Compilation with Explicit Stacks

sk and dbtucker

November 13, 2000

The first step in our compilation process was to convert the program to CPS, which made the stack explicit. The second step was to pass arguments in registers, so that function calls could be modeled by jumps in assembly language. We are left with one construct, *cons*, that we need to translate to a lower level. We saw that there are two different uses of *cons* in our *filter* example, one for the stack representation, and one for the list being passed to *filter*. Today we'll see how to deal with the stack-related uses of *cons*.

So far we have been doing the easy part of the compiler: code generation. There are two parts that are more difficult:

1. The *run-time system* is responsible for providing primitive operations (*even?*, *cons*, *car*, &c.), some abstraction for memory management, and an interface to the operating system (for I/O, process management, &c.).

2. *Optimization*, which we won't cover in this course.

The technique we are using is compilation by transformation. There are several phases in this transformation; the result of each phase is a Scheme program that uses less sophisticated constructs, and relies more on ones that can be mapped directly to the machine. Furthermore, each resulting program is guaranteed to be semantically equivalent to the original program.

The stack

We will model the stack using a vector, which we treat as a chunk of memory. The stack pointer indicates the current top of the stack:

```
(define Stack (make-vector 100))
(define Stack-Ptr 0)
```

We want to rewrite the *Push-* and *Pop-* functions so that they manipulate this stack. Recall the previous definition of *Push-test-pred*:

```
(define (Push-test-pred f/k l Stack)
  (cons (list 'Push-test-pred f/k l)
        Stack)))
```

Instead of making a list and *cons*ing that onto the stack, we'll add each item individually. The new version of *Push-test-pred* mutates the global stack and updates the stack pointer:

```
(define (Push-test-pred f/k l k)
  (vector-set! Stack (+ 0 Stack-Ptr) 'Push-test-pred)
  (vector-set! Stack (+ 1 Stack-Ptr) f/k)
  (vector-set! Stack (+ 2 Stack-Ptr) l)
  (set! Stack-Ptr (+ Stack-Ptr 3)))
```

The definition of *Push-cons-first* is similar. Now let's rewrite *Pop-common*:

```
(define (Pop-common v k)
  (if (zero? Stack-Ptr)
      v
      (let ([tag ...])
        ...)))
```

We have a problem here—if the top frame's tag is 'Push-test-pred, it will be the *third* element from the top of the stack, but if the tag is 'Push-cons-first, it will be *second* from the top. We really should have put the tag on the top of the stack:

```
(define (Push-test-pred f/k l k)
  (vector-set! Stack (+ 0 Stack-Ptr) f/k)
  (vector-set! Stack (+ 1 Stack-Ptr) l)
  (vector-set! Stack (+ 2 Stack-Ptr) 'Push-test-pred)
  (set! Stack-Ptr (+ Stack-Ptr 3)))
```

If we do the same with *Push-cons-first*, the tag is always the top element of the stack (i.e., one less than the value of the stack pointer). Now we can continue writing *Pop-common*:

```
(define (Pop-common v k)
  (if (zero? Stack-Ptr)
      v
      (let ([tag (vector-ref! Stack (- Stack-Ptr 1))])
        (case tag
          [(Push-test-pred)
           (let ([f/k (vector-ref Stack (- Stack-Ptr 3))]
                 [l (vector-ref Stack (- Stack-Ptr 2))])
             (if v
                 (filter/k f/k (rest l) (Push-cons-first l 'dummy))
                 (filter/k f/k (rest l) 'dummy)))]
          [(Push-cons-first)
           (let ([l (vector-ref Stack (- Stack-Ptr 2))])
             (Pop-filter (cons (first l) v) 'dummy))])))
```

This version of *Pop-common* isn't correct. Notice that since we are using a global stack, we just pass a dummy value (instead of *rest-frames*) as the third

argument to *filter/k*. However, when *filter/k* is called, *Stack-Ptr* will still point to the top frame of the stack, rather than the second frame from the top (which corresponds to *rest-frames*). We need update the value of *Stack-Ptr* so that it refers to the second frame from the top (in other words, we pop the top frame of the stack). The correct version of *Pop-common* is:

```
(define (Pop-common v k)
  (if (zero? Stack-Ptr)
      v
      (let ([tag (vector-ref! Stack (- Stack-Ptr 1))])
        (case tag
          [(Push-test-pred)
           (let ([f/k (vector-ref Stack (- Stack-Ptr 3))]
                 [l (vector-ref Stack (- Stack-Ptr 2))])
             (set! Stack-Ptr (- Stack-Ptr 3))
             (if v
                 (filter/k f/k (rest l) (Push-cons-first l 'dummy))
                 (filter/k f/k (rest l) 'dummy)))]
          [(Push-cons-first)
           (let ([l (vector-ref Stack (- Stack-Ptr 2))])
             (set! Stack-Ptr (- Stack-Ptr 2))
             (Pop-filter (cons (first l) v) 'dummy))])))
```

Note that we made a similar modification for the *Push-cons-first* case.

We have covered the code on page 1 of today's handout. Here are short explanations of the further changes we made to the program:

- On page 2, we've removed the k argument (e.g. the third argument to *filter/k*. We saw that this argument is completely useless since we're now using *Stack*.

- On page 3, we break *Pop-common* into three functions: *Empty*, *filter/split/Push-test-pred*, and *filter/split/Push-cons-first*. Now instead of pushing a tag ('Empty, 'Push-test-pred, or 'Push-cons-first) onto the stack, we push a reference to the corresponding function. Then *Pop-common* simply pulls the top function off the stack and invokes it.

- On page 4, we return values through the stack.