MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.001—Structure and Interpretation of Computer Programs
Fall Semester, 1997

Lecture Notes – Dec. 2, 1997

MACHINE ARCHITECTURE – Revised final sections

3. Implementing Pairs and Pointers

Scheme variables can have different types as their values, and we can't tell what they will be necessarily by inspecting the program. It depends what values are passed as arguments, and may vary between different calls.

We will encode Scheme values as 32 bit quantities, but their type will have to be encoded in them. Let us assume that we expect not to have more than 256 megabytes of memory (or 2^{28} bytes). Then we could use the four most significant bits of a 32 bit quantity to specify the type of a Scheme value, and the lower 28 bits to either be the value directly or to be the address in memory of the components that comprise the value[1].

A pair will need to have two 32 bit values as its components, the `car` and `cdr`. Thus a pair 32 bit value will have four bits of type tag, and 28 bits of address into memory where in two successive 4 byte locations will be found the `car` and `cdr`.

Strings will have an address which is a where a series of bytes making up the string are stored. Perhaps there will be a length field there also, or perhaps they will be null terminated.

Symbols will have an address where their components are stored—the only component we need consider is a symbol's print name, which itself will be a string (i.e., a 32 bit quantity with a string tag, pointing off to the string contents).

Numbers will have a four bit tag and then 28 bits of actual numerical value, so they will range from $-134,217,728$ to $134,217,727$.

The emptylist will have its own tag, and the other 28 bits can be zeroes.

Likewise booleans will have special tags, and the 28 bits of data will encode whether they are true or false.

Suppose we choose the following tags, as the most significant four bits for Scheme values:

[1] In a real implementation we can be much more clever than this on a byte addressed machine and encode the type bits in the lower two and half bits of a 32 bit quantity, take no performance hit in dreferencing pointers, and use some of the hardware word alignment testing to do type error checking for us—the Sparc architecture supports a handful of special instructions just so that Lisp can be implemented efficiently on it using these tricks.

```
symbol      0
string      1
pair        2
number      3      ;not a real address
emptylist   4      ;not a real address
boolean     5      ;not a real address
prim proc   6      ;address part is code
other proc  7      ;not needed in our mceval, but a good
                   ;idea for real
```

If symbols are additionally stored at low addresses then our current machine code will let us put their addresses as constants in the `cmp` instruction's available 18 bits. In a real implementation we might want to do something more general.

What we do need to do is provide ways of getting at the tags, and implementing `cons`, `car`, and `cdr`.

If there is a Scheme value in a register, then the tag is the four most significant bits. We could simply shift the contents 28 bits to the right to get ahold of them. Most machines have such an instruction, but to make our code below easier to read we will write it as:

```
gettag  dest, source
```

which will take a value in a `source` register, not disturb it, but place the four most significant bits in the low order four bits of the `dest` register, padded on the left with zeroes.

If we assume a pair is stored as two consecutive four byte quantities then indirect addressing, i.e., using the lower 28 bits of a register as an address to access in memory, possibly with a small offset, will implement `car`, and `cdr`. On a real machine we might need to zero out the four tag bits, but let us assume that:

```
load    dest, 0(reg)   ;works as car
load    dest, 4(reg)   ;works as cdr
```

but to make our code easier to read we will write it as:

```
getcar  dest, reg
getcdr  dest, reg
```

Finally we must deal with `cons`. That will not be a primitive instruction on any realistic machine, and must incorporate garbage collection. We will come back to this later in the lecture. For now let us assume that there is a procedure, with a beginning label of `cons` that expects arguments in registers `arg1`, and `arg2`, and which returns a new pair (tagged correctly) in the `val` register.

4. Writing the MC Evaluator in Machine Code

We are now in a position to translate our meta-circular evaulator, by hand into machine code.

We will give certain registers symbolic names and use those names in our code rather than `r0`, `r1`, etc. The registers we need, along with their usual contents are:

```
env     the current environment
exp     the expression currently being evaluated
unev    a list of things not yet evaluated
args    a list of evaluated arguments
proc    a procedure to apply
val     the place where all values are returned
tag     temporary register for putting tags
arg1    first argument to cons
arg2    second argument to cons
```

We will write each procedure as a label with the name of the procedure. We assume the following convention:

1. To call a procedure which we are implementing by hand simply do a `jsr` to its label.
2. Each procedure expects its arguments in certain registers.
3. Each procedure returns its result in the `val` register.
4. Each procedure returns to the place that was on top of the stack when it was called.
5. Each procedure is free to destroy all registers—it is the caller's responsibility to save any register it needs.

We will only implement a few of the procedures of the meta-circular evaluator; the others are all equally easy, or difficult, to work through. For the procedures we do implement we will use the following conventions for where the arguments are supplied:

```
mceval   exp env
mcapply  proc args
evlist   unev env
lookup   exp env
cons     arg1 arg2
```

Now we are in a position to implement the meta-circular evaluator. We can translate it piece by piece:

```
(define mc-eval
  (lambda (exp env)
    (cond ((number? exp) exp)                    ;base-case
          ((symbol? exp) (lookup exp env))       ;base case
          ((eq? (car exp) 'quote) (car (cdr exp)));special forms
          ((eq? (car exp) 'cond) (evcond (cdr exp) env))
          ((eq? (car exp) 'begin) (evseq (cdr exp) env))
          ((eq? (car exp) 'lambda) (list 'proc (cdr exp) env))
          ((eq? (car exp) 'define) (evdefine (cdr exp) env))
          (else (mc-apply (mc-eval (car exp) env)
                          (evlist (cdr exp) env))))))
```

```
mceval: gettag  tag, exp
        cmp     tag, #3
        jump.ne notnum
        load    val, exp      ;numbers self evaluate
        rts
notnum: cmp     tag, #0
        jump.eq lookup        ;tail recursive call to lookup
        getcar  val, exp      ;must be a combination
        cmp     val, #quote   ;relies on symbols having a 0 tag
        jump.ne notquot
        getcdr  val, exp
        getcar  val, exp
        rts                   ;return (cadr exp)
noquot: cmp     val, #cond
        jump.ne notcnd
        getcdr  exp, exp
        jump    evcond        ;tail recursive call to evcond
...
notdef: getcar  unev, exp     ;the apply case
        push    unev          ;save unevaled operator
        push    env
        getcdr  unev, exp     ;unevaled arguments
        jsr     evlist        ;non tail recursive -- arg vals
        pop     env
        pop     exp           ;restoring unev to exp!
        push    val           ;save the list of values
        jsr     mceval        ;call self to get proc
        load    proc, val     ;1st argument for apply
        pop     args          ;pop list of vals
        jump    mcapply       ;and tail recursively go apply
```

All the pieces of the meta-circular evaluator are similiar easy to translate. E.g.,

```
(define evlist
  (lambda (lst env)           ;map evaluator over list
    (cond ((null? lst) '())
          (else (cons (mc-eval (car lst) env)
                      (evlist (cdr lst) env))))))
evlist: gettag  tag, unev
        cmp     tag, #4      ;is it empty list?
        jump.ne evbody
        load    val, unev    ;simply return empty list
        rts
evbody: getcar  exp, unev    ;save this for later for mc-eval
        push    exp
        getcdr  unev, unev   ;cdr down list
        push    env          ;and save what else we need
        jsr     evlist       ;call self
        pop     env          ;restore what we saved
        pop     exp
        push    val          ;save tail of evaled elements
        jsr     mceval       ;evaluate first element
        load    arg1, val    ;make it the car argument to cons
        pop     val          ;get previous saved tail
        load    arg2, val    ;make it the cdr argument to cons
        jump    cons         ;tail recursively let cons do the work
```

What about applying primitive procedures. Rather than inherit them from the underlying Scheme like we did in our meta-circular evaluator we need to actually implement all the primitives. Let us assume that all the primitives take exactly 1 or 2 arguments, and that they will be passed in registers `arg1` and `arg2`, as we had already agreed for `cons`. A primitive procedure has as its address part the address of some code that will implement the primitive operation and return the result in the `val` register (just as with cons).

We will modify the first dispatch clause in `apply` from

```
(cond ((not (pair? fun)) (apply fun args))   ;ground out
```

to be (in pseudo-Scheme) something like

```
(cond ((primproc? fun) (primapply fun args))
```

Now our apply procedure looks like below, where `primproc?` and `primapply` are pseudo Scheme procedures which will never exist in Scheme, but which we'll write directly in assembly code.

```
(define mc-apply
  (lambda (fun args)
    (cond ((primproc? fun) (primapply fun args)) ;ground out
          ((eq? (car fun) 'proc)
           (mc-eval (car (cdr (car (cdr fun))))  ;procedure body
                    (bind (car (car (cdr fun)))  ;formal params
                          args                   ;supplied args
                          (car (cdr (cdr fun))))));saved env
          (else (error '"Unknown function")))))
```

Now recall that we decided that our assembly translation of this procedure as `mcapply` should take its two arguments in registers called `proc` and `args`, and return its result in the `val` register. The code for the first clause becomes:

```
mcapply:gettag  tag, proc
        cmp     tag, #6
        jump.ne comp
        getcar  arg1, args
        getcdr  args, args
        gettag  tag, args
        cmp     tag, #2
        jump.ne apprim
        getcar  arg2, args
apprim: jump    0(proc)
comp:   ...
```

Notice that at the `apprim` label we jump off to the address specified in the low order 28 bits of the primitive. This should be the address of some appropriate code that will take arguments in `arg1` and possible `arg2`, and return a value in the `val` register. Since the `jump` above is a tail recursive call, it is up to the code for the primitive to find the return address on the stack. It will also be imperative that the primitives return results with the correct types. So let us suppose there is another opcode `settag` for our machine which takes a register and a constant tag, and sets the upper 4 bits of the contents of the register to be that tag. Most modern architectures can do something like this in a single instruction.

Here is the code for some representative primitives[2], `+`, `-`, `car`, `cdr`, and `pair?`.

[2] Note that we are implementing a rather primitive version of Scheme here. All numbers are restricted to be 28

```
add:    add     arg1, arg2
        load    val, arg1
        settag  val, #3
        rts
sub:    sub     arg1, arg2
        load    val, arg1
        settag  val, #3
        rts
car:    getcar  val, arg1
        rts
cdr:    getcdr  val, arg1
        rts
pair?:  load    val, #true
        gettag  tag, arg1
        cmp     tag, #2
        jump.eq ispair
        load    val, #false
ispair: rts
```

5. Storage Management

We have explained how everything works except for storage management and reading and printing. In this section we cover storage management.

There are two things to consider here–how we allocate new storage, such as for a `cons` cell, and why we never have to run out of memory.

In the following we will write Scheme code as an illustration of the algorithms we are going to implement. We could not directly write these things in Scheme code like this, even with a good compiler[3], because we will be using `+`, for instance, to do machine addition, rather than Scheme addition of typed numbers. In other places we use variables that appear to be in the global environment where we mean registers. The Scheme code is meant to convey the essence of the computation. The assembly code is rather more accurate.

We have represented a cons cell as two successive 32 bit quantities, the `car`, and `cdr`. Since we are using a byte addressing machine that is 8 bytes of RAM. A simple implementation would be to have a dedicated register, `free`, say with the address of some unused storage. Whenever `cons` is called it should fill in the `car` and `cdr` at that location, bump the `free` pointer by 8 bytes, and return a pointer to the original location with a pair type tag (recall that pairs in our implementation use a type tag of 2).

We need two new Scheme pseudo primitives, `setcontents!` and `change-type` to write pseudo Scheme code for this.

---

bit integers. A full scale Scheme implementation would have more complex number types, and the add and subtract primitives would have to dispatch on types, do coercions, etc.

[3]In real Lisp systems the garbage collector, `cons`, etc., are written in Lisp itself, then compiled by a high quality compiler. The difference is that a completely separate set of special primitives must be used so that the compiler gets the right machine instructions and does not do undue typing.

```
(define cons
  (lambda (x y)
     (let ((ptr free))
       (set! free (+ 8 free))
       (setcontents! ptr x)
       (setcontenst! (+ ptr 4) y)
       (change-type free 2))))
```

It turns out to be even simpler in assembly code. Note that we need one more machine instruction `store`, which takes a register as its source, and stores the contents at the destination, which is specified by a register and a small constant offest.

```
cons:   load    val, free
        add     free, #8
        store   0(val), arg1
        store   4(val), arg2
        settag  val, #2
        rts
```

The problem with this version of `cons` is that it does not check for the case of running out of storage. A better version would be one which checks against an upper bound, stored in a register called `fromend`, and calls a magic procedure `gc`, for garbage collector, which arranges for there to be some more storage when we run out. Our `cons` then becomes:

```
(define cons
  (lambda (x y)
     (if (>= free fromend)
         (begin (gc)
                (cons x y))
       (let ((ptr free))
         (set! free (+ 8 free))
         (setcontents! ptr x)
         (setcontenst! (+ ptr 4) y)
         (change-type free 2)))))
```

And the equivalent assembly code is:

```
cons:   cmp     free, fromend
        jump.lt docons
        push    arg1
        push    arg2
        jsr     gc
        pop     arg2
        pop     arg1
        jump    cons
docons: load    val, free
        add     free, #8
        store   0(val), arg1
        store   4(val), arg2
        settag  val, #2
        rts
```

Note that we save the `arg1` and `arg2` registers on the stack before we call the `gc` because we don't know what it will do to those registers.

6. Garbage Collection

How does `gc` work? There are a large number of garbage collection algorithms. We will show just one here, called the "stop and copy" algorithm, originally due to Marvin Minsky and his students.

The essential thing to notice is that as we cons things together and use them they often become inaccessible. Once a cons cell is inaccessible it would be okay to re-use it in place of a fresh cons. The problem then is to find those cons cells which are no longer accessible.

What cons cells are accessible. Certainly any that are referred to in by elements on the stack **might** eventually need to be used, so we shouldn't collect those and throw them out. Also at an instantaneous point anything referred to by any of the registers is accessible. But due to our calling convention introduced in section 4 by the time we get to `gc` all our registers are saved on the stack. Recursively the same is true of any `car` and `cdr` of anything that is directly accessible. At first glance then it looks like we will at the very least need to do a tree walk from every element of the stack to find what is accessible and what is not[4]. The stop and copy algorithm does this a different way using a non-recursive procedure. The penalty is that at most half the storage can ever be used at one time.

Consider the following set of operations in a "heap" that only has five cons cells, i.e., 40 bytes of storage.

```
(define x (cons 4 (cons 7 (cons 6 (cons 8 '())))))
(define x (cons (cdddr x) (cddr x)))
(set-cdr! (car x) x)
```

If this heap began address 100, and we use `P` to represent a pair type (2 in our implementation), `N` to represent a number type (3), and `E` for the empty list (4), then our memory would look like:

```
100:  N008
104:  P132
108:  N006
112:  P100
116:  N007
120:  P108
124:  N004
128:  P116
132:  P100
136:  P108
```

and `x` will be `P132`. Thus our only "root" is the fifth cons cell.

Let us call the heap above our *from space*. The stop and copy algorithm is to directly copy over all the root cons cells to a new *to space*, then linearly scan the new space copying over anything it points to. It uses a `free` pointer pointing directly to the next available location in to space, and a `scan` pointer which points to the next location to be checked in to space. The `scan` pointer is scanned through to space and each pointer which is a pair is *forwarded*. If the old pair has not been copied already then it is copied, `free` is bumped up, and a *forwarding pointer* is left on the old from space. If it has already been copied then the forwarding point already there is used to construct a new pointer to be placed in the current location in to space. When `scan` catches up to `free` the process is complete.

[4] And running into difficulties with circular data structures created with `set-car!` and `set-cdr!`.

Suppose our to space starts at location 140. Then in the above example `free` and `scan` are initialized to 140, and once the algorithm is done the are both at 164, as there are only three good cons cells. The root `x` starts with a value of `P132`, and ends with a value of `P140` as that is the first cons cell copied.

```
100:  F148      140:  P148
104:  P132      144:  P156
108:  F156      148:  N008
112:  P100      152:  P140
116:  N007      156:  N006
120:  P108      160:  P148
124:  N004      164:
128:  P116      168:
132:  F140      172:
136:  P108      176:
```

Here we use `F` as out forwarding pointer tag. Let us suppose that in our real implementation we use 13 as the forwarding pointer tag value.

Let us suppose that we have four registers dedicated to pointing to the start and end of from space and to space, namely `tostart`, `toend`, `fromstart`, and `fromend`. The full `gc` algorithm is as follows.

```
(define (gc)
  (set! free tostart)
  (gccopystack)
  (gccopy tostart)
  (let ((oldtostart tostart)
        (oldtoend toend))
    (set! tostart fromstart)
    (set! toend fromend)
    (set! fromstart oldtostart)
    (set! fromend oldtoend)))
```

Most of the code is dedicated to swapping the roles of to and from space once the copying is done. There are two almost identical routines called to do the bulk of the work. `gccopystack` simply copies the stack into to space fixing up pointers as it goes. It differs from `gccopy` which is used to scan to space, only in its termination condition. In a real implementation they would be the same procedure. We just write `gccopy` here for simplicity:

```
(define (gccopy scan)
  (if (/= scan free)
    (let ((pointer (contents scan)))
      (if (pair? pointer)
        (let ((new (forward pointer)))
          (setcontents! scan new)))
      (gccopy (+ scan 4))))))
```

This uses `forward` to copy over the actual cons cells.

```
(define (forward pointer)
  (let ((oldcar (contents pointer)))
    (if (forward? oldcar)
(change-type oldcar 2)
      (begin (setcontents! free oldcar)
             (setcontents! (+ free 4)
                           (contents (+ pointer 4)))
             (let ((newadd free))
               (set! free (+ 8 free))
               (setcontents! pointer
                             (change-type newadd 13))
               (change-type newadd 2))))))
```

These can easily be translated into very compact code, and these few machine instructions then automatically manage all our storage use for us.

```
gccopy: cmp     scan, free
        jump.eq gcdone
        load    arg1, 0(scan)
        gettag  tag, arg1
        cmp     tag, #3
        jump.ne gccont
        jsr     forward
        store   0(scan), val
gccont: add     scan, #4
        jump    gccopy
gcdone: rts

forward:load    val, 0(arg1)
        gettag  tag, val
        cmp     tag, #13
        jump.ne dowork
        settag  val, #2
        rts
dowork: store   0(free), val
        load    arg2, 4(arg1)
        store   4(free), arg2
        load    val, free
        add     free, #8
        settag  val, #13
        store   0(arg1), val
        settag  val, #2
        rts
```