Deriving Z and Y

C311

January 30, 2008

We begin with an expression for *!* using self-application.

```
((let ((! (λ (!)
             (λ (n)
               (if (zero? n) 1 (* n ((! !) (- n 1))))))))
   (! !))
 5)
```

In order to remove the self-application from the body of the (λ (*!*) ...), we add another **let**.

```
((let ((! (λ (!)
            (let ((! (! !)))
              (λ (n)
               (if (zero? n) 1 (* n (! (- n 1))))))))))
   (! !))
 5)
```

But in Scheme, this expression results in an infinte loop. To avoid this divergence, we perform an inverse-η (eta) step on the right-hand side of the **let** we have introduced. The inverse-η of an expression M is (λ (n) (M n)), provided n does not occur free in M.

```
((let ((! (λ (!)
            (let ((! (λ (n) ((! !) n))))
              (λ (n)
                (if (zero? n) 1 (* n (! (- n 1))))))))))
   (! !))
 5)
```

This expression avoids the infinite loop, and once again returns 120. To simplify our expression, we lift the "good stuff" (beginning with (λ (n) ...)) out of the expression, and call it f.

```
(define f
  (λ (n)
    (if (zero? n) 1 (* n (! (- n 1))))))
```

We notice, however, that this definition contains a free variable, *!*. To fix this problem, we abstract over *!*.

```
(define f
  (λ (!)
    (λ (n)
      (if (zero? n) 1 (* n (! (− n 1)))))))
```

Next, we modify our expression to use f. Since we added a (λ (*!*) ...) to f, we must pass *!* to it.

```
((let ((! (λ (!)
            (let ((! (λ (n) ((! !) n))))
              (f !)))))
   (! !))
 5)
```

Now, we replace the reference to *!* with its right-hand side expression and eliminate the **let**.

```
((let ((! (λ (!)
            (f (λ (n) ((! !) n))))))
   (! !))
 5)
```

Once again, we can replace references to the **let**-bound *!* by its right-hand side expression, eliminating the **let**.

```
(((λ (!) (f (λ (n) ((! !) n))))
  (λ (!) (f (λ (n) ((! !) n)))))
 5)
```

We can lift the operator part out of this expression, yielding a definition such that (*almostZ* 5) returns 120.

```
(define almostZ
  ((λ (!) (f (λ (n) ((! !) n))))
   (λ (!) (f (λ (n) ((! !) n))))))
```

The function f is hard-coded in this definition, however. We would like a general definition for Z, such that ((Z f) 5) returns 120. We simply abstract over f.

```
(define Z
  (λ (f)
    ((λ (!) (f (λ (n) ((! !) n))))
     (λ (!) (f (λ (n) ((! !) n)))))))
```

All that remains is to rename our variables to the canonical names used in the literature.

```
(define Z
  (λ (f)
    ((λ (x) (f (λ (y) ((x x) y))))
     (λ (x) (f (λ (y) ((x x) y)))))))
```

Interestingly, this definition still contains two subexpressions of the form $(\lambda\ (n)\ (M\ n))$, where n does not occur free in M. If we perform an η-step on these subexpressions, we arrive at the definition of Y.

```
(define Y
  (λ (f)
    ((λ (x) (f (x x)))
     (λ (x) (f (x x))))))
```

Notice that while $((Z\ f)\ 5)$ returns 120, $((Y\ f)\ 5)$ goes into an infinte loop in Scheme. Just as we needed inverse-η to avoid an infinte loop in the **let** we introduced earlier, the inverse-η in Z is required in applicative order (call-by-value) Scheme. In normal order (call-by-name) λ-calculus, the definition of Y suffices. In call-by-value languages, then, there is one more constraint on the application of the η rule to an expression $(\lambda\ (n)\ (M\ n))$: if M may not terminate, the rule cannot be applied.

We can use the definition of Z to run factorial in our interpreter, despite the fact that it does not support **define**. We simply copy and paste the definitions of Z and f into the call $((Z\ f)\ 5)$:

```
(((λ (f)
    ((λ (x) (f (λ (y) ((x x) y))))
     (λ (x) (f (λ (y) ((x x) y))))))
  (λ (!)
    (λ (n)
      (if (zero? n)
          1
          (* n (! (sub1 n)))))))
 5)
```

This expression should run in your interpreter, and evaluate to 120.