Registerizing and Trampolining Scheme Procedures

Adam C. Foltzer

Why bother?

In Scheme, we often take for granted the performance advantages of tail-recursive procedures. Consider the following implementation of the Fibonacci function:

```
> (define fib
    (lambda (n)
      (cond
        [(zero? n) 0]
        [(= n 1) 1]
        [else (+ (fib (- n 1)) (fib (- n 2)))])))
> (fib 5)
5
```

Although this implementation is clean to the eye and easy to understand, a trace reveals how much burden is placed on the stack:

```
> (trace fib)
(fib)
> (fib 5)
|(fib 5)
| (fib 4)
| |(fib 3)
| | (fib 2)
| | |(fib 1)
| | |1
| | |(fib 0)
| | |0
| | 1
| | (fib 1)
| | 1
| |2
| |(fib 2)
| | (fib 1)
| | 1
| | (fib 0)
| | 0
| |1
| 3
| (fib 3)
| |(fib 2)
| | (fib 1)
| | 1
| | (fib 0)
| | 0
| |1
| |(fib 1)
| |1
| 2
|5
5
```

Let's examine a tail-recursive version:

```
> (define fib
    (lambda (n 1-ago 2-ago)
      (cond
        [(zero? n) 2-ago]
        [(zero? (sub1 n)) 1-ago]
        [else (fib (sub1 n) (+ 1-ago 2-ago) 1-ago)])))
> (trace fib)
(fib)
> (fib 5 1 0)
|(fib 5 1 0)
|(fib 4 1 1)
|(fib 3 2 1)
|(fib 2 3 2)
|(fib 1 5 3)
|5
5
```

With tail recursion, the stack stays flat, keeping the memory requirements low and performance high. We had to significantly change our algorithm in order to make it tail-recursive – what if we wanted to (or, in some cases, had to) use the original algorithm? Continuation-passing style allows exactly this:

```
> (define fib
    (lambda (n k)
      (cond
        [(zero? n) (k 0)]
        [(= n 1) (k 1)]
        [else (fib (- n 1)
                (lambda (1-ago)
                  (fib (- n 2)
                    (lambda (2-ago)
                      (k (+ 1-ago 2-ago))))))])))
```

Although the last case looks quite different from the direct-style version, if you unravel the continuations, the algorithm is the same, but all non-simple calls are now in tail position. Let's look at the trace for some evidence:

```
> (define empty-k
    (lambda ()
      (lambda (v) v)))
> (trace fib)
(fib)
> (fib 5 (empty-k))
|(fib 5 #<procedure>)
|(fib 4 #<procedure>)
|(fib 3 #<procedure>)
|(fib 2 #<procedure>)
|(fib 1 #<procedure>)
|(fib 0 #<procedure>)
|(fib 1 #<procedure>)
|(fib 2 #<procedure>)
|(fib 1 #<procedure>)
|(fib 0 #<procedure>)
|(fib 3 #<procedure>)
|(fib 2 #<procedure>)
|(fib 1 #<procedure>)
|(fib 0 #<procedure>)
|(fib 1 #<procedure>)
|5
5
```

The stack is now clearly much flatter than the original version, but has an important similarity – the number of calls to `fib` in both is exactly 15. Still, where does this get us? If our goal is to keep stack usage low, aren't we done by now? Yes, but that's not our entire goal – we want to keep stack usage low while setting the stage for translating our programs into C.

Registerization

Now that our goal includes translation to C, we should start by making our continuations first-order:

```
> (define apply-k
    (lambda (k v)
      (pmatch k
        [(empty-k) v]
        [(big-k ,n ,k)
         (fib (- n 2)
           (little-k v k))]
        [(little-k ,w ,k)
         (apply-k k (+ w v))])))
> (define empty-k
    (lambda ()
      `(empty-k)))
> (define big-k
    (lambda (n k)
      `(big-k ,n ,k)))
> (define little-k
    (lambda (w k)
      `(little-k ,w ,k)))
> (define fib
    (lambda (n k)
      (cond
        [(zero? n) (apply-k k 0)]
        [(= n 1) (apply-k k 1)]
        [else (fib (- n 1) (big-k n k))])))
```

Without any higher-order representations left, this code is beginning to look quite a bit more like C. Registerization pushes this transformation a little further by turning the arguments to the procedure into top-level variables which the procedure modifies in order to do its work.

The first step is to define the arguments to the original procedure as top-level variables, or registers. To avoid naming conflicts, it can be helpful to add a `*` or other marker to the register names:

```
> (define n*)
> (define k*)
```

Next, let's rewrite the simple parts of the procedure to use the new variables:

```
(define fib
  (lambda ()
    (cond
      [(zero? n*) (apply-k k* 0)]
      [(= n* 1) (apply-k k* 1)]
      [else ???])))
```

When registerizing a procedure, most of the thought is involved in replacing the old procedure call. In the `else` case, we had `(fib (- n 1) (big-k n k))` before, so we need to set `n*` to `(- n 1)` and `k*` to `(big-k n k)` before applying `(fib)`. Since we will be using `set!`, we need to be careful about the order we do this, so that we don't overwrite our variables before we're done using their values:

```
(begin (set! n* (- n* 1)) (set! k* (big-k n* k*))) ;big-k gets the new n*
(begin (set! k* (big-k n* k*)) (set! n* (- n* 1))) ;big-k gets the old n*
```

Since we expect both `n`s in `(fib (- n 1) (big-k n k))` to have the same value, we must use the order which preserves that property in the registerized version. Our `else` line therefore is:

```
> (define fib
    (lambda ()
      (cond
        [(zero? n*) (apply-k k* 0)]
        [(= n* 1) (apply-k k* 1)]
        [else (begin
                (set! k* (big-k n* k*))
                (set! n* (- n* 1))
                (fib))])))
```

Next, we give the same treatment to `apply-k`. Since the `k*` we'll need is the same `k*` we already have for `fib`, we only need to add a `v*` register:

```
> (define v*)
> (define apply-k
    (lambda ()
      (pmatch k*
        [(empty-k) v*]
        [(big-k ,n ,k)
         (begin
           (set! n* (- n 2))
           (set! k* (little-k v* k))
           (fib))]
        [(little-k ,w ,k)
         (begin
           (set! v* (+ w v*))
           (set! k* k)
           (apply-k))])))
```

With `apply-k` registerized, we need to adjust the calls to it in `fib`:

```
> (define fib
    (lambda ()
      (cond
        [(zero? n*) (begin (set! v* 0) (apply-k))]
        [(= n* 1) (begin (set! v* 1) (apply-k))]
        [else (begin
                (set! k* (big-k n* k*))
                (set! n* (- n* 1))
                (fib))])))
```

Our procedure now runs correctly if `n*` and `k*` are initialized correctly, but it would be convenient to have a procedure set up the variables for us:

```
> (define fib-driver
    (lambda (n)
      (begin
        (set! n* n)
        (set! k* (empty-k))
        (fib))))
> (fib-driver 5)
5
```

Trampolining

At this point, our `fib` is nearly ready for translation to C. The tail position applications of `(fib)` and `(apply-k)` still would translate into C function calls, which keeps complete stack management just out of reach. Our goal is to eliminate Scheme procedure applications from all of our non-simple[1] code.

The strategy we'll use to remove these calls is to use a new register `pc*` whose value will always be the next procedure to be executed after the current procedure. This change is simple to implement; everywhere in the code we see a tail call such as `(fib)`, we replace it with `(set! pc* fib)`. Here are our new procedures:

```
> (define pc*)
> (define apply-k
    (lambda ()
      (pmatch k*
        [(empty-k) v*]
        [(big-k ,n ,k)
         (begin
           (set! n* (- n 2))
           (set! k* (little-k v* k))
           (set! pc* fib))]
        [(little-k ,w ,k)
         (begin
           (set! v* (+ w v*))
           (set! k* k)
           (set! pc* apply-k))])))
> (define fib
    (lambda ()
      (cond
        [(zero? n*) (begin (set! v* 0) (set! pc* apply-k))]
        [(= n* 1) (begin (set! v* 1) (set! pc* apply-k))]
        [else (begin
                (set! k* (big-k n* k*))
                (set! n* (- n* 1))
                (set! pc* fib))])))
```

[1]Continuation constructors are simple, because they are guaranteed to terminate. Our fib and apply-k cannot make such a guarantee – consider (fib-driver -1)

Of course, we can't do anything useful with our program now without applying `pc*`, so let's introduce a driver procedure, the `trampoline`:

```
> (define trampoline
    (lambda ()
      (begin (pc*) (trampoline))))
```

`trampoline` applies the contents of `pc*`, and then calls itself. If you're thinking that this looks like an infinite loop, you're correct. We have to build ourselves a way out of the trampoline's loop once we have our answer.

In Scheme, there are several ways to accomplish this, notably by using `call/cc` before starting the trampoline in order to later exit the loop by the continuation it provides. Another way that can be more directly mirrored in C is to use yet another register to tell the trampoline when the program is finished, and then grab the result from the `v*` register:

```
> (define done*)
> (define apply-k
    (lambda ()
      (pmatch k*
        [(empty-k) (set! done* #t)]
        [(big-k ,n ,k)
         (begin
           (set! n* (- n 2))
           (set! k* (little-k v* k))
           (set! pc* fib))]
        [(little-k ,w ,k)
         (begin
           (set! v* (+ w v*))
           (set! k* k)
           (set! pc* apply-k))])))
> (define trampoline
    (lambda ()
      (if done*
          v*
          (begin (pc*) (trampoline)))))
```

```
> (define fib-driver
    (lambda (n)
      (begin
        (set! n* n)
        (set! k* (empty-k))
        (set! done* #f)
        (set! pc* fib)
        (trampoline))))
```

At last, we're finished! Our program is now structured such that they can be adapted to *ParentheC/PC*[2], and then finally to C.

[2]http://www.cs.indiana.edu/classes/c311/ParentheC/reg/ParentheC.pdf