What's in a Template?

Hazel Levine and Adrian German

January 2021

Abstract

First-class, higher-order functions can become their own accumulators, thereby blurring the conceptual separation between the generative recursion template and the template used for accumulator-passing style.

Contents

1 Introduction

1.1 Reversing a List of Numbers

Consider the following generatively[1] recursive procedure in ISL+ that receives a list of numbers and returns its copy in reverse:

[1] Actually, the function presented (named `backwards`) is NOT generative. It is structural because the functional recursion follows the data-shape recursion form, as in "function follows form." And although structural recursion actually IS a type of generative recursion, the distinction is certainly not optional—since it may be possible to show that all APS structural functions can be written without accumulator (which is not true for the non-structural generative recursion, as shown in HtDP).

```
; [ListOf Number] is one of:
;   -- empty
;   -- (cons Number [ListOf Number]
; backwards : [ListOf Number] -> [ListOf Number]
; purpose statement: to reverse a list of numbers
(define (backwards lon)
  (cond ((empty? lon) lon)
        (else (append (backwards (rest lon)) (list (first lon))))))
(check-expect (backwards '(1 2 3)) '(3 2 1))
```

Functions written in APS avoid redundant actions by remembering[2] their intermediate results. We want to find a generative definition for `backwards` that is also fast (i.e., $O(n)$ as opposed to $O(n^2)$).

1.2 Basic Profiling

This how much slower our function runs compared with the equivalent primitive:

```
(time
 (let ((a (reverse (build-list 100000 add1))))
   'done)) ; cpu time: 16 real time: 27 gc time: 16

(time
 (let ((a (backwards (build-list 100000 add1))))
   'done)) ; cpu time: 205953 real time: 217258 gc time: 120334
```

2 Computation in Little Pieces

2.1 One-Step Computations

This is the simplest possible computation step:

```
(lambda (x) x)
```

We can hint at what it means with this template[3]:

```
(let ((x ...))
  x)
```

[2] It's worth being careful here with what we exactly we are trying to say as "accumulator-style does NOT necessarily reduce complexity, especially time-complexity. Such generalizations are dangerous. If you're unlucky, you get different results—see 'Janus' in HtDP—and you may merely save space not time at all. For the latter, think big-num arithmetic."

[3] This is not a template in the sense of HtDP.

Only when snippets are in action can we write actual code to show equivalence:

```
(check-expect
    ( (lambda (x) x)  5)
     (let ((x  5  )) x) )
```

2.2 Composing Monads[4]

Now consider the following sequence of named steps:

```
(check-expect
 (let ((f1 (lambda (v) v)))
   (let ((f2 (lambda (v) (cons 3 (f1 v)))))
     (let ((f3 (lambda (v) (cons 5 (f2 v)))))
       (let ((f4 (lambda (v) (cons 1 (f3 v)))))
         (f4 null)))))
 '(1 5 3))
```

It is entirely equivalent to:

```
(check-expect ((lambda (v)                          ; this is f4
                 (cons 1 ((lambda (v)               ; this is f3
                            (cons 5 ((lambda (v)    ; this is f2
                                       (cons 3 ((lambda (v) ; f1
                                                  v)
                                                v)))
                                     v)))
                          v)))
               null)
              '(1 5 3))
```

2.3 Hand Tracing Code

To explore/internalize the equivalence we start from the[5] simplest case:

```
(check-expect (let ((f1 (lambda (v) v)))
                (f1 null))
              '())
```

[4] Maybe a useful read: `http://pinouchon.github.io/perception/cognition/ai/2016/03/30/monads-jokes-and-analogies.html`, or `https://www.youtube.com/watch?v=yjmKMhJOJos`

[5] Why not use the rules of the book to actually calculate here? The point of keeping the teaching languages functional up to ISL+ is so that pre-algebra explains all computation and beta is just a generalization of pre-algebra.

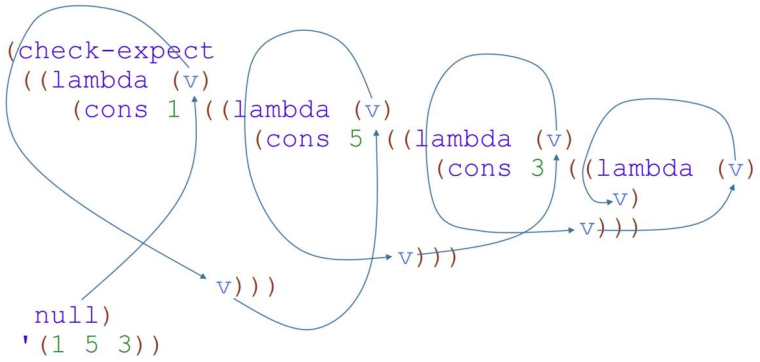

Figure 1: Tracing the input through four chained distinct computation steps.

We then gradually consider cases of increased complexity:

```
(check-expect
 (let ((f1 (lambda (v) v)))
   (let ((f2 (lambda (v) (cons 3 (f1 v)))))
     (f2 null)))
 '(3))
```

Here's one more intermediate step in our example:

```
(check-expect
 (let ((f1 (lambda (v) v)))
   (let ((f2 (lambda (v) (cons 3 (f1 v)))))
     (let ((f3 (lambda (v) (cons 5 (f2 v)))))
       (f3 null))))
 '(5 3))
```

Eventually we can trace the computation as shown in Figure 1.

We now use this approach to split the computation in individual steps and chain them into a resulting function.

3 Growing a Function

3.1 It's a Two-Step Process

The function defined below takes a list of numbers and a one-step computation; it produces a computation. The result built by `calculate` can be considered an accumulator. It may be instructive to try to write signatures for all packaged

computations (that is, functions, both named and/or anonymous) in the definition[6]. In the process one can come up with an invariant for the accumulator we build and eventually a proof that `calculate` does what it's supposed to do.

Here's the code:

```
; calculate : [ListOf Number] [W -> X] -> [Y -> Z]
; returns a computation based on fun that is one step longer
(define (calculate lon fun)
  (cond ((empty? lon) fun)
        (else (calculate (rest lon)
                         ; ... : [M -> N]
                         (lambda (v) ; this is how we freeze and package computations!
                           (cons (first lon)
                                 (fun v)))))))
(check-expect
 ((calculate '(1 2 3) (lambda (v) v)) empty)
 '(3 2 1))
```

So we build a computation, in stages, via the standard generative recursion process. We then apply it (outside of this process of construction) at the end, via the very basic functional (de-)composition template.

```
(time
 (let ((a ((calculate
            (build-list 100000 add1)
            (lambda (v) v)) empty)))
   'done)) ; cpu time: 359 real time: 369 gc time: 157
```

3.2 Once More, With Feeling

Here's the step that builds the computation:

```
; rev : [ListOf Number] -> [ [ListOf Number] -> [ListOf Number] ]
;             a                    b                  c
; rev returns a function that builds c = (append (reverse a) b)
(define (rev lon)
  (cond ((empty? lon) (lambda (v) v))
        (else (let ((future (rev (rest lon)))) ; future is a function
                (lambda (v)  ; package it, as shown in the previous section
                  (future ((lambda (v)
                             (cons (first lon) v))
                           v)))))))
```

[6]To facilitate this exercise I introduced the generic types `W`, `X`, `Y`, `Z`, `M` and `N`.

```
(check-expect ((rev (list 1 2 3 4 5 6)) '(10 11 12))
              '(6 5 4 3 2 1 10 11 12))

; if b is empty c is the reverse of a
(check-expect ((rev (list 1 2 3 4 5 6)) empty)
              '(6 5 4 3 2 1))
```

We can eliminate the reference to the `future` as follows:

```
; omgekeerde: reverse (in Dutch)
; omgekeerde is the same as rev defined earlier
(define (omgekeerde lon)
  (cond ((empty? lon)
         (lambda (v) v)) ; the empty computation
        (else (lambda (v)
                ((omgekeerde (rest lon)) ; apply the future (now in little pieces)
                 ((lambda (v) ; to the current, packaged, one-step computation
                    (cons (first lon)
                          v))
                  v))))))

(check-expect ((omgekeerde (list 1 2 3 4 5 6)) empty)
              '(6 5 4 3 2 1))
```

This is the code that Mardin and I proposed[7] on Nov. 18, 2014.

4 Conclusion

4.1 What's in a Template?

We can now eliminate any references to free[8] variables in our code:

```
; umkehren: reverse (in German)
; umkehren is omgekeerde written without any free variables
(define umkehren (lambda (this lon)
                   (cond ((empty? lon)
                          (lambda (v) v))
                         (else (let ((future (this this (rest lon))))
                                 (lambda (v)
                                   (future ((lambda (v)
                                              (cons (first lon) v)) v)))))))))
```

[7] See Exercise 491 in `https://htdp.org/2018-01-06/Book/part_six.html`

[8] Essentially eliminating any and all `letrec` behavior in the code.

Code is written as a poor man's (i.e., inlined) version of Turing's Y combinator[9]:

$$\Theta = (\lambda xy.y(xxy))(\lambda xy.y(xxy))$$

Here's how we test and profile it:

```
(check-expect
 ((umkehren umkehren (list 2 4 6 8)) empty)
 '(8 6 4 2))

(time
 (let ((a (umkehren umkehren (build-list 100000 add1))))
   'done)) ; cpu time: 219 real time: 218 gc time: 45
```

It is immediate that $\Theta f \mapsto f(\Theta f)$ and very cleanly so.

Let's start with some notation:

$$\tau = (\lambda xy.y(xxy))$$

Then

$$\Theta = (\tau\tau)$$

So now

$$\begin{aligned}\Theta f = (\tau\tau)f = ((\lambda xy.y(xxy))\tau)f = \\ (\lambda y.y(\tau\tau y))f = \\ f((\tau\tau)f) = f(\Theta f)\end{aligned}$$

We conclude that in the presence of anonymous and higher-order functions in the ISL+ language, a `lambda` can effectively become a self-accumulator, thus obfuscating the provided template. This is good, as it further demonstrates the power of abstraction, and opens up paths to explore additional advanced topics.

[9] `https://en.wikipedia.org/wiki/Fixed-point_combinator`