Assignment 7: Continuations and Representation Independence

So, you're telling us that the universe is written in continuation-passing style?

—Phil Wadler, sometime during ICFP 2004

Note

As you proceed with this assignment, you may find the following resources **extraordinarily** helpful.

- Notes on making continuations representation-independent methodically.

Assignment

- We provide a small test suite a7-student-tests for Part 2; to test Part 1 you should modify the calls provided in this assignment to be invocations of your CPSed implementations.

Part I: Recap

1. Define and test a procedure `binary-to-decimal-cps` that is a CPSed version of the following `binary-to-decimal` procedure:

```
(define binary-to-decimal
  (lambda (n)
    (cond
      [(null? n) 0]
      [else (+ (car n) (* 2 (binary-to-decimal (cdr n))))])))
```

`binary-to-decimal` uses little-endian binary numbers; you should binary sequences with one or more trailing `0`s to be ill-formed binary numbers (bad data). Here are a few sample calls to make the meaning clear.

```
> (binary-to-decimal '())
0
> (binary-to-decimal '(1))
1
> (binary-to-decimal '(0 1))
2
> (binary-to-decimal '(1 1 0 1))
11
```

2. Define and test a procedure `rember*1-cps` that is a CPSed version of the following `rember*1` procedure, which removes the first `?` in the arbitrarily nested list `ls`:

```
(define rember*1
  (lambda (ls)
    (cond
      [(null? ls) '()]
      [(pair? (car ls))
       (cond
         [(equal? (car ls) (rember*1 (car ls)))
          (cons (car ls) (rember*1 (cdr ls)))]
         [else (cons (rember*1 (car ls)) (cdr ls))])]
      [(eqv? (car ls) '?) (cdr ls)]
      [else (cons (car ls) (rember*1 (cdr ls)))])))
```

Part II: The interpreter

CPS your interpreter. After CPSing, create two copies of your interpreter that are **representation-independent with respect to continuations**, using **functional representation** for one copy, and **data-structural representation** for the other.

You will also need to implement the helpers for environments and procedures, using a representation of your choice. Feel free to share the environment and procedure helpers among all the interpreters. **Do not CPS these helpers**.

In all, you should implement three interpreters. Name your interpreters as follows:

Description	Name
CPSed interpreter that is representation-**dependent** WRT continuations	`value-of-cps`
CPSed interpreter that is representation-**independent** WRT continuations, and uses functional	`value-of-`

continuation helpers	`cps-fn`
CPSed interpreter that is representation-**independent** WRT continuations, and uses data-structural continuation helpers	`value-of-cps-ds`

Scheme's `call/cc` **may not** be used in any of your interpreters.

The code for `value-of-cps-fn` and `value-of-cps-ds` will be identical, except that they will call different sets of continuation helpers. When you define empty continuation constructors for `value-of-cps-fn` and `value-of-cps-ds`, name them `empty-k-fn` and `empty-k-ds`, respectively. This will make it possible for us to test and grade your code.

After your interpreter has been CPSed and you're ready to move on to the next phase of the assignment, you may want to make use of this method for making continuations representation-independent. (There are many valid ways to make continuations representation-independent, and it's up to you whether to use this technique or not.)

Start with this interpreter:

```
(define value-of
  (lambda (expr env)
    (pmatch expr
      [`,n (guard (or (number? n) (boolean? n))) n]
      [`(+ ,x1 ,x2) (+ (value-of x1 env) (value-of x2 env))]
      [`(* ,x1 ,x2) (* (value-of x1 env) (value-of x2 env))]
      [`(sub1 ,x) (sub1 (value-of x env))]
      [`(zero? ,x) (zero? (value-of x env))]
      [`(if ,test ,conseq ,alt) (if (value-of test env)
                                    (value-of conseq env)
                                    (value-of alt env))]
      [`(capture ,k-id ,body) (call/cc (lambda (k)
                                         (value-of body (extend-env k-id k env))))]
      [`(return ,v-exp ,k-exp) ((value-of k-exp env) (value-of v-exp env))]
      [`,x (guard (symbol? x)) (apply-env env x)]
      [`(lambda (,id) ,body) (closure id body env)]
      [`(,rator ,rand) (apply-closure (value-of rator env) (value-of rand env))])))
```

Here are a few test programs that work with every version of the interpreter. Remember to test your interpreter after each step in the transformation!

```
> (define fact-5
    '((lambda (f)
 ((f f) 5))
      (lambda (f)
 (lambda (n)
   (if (zero? n)
       1
       (* n ((f f) (sub1 n))))))))
> (define capture-fun
    '(* 3 (capture q (* 2 (return 4 q)))))
> (value-of-cps fact-5 (empty-env) (empty-k))
120
> (value-of-cps capture-fun (empty-env) (empty-k))
12
> (value-of-cps-fn fact-5 (empty-env) (empty-k-fn))
120
> (value-of-cps-fn capture-fun (empty-env) (empty-k-fn))
12
> (value-of-cps-ds fact-5 (empty-env) (empty-k-ds))
120
> (value-of-cps-ds capture-fun (empty-env) (empty-k-ds))
12
```

Place all of your code in a file named `a7.rkt` and submit it via Oncourse.

Brainteaser

For the brainteaser this week, you'll get to learn about `streams`, a data-structure that enables us to process infinite lists of items. Its a lazily-evaluated, memoized (also termed *delayed*) list.

To play around, we'll first need to implement a few tools.

```
(define-syntax cons$
  (syntax-rules ()
    ((cons$ x y) (cons x (delay y)))))

(define car$ car)

(define cdr$
  (lambda ($) (force (cdr $))))
```

We'll get back to those helpers. For now, its enough that they're tweaked `cons`, `car`, and `cdr`. The first question to ask is: how do you build an infinite list? The only reasonable answer is: one item at a time, as needed. Here, we're going to define an infinite stream of ones.

```
(define inf-1s (cons$ 1 inf-1s))
```

It looks like that can't possibly work. We're definining the stream in terms of itself. That's a circular definition. But, in fact, that's precisely what we're after. We're defining a list that has a 1 in front and whose cdr - well, whatever that thing is, it has a 1 in the front of it. And thus its 1s all the way down.

So we can build a procedure `take$`

```
(define take$
  (lambda (n $)
    (cond
      ((zero? n) '())
      (else (cons (car$ $) (take$ (sub1 n) (cdr$ $)))))))
```

that pulls n items from the stream, and returns them in a list (the $ stands for $tream, by the way).

```
> (take$ 5 inf-1s)
(1 1 1 1 1)
> (take$ 10 inf-1s)
(1 1 1 1 1 1 1 1 1 1)
```

So how is this all working? There's nothing to `car$`. That's just `car` with fancy window-dressing. `cons$` is the first macro definition we've looked at in class. But there's nothing much to it, either. You can think of it as performing a textual transformation – every time we see something of the form (`cons$ <thing1> <thing2>`), that code is actually transformed as (`cons <thing1>` (`delay <thing2>`)) Importantly, `<thing2>` isn't evaluated in the process. But do take note of the `delay` form, and the `force` form in `cdr$`. As we've already seen, there are times in which we might like to evaluate an expression only once, and thereafter just return that already computed value. And we might like to hold off on doing that evaluation, instead of doing it right away. `delay` does both of those – it creates a *promise*, of which `force` can then force the evaluation. From there on out, every time we force that promise, we get the same value.

```
> (define worst-random
    (delay (random 4)))
> (force worst-random)
2
> (force worst-random)
2
> (force worst-random)
2
```

So, to put it all together, we define `inf-1s` to be a pair whose `car` is 1 and whose `cdr` is a promise. When we finally get around to evaluating that promise, we find that its value is in fact `inf-1s` – that is, a pair whose `car` is 1 and whose `cdr` is a promise.

Hopefully that all makes sense. Your task this week is to implement the tribonacci stream. Its the sequence from the exam: 0 is the first tribonacci number, 1 is the second, 1 is the third, and the values thereafter are each the sum of the three previous values in the sequence. Call it `trib$`.

```
> (car$ trib$)
0
> (car$ (cdr$ trib$))
1
> (take$ 7 trib$)
(0 1 1 2 4 7 13)
```

Just Dessert

By now, you probably have a pretty strong intuition as to the mechanical process by which you CPS programs. As mentioned in class, it is possible to write a program that automatically performs CPS transformations. Write a procedure that takes an expression and returns a CPSed version of that expression. You may find it exceedingly helpful to consult Danvy and Nielsen's "A First-Order One-Pass CPS Transformation", specifically Figure 2 on page 244 and the surrounding discussion. Take note of the four specific cases in which they treat applications.

This is a little more open-ended than most of our problems; you can get it to work on the lambda-calculus, or add more forms if you want (this will probably make it easier to test and to use it). Add a comment in your assignment to tell us how to call and to use your CPSer.

assignment-7.txt · Last modified: 2014/10/09 21:03 by jhemann