Low-Level Representations

sk and dbtucker

2002-10-23

1 A Sample Program

Let's begin with the following program.

```
(define (filter-pos l)
  (cond
    [(empty? l) empty]
    [else
     (if (> (first l) 0)
         (cons (first l) (filter-pos (rest l)))
         (filter-pos (rest l)))]))
```

Its representation in CPS is

```
(define (filter-pos/k l k)
  (cond
    [(empty? l) (k empty)]
    [else
     (if (> (first l) 0)
         (filter-pos/k (rest l)
                       (lambda (v)
                         (k (cons (first l) v))))
         (filter-pos/k (rest l) k))]))

(define (filter-pos l)
  (filter-pos/k l (lambda (x) x)))
```

The next thing to do is change the representation of the stack, first to a list of procedures:

```
(define (filter-pos/k l stack)
  (cond
    [(empty? l) (Pop empty stack)]
    [else
     (if (> (first l) 0)
         (filter-pos/k (rest l)
                       (Push (lambda (v s)
                               (Pop (cons (first l) v)
                                    s))
                             stack))
         (filter-pos/k (rest l) stack))]))

(define (filter-pos l)
  (filter-pos/k l EmptyStack))

(define EmptyStack empty)
```

```
(define (Push frame stack) (cons frame stack))
(define (Pop value stack)
  (cond
    [(empty? stack) value]
    [else ((first stack) value (rest stack))]))
```

Now we convert the representation of the stack frames so that they use a list of datatype records rather than procedures.

```
(define-datatype StackFrame StackFrame?
  [terminal-frame]
  [filter-frame (n number?)])

(define (filter-pos/k l stack)
  (cond
    [(empty? l) (Pop empty stack)]
    [else
     (if (> (first l) 0)
         (filter-pos/k (rest l)
                       (Push (filter-frame (first l))
                             stack))
         (filter-pos/k (rest l) stack))]))

(define (filter-pos l)
  (filter-pos/k l EmptyStack))

(define EmptyStack (list (terminal-frame)))
(define (Push frame stack) (cons frame stack))
(define (Pop value stack)
  (cases StackFrame (first stack)
    [terminal-frame () value]
    [filter-frame (n) (Pop (cons n value) (rest stack))]))
```

So far, this is pretty similar to what we saw before. At this point, however, we're ready to first make an observation, then branch off into even lower-level representations.

2 Tail Calls

Study the code in *filter-pos/k* more carefully. There are two recursive calls in that body, and they differ slightly. By reading the original version of the program (*filter-pos*), it's easy to tell that the difference is whether or not we want to retain the first element of the list.

Now let's look at the corresponding calls in the explicit stack version. The version that retains the first element becomes

```
(filter-pos/k (rest l)
              (Push (filter-frame (first l))
                    stack))
```

while the other version is

```
(filter-pos/k (rest l) stack))]))
```

From these differences, a few things become clear:

- As we discussed in our previous lecture, the stack exists solely to *evaluate arguments*, not for making function calls. It so happens that in this case, the call to *filter* is itself a mere "argument evaluation"—from the perspective of the pending *cons*.

- After the stack has grown, the function invocation itself does not need to depend on any stack. We discussed this in our lengthy build-up to CPS, where we said to consider the Web programming context, where any remaining computation is lost anyway. Therefore, the actual function call is itself simply a direct jump, or "goto".

- Converting the program to CPS helps us clearly see which calls are already just gotos, and which ones need stack build-up. The ones that are just gotos are those invocations that use the same continuation argument as the one they received. Those that build a more complex continuation are relying on the stack.

Procedure calls that do not place any burden on the stack are called *tail calls*. Converting a program to CPS helps us identify tail calls, though it's possible to identify them from the program source itself. An invocation of g in a procedure f is a tail call if, in the control path that leads to the invocation of g, the value of f is the same as the value computed by g. In that case, g can send its value directly to whoever is expecting f's value. This insight is employed by compilers to perform a *tail call optimization*, whereby they ensure that tail calls incur no stack growth.

Here are just a few of the issues that arise as a consequence of the notion of tail calls:

- With tail calls, it no longer becomes necessary for a language to provide looping constructs. Whatever was previously written using a custom-purpose loop can now be written as a recursive procedure. So long as all recursive calls are tail calls, the compiler will convert the calls into gotos, accomplishing the same efficiency as the loop version. For instance, here's a very simple version of a `for` loop, written using tail calls:

```
(define (for init condition change body result)
  (if (condition init)
      (for (change init)
           condition
           change
           body
           (body init result))
      result))
```

  By factoring out the invariant arguments, we can write this more readably as

```
(define (for init condition change body result)
  (local [(define (loop init result)
            (if (condition init)
                (loop (change init)
                      (body init result))
                result))]
    (loop init result)))
```

  To use this as a loop, write

```
(for 10 positive? sub1 + 0)
```

  which evaluates to 55. It's possible to make this look more like a traditional `for` loop using macros, which we will discuss later this semester. In either case, notice how similar this is to a *fold* operator! Indeed, *foldl* employs a tail call in its recursion, meaning it is just as efficient as looping constructs in more traditional languages.

- While tail calls are traditionally associated with functional languages such as Scheme and ML, there's no reason they must be. It's perfectly possible to have tail calls in any language. Indeed, as our analysis above has demonstrated, tail calls are the *natural consequence of understanding the true meaning of function calls*. A languages that deprives you of tail calls is cheating you of what is rightfully yours—stand up for your rights! Because so many language designers and implementors mistreat their users, however, programmers have come to think of tail calls as inherently expensive, even though they're not.

- A special case of a tail call is known as *tail recursion*, which occurs when the tail call within a procedure is to itself. This is the behavior we see in the procedure *for* above. In fact, tail recursion is only a special case of tail calls in general. While it is an *important* special case (since it enables the implementation of loops), it is not the most *interesting* case.

Sometimes, programmers will find it natural to split a computation across two procedures, and use tail calls to communicate between them.[1] This leads to very natural program structures. A programmer using a language like Java, however, is forced into an unpleasant decision. If they split code across methods, they pay the penalty of method invocations that use the stack needlessly. But even if they combine the code into a single procedure, it's not clear that they can easily turn the two code bodies into a single loop. Even if they do, the structure of the code has now been altered irrevocably. Consider the following example:

```
(define (even? n)
  (if (zero? n)
      true
      (odd? (sub1 n))))

(define (odd? n)
  (if (zero? n)
      false
      (even? (sub1 n))))
```

Therefore, even if a language gives you tail recursion, remember that you are getting less than you deserve. Indeed, it sometimes (but not always: there are notable counterexamples to the following claim) suggests a particularly clueless language implementor because they realized that the true nature of function calls permitted calls that consumed no new stack space, but restricted its use, possibly owing to a failure of imagination. The primitive you really want a language to support is tail *calling*. With it, you can express solutions more naturally, and also build very interesting abstractions of control flow patterns.

- Note that CPS converts every program into a form where every call is a tail call!

3 Machine-Level Representations

We now return to the representations used by *filter-pos/k*. When we last left our hero, the program used datatype representations for stack frames and a list for the stack itself. The list that represents the stack is, however, a pretty high-level data structure. Furthermore, the program itself uses lists, which are definitely exploiting the primitives of Scheme. Let's eliminate each of these.

3.1 Replacing the Stack

Let's choose a more primitive representation for the stack. We'll use a vector of stack frames. We won't bother changing the datatype itself, though it's easy if we wanted to do so: simply use a number or symbol to tag each variant, and so on. We will illustrate this principle in the next section. For now, let's focus on the mechanics of using a vector instead of a list.

We could pass the stack vector around, but the reason we defined it in the first place is to more accurately capture the machine, and in the machine procedures do not "pass the stack as arguments". Therefore, we'll use a global stack instead:

```
(define STACK-SIZE 1000)
(define STACK-POINTER (box 0))
(define Stack (make-vector STACK-SIZE))
```

We'll find it useful to have the following primitives to modify the stack pointer:

```
(define (increment-box b)
  (set-box! b (add1 (unbox b))))
(define (decrement-box b)
  (set-box! b (sub1 (unbox b))))
```

[1] They may not even communicate mutually. In the second version of the loop above, *for* invokes *loop* to initiate the loop. That call is a tail call, and well it should be, otherwise the entire loop will have consumed stack space. Because Scheme has tail calls, notice how effortlessly we were able to create this abstraction. If the language supprted only tail *recursion*, the latter version of the loop, which is more pleasant to read and maintain, would actually consume stack space against our will.

We must rewrite *filter-pos/k* to no longer consume or pass the stack:

```
(define (filter-pos/k l)
  (cond
    [(empty? l) (Pop empty)]
    [else
     (if (> (first l) 0)
         (begin
           (Push (filter-frame (first l)))
           (filter-pos/k (rest l)))
         (filter-pos/k (rest l)))]))
```

(Notice how we have undone the effect of transforming to CPS!) We have to now define the stack primitives:

```
(define (Push frame)
  (begin
    (increment-box STACK-POINTER)
    (vector-set! Stack (unbox STACK-POINTER) frame)))

(define (Pop value)
  (cases StackFrame (vector-ref Stack (unbox STACK-POINTER))
    [terminal-frame () value]
    [filter-frame (n)
                  (begin
                    (decrement-box STACK-POINTER)
                    (Pop (cons n value)))]))
```

Finally, the interface procedure:

```
(define (filter-pos l)
  (begin
    (vector-set! Stack 0 (terminal-frame))
    (filter-pos/k l)))
```

As this point, we've entirely transformed the stack's representation: the only lists left in the program are those consumed and produced by the *filter* procedure itself, but the stack is now very close to its machine representation. In particular, notice that the tail call in the body of *filter* has become

```
(filter-pos/k (rest l))
```

while the non-tail invocation is

```
(begin
  (Push (filter-frame (first l)))
  (filter-pos/k (rest l)))
```

which very nicely captures the distinction between the two!

3.2 Replacing Lists

Our last task to finish hand-compiling this program is to get rid of the Scheme lists used in this example. What is a list, anyway? A list has two references: one to its first value, and one more to the rest of the list. At the machine level, we need to hold one more item of information, which is a tag to differentiate between lists and other kinds of values (and, also, between empty and non-empty lists).

We've learned from basic computer systems courses that dynamic data structures must be stored on the heap. It's very important that we not accidentally allocate them on the stack, because data such as lists have *dynamic extent*, whereas the stack only represents *static scope*. We discussed this in a little detail when we introduced state (identifiers have static scope while the values they are bound to have dynamic extent; the environment holds values of free variables, which in turn are what reside in a stack frame, so the stack really does represent the environment, while the store ranges globally). We would rather play it safe and allocate data on the heap, then determine how to get rid of unwanted ones, than to allocate them on the stack and create dangling references (the stack frame might disappear while there are still references to data in that frame; eventually the frame gets overwritten by another procedure, at which point the references that persist are now pointing to nonsense).

Having established that we need a global place for values, known as the *heap*, we'll model it the same way we do the stack: as a vector of values. (All the stack code remains the same; we'll continue to use the global vector representation of stacks, so *filter-pos/k* will consume only one argument.)

```
(define HEAP-SIZE 1000)
(define HEAP-POINTER (box 0))
(define Heap (make-vector HEAP-SIZE 'unused))
```

The following procedure will come in handy when manipulating either the heap or the stack: *entity* refers to a vector, such as one of those two, *pointer* is an address in that vector, and *value* is the new value to assign to that location.

```
(define (set-and-increment entity pointer value)
  (begin
    (vector-set! entity (unbox pointer) value)
    (increment-box pointer)))
```

Now we're ready for the heart of the implementation. How shall we represent a *cons* cell? We'll define the procedure *NumCons* of two arguments. Both will be numbers, but we'll assume that the first represents a literal number, while the second represents a location.

Clearly, we must allocate new space in the heap. The cell will therefore range over several locations. When it does so, we must pick a canonical location to represent the cell. As a convention (which reflects what many compilers do), we'll pick the first. Therefore, the code for *NumCons* might look like this:

```
(define (NumCons v1 v2)
  (local [(define starting-at (unbox HEAP-POINTER))]
    (begin
      (set-and-increment Heap HEAP-POINTER v1)
      (set-and-increment Heap HEAP-POINTER v2)
      starting-at)))
```

Having given the ability to create new cons cells, we must also be able to determine when we are looking at one, so that we can implement a procedure like *cons?*. Unfortunately, our representation hasn't given us the ability to distinguish between different kinds of values. There's an easy way to solve this: we simply deem that at the location representing a value, we will always store a tag that tells us what kind of value we're looking at. This leads to

```
(define (NumCons v1 v2)
  (local [(define starting-at (unbox HEAP-POINTER))]
    (begin
      (set-and-increment Heap HEAP-POINTER 'num-cons)
      (set-and-increment Heap HEAP-POINTER v1)
      (set-and-increment Heap HEAP-POINTER v2)
      starting-at)))
```

so that we can define

```
(define (NumCons? location)
  (eq? (vector-ref Heap location) 'num-cons))
```

Similarly, we can also define

```
(define (NumEmpty)
  (local [(define starting-at (unbox HEAP-POINTER))]
    (begin
      (set-and-increment Heap HEAP-POINTER 'num-empty)
      starting-at)))
```

```
(define (NumEmpty? location)
  (eq? (vector-ref Heap location) 'num-empty))
```

Having defined this representation, we can now easily define the analog of *first* and *rest*:

```
(define (NumFirst location)
  (if (NumCons? location)
      (vector-ref Heap (+ location 1))
      (error 'NumFirst "not a NumCons cell")))
(define (NumRest location)
  (if (NumCons? location)
      (vector-ref Heap (+ location 2))
      (error 'NumRest "not a NumCons cell")))
```

Given these new names for primitives, we rewrite code to use them:

```
(define (filter-pos/k l)
  (cond
    [(NumEmpty? l) (Pop (NumEmpty))]
    [else
     (if (> (NumFirst l) 0)
         (begin
           (Push (filter-frame (NumFirst l)))
           (filter-pos/k (NumRest l)))
         (filter-pos/k (NumRest l)))]))
```

While most of the stack code can stay unchanged, any code that does refer to lists must adapt to the new primitives, which signify a new implementation:

```
(define (Pop value)
  (cases StackFrame (vector-ref Stack (unbox STACK-POINTER))
    [terminal-frame () value]
    [filter-frame (n)
                  (begin
                    (decrement-box STACK-POINTER)
                    (Pop (NumCons n value)))]))
```

Finally, we have to be careful to not use Scheme's primitive lists inadvertently. For instance, invoking

```
> (filter-pos '(0 2 3 -1 -5 1 -1))
vector-ref: expects type <non-negative exact integer> as 2nd argument, ···
```

Instead, we must write

```
> (filter-pos
   (NumCons 0
   (NumCons 2
   (NumCons 3
   (NumCons −1
   (NumCons −5
   (NumCons 1
   (NumCons −1 (NumEmpty)))))))))
```

When we run this test case, we get the surprising result . . . 29.

29? We were supposed to get a list back. How did a series of list operations result in a number, that too one that wasn't even on our original list?

The answer is simple: the procedure now returns the *location* of the answer. Indeed, in a language like Scheme, locations *are* the representation of values. Everything is a location. What we would find handy is a procedure that extracts the value at that location and makes it suitable for human consumption:

```
(define (location->list location)
  (if (NumEmpty? location)
      empty
      (cons (NumFirst location)
            (location->list (NumRest location)))))
```

With this, we finally get the desired interaction:

```
> (location->list
    (filter-pos
     (NumCons 0
     (NumCons 2
     (NumCons 3
     (NumCons −1
     (NumCons −5
     (NumCons 1
     (NumCons −1 (NumEmpty)))))))))
(list 2 3 1)
```

Puzzles

- A program that consumes some amount of stack, when converted to CPS and run, suddenly consumes no stack space at all. What's going on?
- Our implementation of *NumEmpty* is displeasing, because it allocates a new tag every single time. In fact, there is really only one empty list in the world, so you would think they could share their representation and thereby consume less heap space. Convert the procedure *NumEmpty* so it allocates space for the empty list only the first time is invoked, and returns the same address on every subsequent invocation.

  Note:

  1. If you find it easier, you may allocate space for the empty list before the user's program even begins execution.
  2. The idea of using a single instance in place of numerous isomorphic instances, to save both construction time and space, is known as the Singleton Pattern in the *Design Patterns* book (Gamma, et al).