UROC: What's in a Template?

Hazel Levine, mentor Dan-Adrian German

Luddy School of Informatics, Computing, and Engineering — Indiana University Bloomington

Introduction

This project presents a counterexample to the introduction of accumulator-passing style as shown in *How to Design Programs*. We note that in the presence of lambdas and higher-order functions in the ISL+ language, a lambda can effectively become a self-accumulator, thus obfuscating the provided definition. To accomplish this, we use the fix-point of a non-recursive function that builds a lambda. Because of the meaning of a fix-point in the lambda calculus, this is a recursive function that uses itself as an accumulator.

Summary

To construct this counterexample, we construct a program in ISL+ utilizing a naive, non-combinator implementation of the fix point. This implementation is equivalent to something akin to the Y combinator:

$$\text{lambda } f.(\text{lambda } x.f(xx))(\text{lambda } x.f(xx))$$

This implementation uses the fix point to calculate a function that when applied, reverses a list. A non-accumulator implementation of a reverse function typically uses up $O(n^2)$ time, however our implementation runs in $O(n)$. Our implementation uses up linear space, however, as opposed to the traditional accumulator-based solution using constant space.

Implementation

We define a function `fix` taking two arguments, a function and an argument to apply to it. Per the definition of a fix-point, this function continually applies itself to its result until the output converges. We also define a data structure `Pair` that contains two arguments to work with `fix`.

We then define a function `rev-step` that takes a `Pair` containing a list and a function that takes a list and returns a list. This function follows the structural recursion template for list processing, and via a somewhat roundabout way, the template for processing a `Pair`. However, this function is not recursive. Upon application to a non-empty list, it returns a pair of the rest of the list and a function that takes *the result of future computation* and `cons`es the element of the list to it.

Finally, we define a function `rev` that calculates the fix point of `rev-step` and applies its result to the empty list. This results in a fully functional reverse function.

Code

```
(define (fix f xs)
  (let ([res (f xs)])
    (if (equal? xs res)
        res
        (fix f res))))

(define-struct pair [fst snd])

; rev-step : [Pair [Listof A] [[Listof A] -> [Listof A]]
;               -> [Pair [Listof A] [[Listof A] -> [Listof A]]
(define (rev-step a)
  (let ([lst (pair-fst a)]  ; the list to work with
        [fun (pair-snd a)]) ; the function for the next computation
    (cond [(empty? lst) (make-pair lst fun)]
          [else (make-pair (rest lst)
                           (lambda (res) ; res is the result of future computation
                             (cons (first lst) (fun res))))])))

; rev : [Listof A] -> [Listof A]
(define (rev lst)
  ((pair-snd (fix rev-step (make-pair lst identity))) empty))
```