Chapter 22

Continuations and Compilation: Machine Representations

Since we are simulating the process of compilation, we must shed as much dependency on Scheme as possible. The tranformed version of the procedure *filter-pos/k* (Section 21.1.3) uses structure 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.

22.1 The Stack in Memory

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 around". 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))))
```

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)
  (type-case 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!

22.2 Lists in Memory

Our last task to finish hand-compiling this program is to eliminate 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 would rather play it safe and allocate data on the heap, then determine how to eliminate 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 new value we write into the heap will range over several locations. We must therefore pick a canonical location to represent the cell. As a convention (which reflects what many compilers do), we'll pick the first heap location. Therefore, the code for *NumCons* is as follows:

```
(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, reflecting the new implementation strategy for lists:

```
(define (Pop value)
  (type-case 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, ...
```

results in an error; we must instead 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! 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. (Recall from Section 13 that locations are values for mutable data structures.) We would, nevertheless, find useful a procedure that prints the content of this location in a fashion more 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)
```

Exercise 22.2.1 *Our implementation of NumEmpty is displeasing, because it allocates a new tag every single time. In fact, we need only one instance of the empty list, so you would think all these instances 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 book* Design Patterns.