[SPOILERS] Practice Problems for Section 6 -- Solutions

Pavel Lepin COMMUNITY TA · a month ago

```
#lang racket

(provide (all-defined-out))

;;; The Usual Suspects ;;;

;; provided definitions
(struct btree-leaf () #:transparent)
(struct btree-node (value left right) #:transparent)

(define (btree-fold f init t)
  (if (btree-leaf? t)
      init
      (f (btree-fold f init (btree-node-left t)) (btree-node-value t)
         (btree-fold f init (btree-node-right t)))))

(define (btree-unfold f state)
  (define res (f state))
  (if res
      (btree-node (car res)
                  (btree-unfold f (cadr res))
                  (btree-unfold f (cddr res)))
      (btree-leaf)))

(define (gardener t)
  (btree-unfold
   (lambda (t)
     (if (or (btree-leaf? t) (not (btree-node-value t)))
         #f
         (cons (btree-node-value t)
               (cons (btree-node-left t)
                     (btree-node-right t)))))
   t))

;;; So Dynamic ;;;

;; Crazy Sum ;;
(define (crazy-sum xs)
  (cdr
   (foldl
    (lambda (x st)
      (if (procedure? x)
          (cons x (cdr st))
          (cons (car st) ((car st) (cdr st) x))))
    (cons + 0)
    xs)))

;; Universal Fold ;;

(define (universal-fold f init data)
  (if (list? data)
      (foldr f init data)
      (btree-fold f init data)))

(define (universal-sum data)
  (universal-fold + 0 data))

;; Stomp! ;;
(define (flatten xs)
  (if (empty? xs)
      null
      (if (list? (car xs))
          (if (empty? (car xs))
              (flatten (cdr xs))
              (flatten (cons (caar xs) (cons (cdar xs) (cdr xs)))))
          (cons (car xs) (flatten (cdr xs))))))

;;; Lambda Madness ;;;

;; provided definitions
;; MUPL structs from Section 5 homework assignment in Programming Languages class
(struct var  (string) #:transparent)  ;; a variable, e.g., (var "foo")
(struct int  (num)    #:transparent)  ;; a constant number, e.g., (int 17)
(struct add  (e1 e2)  #:transparent)  ;; add two expressions
(struct ifgreater (e1 e2 e3 e4)    #:transparent) ;; if e1 > e2 then e3 else e4
(struct fun  (nameopt formal body) #:transparent) ;; a recursive(?) 1-argument functio
```

```
n
(struct call (funexp actual)       #:transparent) ;; function call
(struct mlet (var e body) #:transparent) ;; a local binding (let var = e in body)
(struct apair (e1 e2)     #:transparent) ;; make a new pair
(struct fst  (e)    #:transparent) ;; get first part of a pair
(struct snd  (e)    #:transparent) ;; get second part of a pair
(struct aunit ()    #:transparent) ;; unit value -- good for ending a list
(struct isaunit (e) #:transparent) ;; evaluate to 1 if e is unit else 0

(define (simplify e)
  (cond [(mlet? e)
         (call (fun #f (mlet-var e) (simplify (mlet-body e))) (simplify (mlet-e e)))]
        [(apair? e)
         (fun #f "_f" (call (call (var "_f") (simplify (apair-e1 e))) (simplify (apair
-e2 e))))]
        [(fst? e)
         (call (simplify (fst-e e)) (fun #f "_x" (fun #f "_y" (var "_x"))))]
        [(snd? e)
         (call (simplify (snd-e e)) (fun #f "_x" (fun #f "_y" (var "_y"))))]
        [(add? e)
         (add (simplify (add-e1 e)) (simplify (add-e2 e)))]
        [(ifgreater? e)
         (ifgreater (simplify (ifgreater-e1 e))
                    (simplify (ifgreater-e2 e))
                    (simplify (ifgreater-e3 e))
                    (simplify (ifgreater-e4 e)))]
        [(fun? e)
         (fun (fun-nameopt e) (fun-formal e)
              (simplify (fun-body e)))]
        [(call? e)
         (call (simplify (call-funexp e))
               (simplify (call-actual e)))]
        [(isaunit? e)
         (isaunit (simplify (isaunit-e e)))]
        [else e]))
```

0 · flag

Nikolai Saeverud · a month ago

Pavel,
Your solution to crazy-sum is very elegant. I like the way you use foldl. I tend to solve problems with solutions that are similar to what I have done before, but then to see your solution being quite different, I get inspired to learn new things. So thank you. Just thought to post my solution as well. If you see anything that can be improved, don't hesitate to give me feedback.

```
(define (crazy-sum xs)
  (define (helper f xs acc)
    (cond [(null? xs) acc]
          [(procedure? (car xs)) (helper (car xs) (cdr xs) acc)]
          [#t (let ([acc (f acc (car xs))])
            (helper f (cdr xs) acc))]))
  (helper + xs 0))
```

0 · flag

Pavel Lepin COMMUNITY TA · a month ago

I should note that the "elegance" stems mainly from the fact that I came up with a somewhat contrived example of non-trivial stuff we can do with heterogeneous lists thanks to dynamic typing, then wrote a problem statement to match the code I already had.

You solution is perfectly fine, but I do recommend remembering that `helper`'s recursion pattern over a list can *always* be abstracted by a left fold. It's not a significant win in terms of code complexity or anything like that, but a left fold is a clear indicator of what we're doing here, and that essentially hides the boilerplate from whoever's gonna read the code.

0 · flag

Nikolai Saeverud · a month ago

Yes, I recognized that my helper function is a kind of homemade foldl. I am kind of slowly realizing that fold map and filter are very universally known among programmers, so why not use them, so that it can be known instantly what is going on..... Anyway, I should stop practicing now, and get on with this week's assignment, due in a couple of days. :-)

0 · flag

+ Comment

[✓] Subscribe to this thread at the same time

Add post