* state indicates the machine state. in EXECUTION, the instruction at the
     current execution point is executed at the next cycle. in either ABNORMAL
     or NORAML, no more instructions are executed. in NORMAL, the machine
     keeps an execution result. *)
type state = EXECUTION | ABNORMAL | NORMAL of avalue
```

There are instructions shifting the machine state from `EXECUTION` to `ABNORMAL` or `NORMAL`, but the standard state for a running machine is `EXECUTION`.

2.1 Memory

Every value manipulated by instructions is stored in one of three different memory regions. These regions are all represented as collections of "memory cells" each of which can store a single integer, boolean value, unit value, string, or address of some other memory location.

The type `avalue` represents these atomic values.

```
type avalue =                (* atomic value *)
    AINT of int              (* integer constant *)
  | ABOOL of bool            (* boolean constant *)
  | AUNIT                    (* unit *)
  | ASTR of string           (* string constant *)
  | AADDR of addr            (* adderss constant *)
```

The aforementioned three regions of memory are:

- *registers*, an array of 32 memory cells used for temporary storage of values.
- *stack*, a linearly growing/shrinking structure of memory cells for storing temporary values or for pushing arguments for function calls. It has a finite size. (See `stackSize` in `mach.ml`.)
- *heap*, memory cells allocated in chunks using the MALLOC and FREE instructions (analogous to `malloc()` and `free()` in C).

Addresses of these three kinds of memory regions are tagged and are all distinct. The following section of ML code describes the type for representing memory addresses:

```
(* label is a code label specified by LABEL instruction. *)
type label = string       (* code label *)
(* addr is an address. it is a code label (CADDR), a heap handle (HADDR),
     or an index into the machine stack (SADDR). *)
type addr =                 (* address *)
    CADDR of label          (* code address *)
  | HADDR of int            (* heap address *)
  | SADDR of int            (* stack address *)
```

`CADDR`s are labels marking sections of the code, used in JUMP instructions. `SADDR`s and `HADDR`s are stack and heap addresses, represented as integers. Registers are accessed directly, without using addresses, as explained below.

2.1.1 Registers

There are 32 memory registers, numbered from 0 to 31. Each register can hold a single memory cell's worth of data. Several of these registers are special purpose registers and cannot store data.

- Register 0 (sp) holds a pointer to the empty location at the top of the stack. The PUSH and POP instructions will automatically update it.
- Register 1 (bp) holds a pointer to the start of the current stack frame. The CALL and RETURN instructions will automatically update it. Use of PUSH, POP, CALL, and RETURN will be described in Section 2.1.2.
- Register 31 (zr) always holds zero. Writing to it has no effect.
- Register pc, program counter, is a special-purpose register that holds the current execution point. It is not accessible to programmers.

All the other registers are equivalent, and can store any `avalue`. Some registers are given mnemonics, like those described above, and can be accessed either by number or by name. The code section below defines mnemonics for registers.

```
type reg = int              (* register *)
val numReg : int            (* number of registers *)
val sp : reg                (* = 0 *)
val bp : reg                (* = 1 *)
val cp : reg                (* = 2 *)
val ax : reg                (* = 3 *)
val bx : reg                (* = 4 *)
val tr : reg                (* = 30 *)
val zr : reg                (* = 31 *)
```

Section 2.2 describes how to use these registers.

2.1.2 Stack

Programming without procedure calls would be a tedious and silly exercise. The abstract machine includes direct support for function calls, through the CALL and RETURN instructions. A function call is represented as a jump to a new location in the code. The code to be executed has access to the same set of registers also visible to the caller, and thus may overwrite them. To resolve this issue, Mach provides a stack space to save data that should survive procedure calls.

Pushing and popping

The PUSH and POP operations are used to save and recover data to and from the top of the stack. Since the stack is just a contiguous section of memory, individual memory cells in the stack can also be accessed by address. For example, sp is pointing to the empty location at the top of the stack. Reading sp will produce an integer stack address `AADDR (SADDR` i`)` for an integer i. The stack grows upward, so the stack pointer *minus* some offset will be the address of a value located further down in the stack.

Calling and returning

CALL and RETURN are the instructions provided by Mach for achieving procedure calls/returns. When a CALL is executed, the address of the instruction after the CALL is pushed onto the stack. Then, the current value of bp (base pointer register) is pushed onto the stack, and execution jumps to the code address `CADDR label` given as an argument to the CALL instruction.

When a RETURN is executed, the stack is popped and the address from the stack (which is the address pushed by the most recent CALL) is saved in bp. Then the stack is popped again and the program counter is set to the value from the stack. The value is the location of the instruction stored after the most recent CALL. Execution then continues from that location.

Procedure calls are all well and good, but in order to make them useful, we must be able to pass an argument to a call and also retrieve a value from a call. Before calling a procedure, we may push a value as an argument onto the stack. Inside the callee, this argument can be accessed by subtracting 3 from the bp (base pointer). A procedure should use the register ax to store the value it wishes to return.

2.1.3 Heap

The heap is just a large collection of memory cells. It is usually used to store non-atomic values such as tuples, types, and closures. The MALLOC instruction takes a size s and returns an address to a chunk containing s memory cells. This storage is then marked as allocated, and can be freely modified by the code. When finished, the code should FREE the address returned by MALLOC, *i.e.*, execute the FREE instruction using the address returned by MALLOC as an argument.

2.2 The Instruction Set

Every instruction available on the Mach virtual machine, except LABEL and DEBUG, takes up a single memory cell in the code segment. Instructions are executed sequentially, starting from START_LABEL and ending when a HALT or EXCEPTION is reached.

2.2.1 Lvalues and Rvalues

Every Mach instruction has arguments which are either lvalues or rvalues. Consider the following C code:

```
y = x + 1;
```

`y` is used as the destination for the result of the computation. `x` and `1` are used as source values for the computation. Lvalues are destination values such as `y`, and rvalues are source values such as `x` and `1`. The type `lvalue` defines legitimate lvalues:

```
type lvalue =              (* lvalue *)
    LREG of reg            (* register *)
  | LREFADDR of addr * int  (* address with offset *)
    (* the same rule applies as in REFADDR in locating the destination. *)
  | LREFREG of reg * int    (* register with offset *)
    (* reg must hold an AADDR avalue. the same rule applies as in REFADDR
       in locating the destination. *)
```

As seen above, registers are legitimate destinations for computations. The others are slightly more complex. `LREFADDR` takes an address and an integer. An operation where `LREFADDR` (*addr*, *i*) is used as an lvalue uses as the destination the location with address `addr` plus the given offset i. `LREFREG` introduces another level of abstraction. Storing a value to an `LREFREG` will first examine the given register, which must contain an address. The given offset is then added to the address to obtain a new location, and the value is placed in the new location.

Rvalues, being the sources of values, are much more varied. The type `rvalue` defines legitimate rvalues:

```
type rvalue =                (* rvalue *)
    INT of int               (* integer constant *)
  | BOOL of bool             (* boolean constant *)
  | UNIT                     (* unit *)
  | STR of string            (* string constant *)
  | ADDR of addr             (* address *)
  | REG of reg               (* register *)
  | REFADDR of addr * int    (* dereferencing with address and offset *)
    (* addr cannot be a code address. if addr is a heap address, int is the
       offset within the heap chunk associated with the heap address, and
       it must be a non-negative interger less than the size of the heap
       chunk. if addr is a stack address, int is the offset from the stack
       address within the machine stack, and it can be a negative integer. *)
  | REFREG of reg * int      (* dereferencing with register and offset *)
    (* reg must hold an AADDR avalue. the same rule applies as in REFADDR in
       dereferencing. *)
```

`INT`, `BOOL`, `UNIT`, and `STR` are used for literals of the specified type. `ADDR` is used for an address literal of type `addr`. These will all be converted to their respective `avalue`s before they are used in the computation.

`REFADDR` will add the offset to the given address, and obtain the `avalue` stored in that memory cell. You should be aware of several restrictions when using `REFADDR`. `addr` cannot be a code address, since that would mean you were attempting to use part of the code as an `avalue`. If `addr` is a heap address, the offset is a non-negative integer giving a number of memory cells past the start of the heap chunk allocated by MALLOC. This offset must be less than the size of the heap chunk. If `addr` is a stack address, the offset can be positive or negative, and is added to the address to access the `avalue` in a memory cell of the stack.

`REG` looks into the given register to obtain the `avalue` contained within. `REFREG` looks into the register to obtain an address, adds the offset to it, and obtains the `avalue` at that location. The same restrictions as with `REFADDR` apply to offsets and addresses.

2.2.2 Instructions

We will use the notation val(r) to denote the `avalue` obtained from a given rvalue r, and loc(l) to denote the location obtained from a given lvalue l. These instructions are defined in the type `instr`.

- MOVE (l,r) moves val(r) to loc(l).
- ADD (l, r_1, r_2) moves 'val(r_1) + val(r_2)' to loc(l). Both val(r_1) and val(r_2) must be `AINT` `avalue`s.
- SUB (l, r_1, r_2) works like ADD, but with subtraction.
- MUL (l, r_1, r_2) is similarly defined.
- XOR (l, r_1, r_2) moves 'val(r_1) $\oplus$ val(r_2)' to loc(l). Both val(r_1) and val (r_2) must be `ABOOL` `avalue`s.

- NOT (l, r) moves 'not val(r)' to loc(l). val(r) must be an `ABOOL avalue`.
- PUSH (r) moves val(r) to loc(`LREFREG` (sp, `0`)) and increments sp by 1. In other words, it pushes val(r) onto the top of the stack.
- POP (l) moves val(`REFREG` (sp, `-1`)) to loc(l) and decrements sp by 1. It pops the stack to loc(l).
- MALLOC (l, r) allocates a new memory chunk of size val(r) in the heap, and moves its heap address to loc(l). val(r) must be an `AINT avalue`.
- FREE (r) frees the heap chunk associated with val(r). val(r) must be an `AADDR avalue` and it must also be a heap address.
- JUMP (r) jumps to the code address represented by val(r). val(r) must be an `AADDR avalue` and it must also be a code address.
- JMPNEQ (r_1, r_2, r_3) jumps to the code address represented by val(r_1) if val(r_2) is not equal to val(r_3). val(r_1) must be an `AADDR avalue`, and it must be a code address. Both val(r_2) and val(r_3) must be `AINT avalue`s.
- JMPNEQSTR (r_1, r_2, r_3) jumps to the code address represented by val(r_1) if val(r_2) is not equal to val(r_3). val(r_1) must be an `AADDR avalue`, and it must be a code address. Both val(r_2) and val(r_3) must be `ASTR avalue`s.
- JMPTRUE (r_1, r_2) jumps to the code address represented by val(r_1) if val(r_2) is a (`ABOOL true`) `avalue`. val(r_1) must be an `AADDR avalue`, and it must be a code address. val(r_2) must be an `ABOOL avalue`.
- CALL (r) pushes pc (program counter) onto the stack, pushes val(`REG` bp), and jumps to the code address represented by val(r). val(r) must an `AADDR avalue`, and it must be a code address.
- RETURN pops the stack onto loc(`LREG` bp), then pops the stack onto pc, then adjusts pc so that it points to the next instruction.
- HALT (r) changes the machine state to NORMAL with the execution result val(r).
- EXCEPTION changes the machine state to ABNORMAL.
- DEBUG is ignored.

3 Description of the Code

To execute a program on Mach, your translation function must produce a `Mach.code`, which is an abstract type representing a Mach program. The signature `Mach` provides several functions for generating the code of type `Mach.code`, and also several functions which will probably be useful in the translation.

As `Mach.code` is a representation of a sequential list of instructions, your translation functions will be dealing extensively with `instr lists`. The signature `Mach` provides helper functions for converting `instr lists` into `code`.

```
(* functions to create new code *)
(* code0 is empty code.
     clist il creates code from a list of instructions il.
     cpre il c creates code by prepending (clist il) to c.
     cpost c il creates code by appending (clist il) to c.
     (@@) c1 c2 creates code by concatenating c1 and c2. *)
val code0 : code
val clist : instr list -> code
val cpre : instr list -> code -> code
val cpost : code -> instr list -> code
val (@@) : code -> code -> code
```

The signature `Mach` also defines a `val START_LABEL : label`, which you will need to put into the final code to indicate the start point.

You will need to generate labels on the fly in order to designate different parts of the code as jump targets. The signature `Mach` provides functions for generating labels:

```
(* label is a code label specified by LABEL instruction. *)
type label = string       (* code label *)
...
(* functions to create new labels *)
(* labelNew () creates a new label.
   labelNewStr s creates a new label using s.
   labelNewLabel l s creates a new label using l and s. *)
val labelNew : unit -> label
val labelNewStr : string -> label
val labelNewLabel : label -> string -> label
```

These functions take strings in order to help you generate informative label names for debugging purposes.

Finally the signature `Mach` provides a function `val code2str : code -> string` for converting `code` to a string representation, perhaps for debugging purposes.