Login

Assignment 7: Continuations and Representation Independence

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

Note

As you proceed with this assignment, you may find the following resources helpful:

- New and old notes on making continuations representation-independent methodically.

Assignment

- We also provide a test suite: a7-student-tests

Part I: let/cc

For part I, you should *not* CPS anything or use `empty-k`.

1. Complete the following definition of `last-non-zero`, a function which takes a list of numbers and return the last cdr whose car is 0. In other words, when starting from the right of the list, it should be all numbers before the first 0 is reached. See the test cases below and student test file for examples. Your solution should be naturally recursive, and should not contain any calls to member-like operations, nor should you be reversing the list.

```
(define last-non-zero
  (lambda (ls)
    (let/cc k
      (letrec
        ((last-non-zero
           (lambda (ls)
             (cond
               ;; fill in lines here
               ))))
        (last-non-zero ls)))))
```

```
> (last-non-zero '(0))
()
> (last-non-zero '(1 2 3 0 4 5))
(4 5)
> (last-non-zero '(1 0 2 3 0 4 5))
(4 5)
> (last-non-zero '(1 2 3 4 5))
(1 2 3 4 5)
>
```

Advice on this problem: first, get it to work for the base case. Then, get it to work for a list with no `0`s in it. Then, get it to work for a list with one `0` in it. Then, get it to work for a list with more than one `0` in it. And you're going to be using that continuation `k` somewhere in your definition.

Part II: lex

1. Just when you thought you'd seen the last of it—`lex` is back. If you've got the previous version working, then all we're asking for is a handful of changes.

1. Replace your application line with the following `[`(,rator ,rand) `(app ,(lex rator acc) ,(lex rand acc))]`
2. Replace your `zero?` line with the following `[`(zero? ,nexp) `(zero ,(lex nexp acc))]`
3. Replace your `*` line with the following `[`(* ,nexp1 ,nexp2) `(mult ,(lex nexp1 acc) ,(lex nexp2 acc))]`
4. Add lines to `lex` that implement `letcc` (a lexed form of `let/cc`) and `throw`. Of these, `letcc` is the more interesting one.

These first three changes aren't required to correctly perform lexical addressing, but will make your `lex` useful in part III.

Part III: The interpreter

This part of the assignment will be completed in several stages. You should do each of these steps in a separate file. Turn in a file containing the last step, and call it a7.rkt. But it is **imperative** that you keep each of the previous steps. We will ask to see them at some point in the future. Again, it is **imperative** that you keep previous steps.

You should begin with the interpreter below. This is an interpreter for a lexed language, like the interpreter of problem 6 on assignment 3.

```
(define value-of
  (lambda (expr env)
    (match expr
      [`(const ,expr) expr]
      [`(mult ,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))]
      [`(letcc ,body) (let/cc k
                        (value-of body (lambda (y) (if (zero? y) k (env (sub1 y))))))]
      [`(throw ,k-exp ,v-exp) ((value-of k-exp env) (value-of v-exp env))]
      [`(let ,e ,body) (let ((a (value-of e env)))
                         (value-of body (lambda (y) (if (zero? y) a (env (sub1 y))))))]
      [`(var ,y) (env y)]
      [`(lambda ,body) (lambda (a) (value-of body (lambda (y) (if (zero? y) a (env (sub1 y))))))]
      [`(app ,rator ,rand) ((value-of rator env) (value-of rand env))])))

(define empty-env
  (lambda ()
    (lambda (y)
      (error 'value-of "unbound identifier"))))

(define empty-k
  (lambda ()
    (lambda (v)
      v)))
```

You should take `value-of` as well as the definitions of `empty-env` and `empty-k`, which will prove useful as you begin your transformations. When you write your own test programs, you will probably prefer not to write them directly in the language of this interpreter. You can write test programs in Racket, use your `lex` to transform them to the

language of our interpreter, and the results in `value-of-cps`. Use `empty-env` and `empty-k` when testing.

Steps for Part III

1. CPS this interpreter. Call it `value-of-cps`. Use the "let trick" to eliminate the let binding when you CPS the `let` line. You might consider always using `k^` as the additional continuation variable to your extended environments. Do not apply `k^` to the call to `error` in `empty-env`. This is similar to the behavior of `times-cps-shortcut` from Assignment 6. Racket's `let/cc` **may not** be used in your CPSed interpreter. You might consider comment out some of your match clauses, and CPSing the interpreter a few lines at a time. But try to finish this step entirely before you move on to the next one. Since your closures and environments are both implemented as functions, you should consider renaming them to `env-cps` and `c-cps`, respectively.
2. Define `apply-env`, `apply-closure`, and `apply-k`, and add all the calls to those functions. Notice that after CPSing, your `apply-closure` and `apply-env` now take three arguments.
3. Define `extend-env` and replace all 3 of your explicitly higher-order representations of environments with calls to `extend-env`.
4. Add a `^` to each of the formal parameters of `extend-env` (not the inner function, mind you). Ensure that the inner functions in both `extend-env` and `empty-env` use the same formal parameters as the second argument to `apply-env` (here, typically `y` and `k^`).
5. Replace your higher-order function representation of environments with a tagged-list data structure representation. Consider first ensuring that the last two formal parameters to `apply-env` (`y` and `k`) are the same as the formal parameters to the inner functions in `extend-env` and `apply-env`. Remember, if you add `(else (env-cps y k))` as the last line of your `match` expression, you can test each transformation one at a time.
6. If you used it, remove the `(else (env-cps y k))` as the last line of your `match` expression in `apply-env`.
7. Define `make-closure` and replace your explicitly higher-order representation of a closure with a call to `make-closure`.
8. Replace your higher-order function representation of closures with a tagged-list data structure representation. Consider first ensuring that the last two formal parameters to `apply-closure` (`a` and `k`) are the same as the formal parameters to the inner function in `make-closure`.
9. Define constructors for your continuations, and replace your explicitly higher-order function continuations with calls to these constructors. For nested continuations, you should consider working from the inside out. Remember, if you transform your constructors one at a time, you can test your program after each replacement.
10. Add a `^` to each of the formal parameters of your continuation helpers, and change the body to correspond with these changed variable names (that is, do an alpha substitution). Then, ensure that the inner function in each of your continuation helpers uses the same formal parameter as the second argument to `apply-k` (typically, `v`).
11. Replace your higher-order function representations of continuations with a tagged-list data structure representation. Remember, if you add `(else (k v))` as the last line of your match, you can test each transformation one at a time. If you're good, you can do almost all of this step with a keyboard macro.
12. Remove the `(else (k v))` line to ensure that you've properly removed all higher-order function representations of continuations.

Place your final version in a file named `a7.rkt` and submit it via Canvas. Make sure to include parts I and II of the assignment as well.

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, it's 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 defininging 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 it's 1s all the way down.

So we can build a procedure `take$`

```
(define take$
  (lambda (n $)
    (cond
      ((zero? n) '())
      (else (cons (car$ $)
              (let ((n- (sub1 n)))
                (cond
                  ((zero? n-) '())
                  (else (take$ 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 worlds-worst-random
    (delay (random 4)))
> (force worlds-worst-random)
2
> (force worlds-worst-random)
2
> (force worlds-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" [http://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=4&cad=rja&ved=0CDgQFjAD&url=http%3A%2F%2Fciteseerx.ist.psu.edu%2Fviewdoc%2Fdownload%3Fdoi%3D10.1.1.94.1279%26rep%3Drep1%26type%3Dp 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: 2019/02/27 17:00 by kl13