Compilation: Tail-Calls and Register Form

sk and dbtucker

November 8, 2000

We are exploring how to compile programs in our language. The technique we are using is program transformation: each phase of compilation will transform the program into another program that is semantically equivalent, but more closely resembles assembly language. The first step in the process was to convert the program into CPS, which made the stack an explicit entity. We showed how to put an example program, *filter*, into CPS form. We will continue with our hand compilation of the *filter* function in this lecture.

Tail calls

What is a tail call? It's a call to a function where the caller doesn't do any computation with the value returned. For example, in *filter*:

```
(define (filter f l)
  (if (empty? l)
      l
      (if (f (first l))
          (cons (first l) (filter f (rest l)))
          (filter f (rest l)))))
```

the call to *filter* on the last line is a tail call, whereas the call on the second-to-last line is not. *Tail recursion* is when you make a tail call to the same function, as in the recursive call to *filter* in this example.

Look at the code for the CPS version of *filter*, called *filter/k*. Where does the above tail call occur?

```
(define (filter/k f/k l k)
  (if (empty? l)
      (k l)
      (f/k (first l)
           (λ (v)
             (if v
                 (filter/k f/k
                           (rest l)
                           (λ (v)
                             (k (cons (first l) v))))))
```

```
(filter/k f/k
          (rest l)
          k))))))
```

The second call to *filter/k* is the tail call we are looking for. Note that since we aren't adding any computation to the current context, the context passed to the recursive call is k, which is the same context that was passed in. We'll use this property to define tail calls:

> Turn the program into CPS. A function call is a *tail call* if the context you use for the call is the same as the context you receive.

Where are the non-tail calls in the CPS program *filter/k*? The expression (*empty? l*) is not a tail call, since its value used as the condition of an **if**-expression, but *empty?* is a primitive. Other than primitives, in a CPS program, all function calls are tail calls. If this is true, where did Scheme's stack go? We made it explicit—it's the parameter k.

Now you can see how we are getting closer to assembly language. A tail call can be represented in assembly language by just a jump. So, we can translate all our function calls down to assembly by turning each function into a label.

The stack

Now look at the code for *filter/k* where the continuation operations for both *filter* and *even?* have been abstracted into *Push*es and *Pop*s.

```
(define filter/k
  (λ (f/k l k)
    (if (empty? l)
        (Pop-filter l k)
        (f/k (first l)
             (Push-test-pred f/k l k)))))

(define Push-test-pred
  (λ (f/k l k)
    (λ (v)
      (if v
          (filter/k f/k
                    (rest l)
                    (Push-cons-first l k))
          (filter/k f/k
                    (rest l)
                    k)))))
```

What do the calls to *Push-test-pred* and *Push-cons-first* correspond to in the original (non-CPS) *filter*?

- The call to *Push-test-pred* corresponds to the (*first l*) in the (*f* (*first l*)).

- The call to *Push-cons-first* corresponds to the the (*first l*) in (*cons* (*first l*) (*filter* ...)).

The moral is that the stack helps you manage arguments, not function calls. If all arguments were values, we wouldn't need a stack.

Passing arguments in registers

If we turn the functions into labels, and the function calls into jumps, where do the arguments go? In assembly language, you would put each argument in a register. We'll simulate registers with variables written like =*arg1*=, and we'll use **set!** to mutate these variables.

We want to transform *filter/k* so that the arguments are passed in registers. First, we rename the parameters: *f/k* to =*arg1*=, *l* to =*arg2*=, and *k* to =*Stack*=. The registers start off with meaningless values:

```
(define =arg1= 'undefined)
(define =arg2= 'undefined)
(define =Stack= 'undefined)
```

Now we remove the parameters from function, and rename the variables in the body to use these "registers":

```
(define (filter/k)
  (if (empty? =arg2=)
      (Pop-filter =arg2= =Stack=)
      (=arg1= (first =arg2=)
              (Push-test-pred =arg1= =arg2= =Stack=)))))
```

Since *f/k* is itself a function, we must pass its arguments in registers as well:

```
(define (filter/k)
  (if (empty? =arg2=)
      (Pop-filter =arg2= =Stack=)
      (begin
        (set! =arg1= (first =arg2=))
        (set! =arg2= (Push-test-pred =arg1= =arg2= =Stack=))
        (=arg1=)))))
```

Oops! We mutated the register =*arg1*= (which was *f/k*) before we invoked it. We need to use temporary registers in order to hold on to the values passed in. The correct version of the registerized code is this (we renamed it *filter/k/reg*):

```
(define (filter/k/reg)
  (let ([arg1/in =arg1=]
        [arg2/in =arg2=]
        [Stack/in =Stack=])
    (if (empty? arg2/in)
        (begin
```

```
    (set! =arg1= arg2/in)
    (set! =Stack= Stack/in)
    (Pop-filter/reg))
  (begin
    (set! =arg1= (first arg2/in))
    (set! =Stack= (Push-test-pred arg1/in arg2/in Stack/in))
    (arg1/in)))))
```

Three items of import:

1. Since *Pop-filter* is a function, we must pass its arguments in registers as well. We treat *Push-test-pred* as a primitive.

2. The general problem of determining how many temporary registers we need is a research topic.

3. We need to convert the other functions (*even?/k*, *Pop-filter/k*, &c.) to this register form as well.

This conversion to register form is a mechanical transformation, and so we can make it a phase of our compilation process. We have now transformed the *filter* function into a form that is closer to assembly language; functions can now be safely replaced with labels, and function applications can be replaced with jumps.

We still have *cons* in our program (two different uses), which we need to eliminate to get closer to machine code. This will be the topic of the next few lectures.