CSC324 Principles of Programming Languages

Lecture 8

November 6, 2019

Mid-course Survey Responses

- Exercises
- Assignment 1
- Working with a partner
- Examples
- Grading (show distributions in class only)

What to do going forward? (grading)

1. Assignment 1 Resubmission Possibility
2. Exercise bonuses
3. Video bonuses

No other resubmissions for this course

Assignment 1 Resubmit Proposal

- I'll run one round of tests tomorrow November 7th
- Resubmission by November 14th
  - If you would like a different deadline, email me with a reason (e.g. other assignments due around the same time) and a proposed deadline **by November 11th**.
- Additional instructor office hours on Friday November 8th, 3pm-4pm

Other ideas?

Last time...

- Continuations! (shift/reset)
- We have one implementation of `-<` and `next!`

```
> (define g (reset (+ 1 (-< 10 20))))
> (next! g)
11
> (next! g)
21
> (next! g)
'DONE
```

But what are we doing?

- We are building a *logic programming* language inside Racket

Multiple use of -<

Right now, multiple use of -< does not work:

```
> (define g (reset (+ (-< 10 20) (-< 2 3))))
> (next! g)
'(#<procedure> .  #<procedure>)
```

Today

- Multiple use of `-<`
  - Exercise 6, rank 1 expressions:
  - `(list (-< '+ '*)  (-< 1 2 3 4) (-< 1 2 3 4))`
- The query operator `?-`
- Exercise 8

Review

What will these Racket expressions evaluate to?

```
> (+ 10 (shift k 1))

> (+ 10 (shift k (k 1)))

> (+ 10 (shift k (+ (k 1) (k 2))))

> (define x (+ 10 (shift k (+ (k 1) (k 2)))))
> x
```

Review (Part 2)

What will these Racket expressions evaluate to?

```
> (define (f x) (shift k x))
> (+ 10 (f 7))

> (define (f . x) (shift k x))
> (+ 10 (f 7))

> (define (amb . x) (shift k (map k x)))
> (+ 10 (amb 2 3 4))
```

Review (Part 3)

This is the definition of `-<` that we ended up with last week:

```
(define (-< . lst)
  (shift k (map-stream k lst)))

(define (map-stream k lst)
  (if (empty? lst)
      s-null
      (s-cons (k (first lst))
              (map-stream k (rest lst)))))
```

The problem with our current -<

Problem: This code does not allow us to use multiple -< in one expression like this:

```
(list (-< '+ '*)  (-< 1 2 3 4) (-< 1 2 3 4))
```

Last time, we saw that `(+ (-< 10 20) (-< 2 3))` produced a stream where the first element is a stream.

We'll solve this problem today. But first, a little more review...

Review (Part 4)

```
> (define (amb . x) (shift k (map k x)))
> (list (+ 10 (amb 2 3 4)))

> (define (amb . x) (shift k (append-map k x)))
> (list (+ 10 (amb 2 3 4)))
```

Review (Part 4)

```
> (define (amb . x) (shift k (map k x)))
> (list (+ 10 (amb 2 3 4)))

> (define (amb . x) (shift k (append-map k x)))
> (list (+ 10 (amb 2 3 4)))

    (list (+ 10 (amb 2 3 4)))
==> (append-map k '(2 3 4)) ; k = (list (+ 10 _))
==> (append '(12) '(13) '(14))
==> '(12 13 14)
```

But wait...

```
> (define (amb . x) (shift k (append-map k x)))
> (list (+ (amb 10 20) (amb 2 3 4)))
```

But wait...

```
> (define (amb . x) (shift k (append-map k x)))
> (list (+ (amb 10 20) (amb 2 3 4)))

    (list (+ (amb 10 20) (amb 2 3 4)))
==> (shift k (append-map k '(10 20))))
      ; k = (list (+ _ (amb 2 3 4)))
==> (append-map k '(10 20))

To evaluate the map, we evaluate
    (k 10) ==> (list (+ 10 (amb 2 3 4)))
           ==> '(12 13 14) ; previous slide
    (k 20) ==> (list (+ 20 (amb 2 3 4)))
           ==> '(22 23 24) ; previous slide

==> (append '(12 13 14) '(22 23 24))
==> '(12 13 14 22 23 24)
```

What just happened?

```
> (define (amb . x) (shift k (append-map k x)))
> (list (+ (amb 10 20) (amb 2 3 4)))
'(12 13 14 22 23 24)
```

This is almost the exact result that we want, except the output is a *list* instead of a stream!

Key idea (list)

- Assume that `k` always returns *a list*
  - (note: we can show that this is true by induction)
- Have a `(list ...)` call on the *outside*, right after the *reset*

```
> (define (amb . x) (shift k (append-map k x)))
;                                  ^--------
> (reset (list (+ (amb 10 20) (amb 2 3 4))))
;           ^---
'(12 13 14 22 23 24)
```

Now, we apply the same idea to our implementation -<

1. Assume that `k` always returns a *stream*
2. Have a `(make-stream ...)` call on the *outside*, right after the *reset*

The problem with `make-stream`

What does the following expressions evaluate to?

```
> (make-stream (+ 3 4))

> (define (amb . x) (shift k (append-map k x)))
> (make-stream (amb 3 4))
; Options:
;   - a list containing 2 streams, each with a number
;   - a stream with 2 numbers as its elements
;   - a stream with a list as its first element
;   - an error
```

The problem with `make-stream` ...

```
> (make-stream (amb 3 4))
; ==> a stream with a list as its first element!!
```

Why? Because macro expansion happens *before* the call to amb

```
    (make-stream (amb 3 4))
==> (s-cons (amb 3 4) (make-stream))
```

The solution: make a new function

```
(define (singleton x) (make-stream x))

> ; note: using an older verison of `amb`
> (define (amb . x) (shift k (map k x)))
> (singleton (amb 3 4))
```

Recap of what happened so far:

1. Our implementation of `-<` from last time doesn't handle multiple calls to `-<`
2. We found a solution to this problem that uses lists instead of streams:

```
> (define (amb . x) (shift k (append-map k x)))
> (reset (list (+ (amb 10 20) (amb 2 3 4))))
```

3. The idea is to assume that `k` always returns a list, and have the `(list ...)` creation right after the `(reset ...)`
4. We're trying to apply the same idea to the implementation of `-<` that uses streams
5. We couldn't use `(make-stream ...)` because it is a macro, and macro expansion happens before code evaluation
6. We create a function `singleton` instead, so that eventually, we could do this:

```
> (reset (singleton (+ (-< 10 20) (-< 2 3 4))))
```

Implementing -<

To implement `-<` to assume that `k` always returns a stream, we need to implement the equivalent of `(append-map k _)` for streams.

```
(define (amb . x) (shift k (append-map k x)))
```

If we had such a function, which we will call `s-append-map`, then our implementation of `-<` becomes:

```
(define (-< . lst) (shift k (s-append-map k lst)))
```

Implementing `s-append-map`

This function will take

- a continuation `k`
- a list `lst`,

and

- return a stream containing all the elements in each of the streams `(k x)`, for every `x` in `lst`.

Remember s-append from two weeks ago?

```
(define (s-append s t) ; from two weeks ago
  (cond                ; not the final version
    [(s-null? s) t]
    [(pair? s)
     (s-cons (s-first s)
             (s-append (s-rest s) t))]))
```

Implementing s-append-map

We can use s-append as a helper function!

```
(define (s-append-map k lst) ; not the final version
  (if (empty? lst)
      s-null
      (s-append (k (first lst))
                (s-append-map k (rest lst)))))
```

Implementing `s-append-map`

We can use `s-append` as a helper function!

```
(define (s-append-map k lst) ; not the final version
  (if (empty? lst)
      s-null
      (s-append (k (first lst))
                (s-append-map k (rest lst)))))
```

Problem: we don't want the recursive `(s-append-map ...)` call in the last line to be evaluated when the stream is created

A bit more laziness

```
(define (s-append s t) ; the argument `t` is a thunk
  (cond
    [(s-null? s) (t)]
    [(pair? s)
     (s-cons (s-first s)
             (s-append (s-rest s) t))]))

(define (s-append-map k lst)
  (if (empty? lst)
      s-null
      (s-append (k (first lst))
                (thunk (s-append-map k (rest lst))))))
```

Simple example

```
> (define g (reset (singleton (+ 10 (-< 2 3)))))
> (next! g)
12
> (next! g)
13
> (next! g)
'DONE
```

Combining `(reset (singleton _))`

To avoid remembering to type `(reset (singleton _))` all the time, we'll create a syntax `do/-<`:

```
(define-syntax do/-<
  (syntax-rules ()
     [(do/-< <expr>) (reset (singleton <expr>))]))
```

Simple example, again

```
> (define g (do/-< (+ 10 (-< 2 3))))
> (next! g)
12
> (next! g)
13
> (next! g)
'DONE
```

Multiple use of -<

```
> (define g (do/-< (+ (-< 10 20) (-< 2 3))))
> (next! g)
12
> (next! g)
13
> (next! g)
22
> (next! g)
23
> (next! g)
'DONE
```

Isolating Errors

Our implementation has one annoying issue:

```
> (define g (do/-< (/ 1 (-< 2 1 0 4))))
> (next! g)
1/2
> (next! g)          ; error is reported here
; /: division by zero [,bt for context]
1
> (next! g)          ; error should happen here, and we shou
; car: contract violation
;   expected: pair?
;   given: #<void>
; [,bt for context]
```

We are always evaluating *one more element* than necessary!

Solution: more thunks!

- Have `-<` return a *thunk* that evaluates to a stream
- Have `next!` take as input *thunk* that evaluates to a stream

See `amb.rkt` posted

Using -< to solve ex6

Exercise 6: Rank 1 expressions

```
> (define number    (list 1 2 3 4))
> (define operation (list '+ '*))
> (define g (do/-< (list (apply -< operation)
                         (apply -< number)
                         (apply -< number))))
```

Expressions of ranks 0 and 1

```
(define number    (list 1 2 3 4))
(define operation (list '+ '*))
(define/match (expressions-of-rank k)
  [(0) (apply -< number)]
  [(1) (list (apply -< operation)
             (apply -< number)
             (apply -< number))]
  ...)
```

Expressions of arbitrary rank `k`

Two mutually exclusive cases:

1. The *left* subexpression is of rank `(- k 1)`, and the *right* subexpression is of rank either `0, 1, ..., (- k 1)`.
2. The *left* subexpression is of rank *less than* `(- k 1)`, and the *right* subexpression is of rank exactly `(- k 1)`.

Case 1...

```
(define (gen-left-equal-k k)
  (let* ([op (apply -< operation)]
         [left (expressions-of-rank (- k 1))]
         [k-right (apply -< (range 0 k)))] ; excludes k
         [right (expressions-of-rank k-right)])
    (list op left right)))
```

Case 2...

```
(define (gen-left-smaller-k k)
  (let* ([op (apply -< operation)]
         [right (expressions-of-rank (- k 1))]
         [k-left (apply -< (range 0 (- k 1)))] ; excludes i
         [left (expressions-of-rank k-left)])
    (list op left right)))
```

Put it together

Use -< to choose.

```
(define/match (expressions-of-rank k)
  [(0) (apply -< number)]
  [(1) (list (apply -< operation)
             (apply -< number)
             (apply -< number))]
  [(k) (let* [(fn (-< gen-left-smaller-k
                      gen-left-equal-k))]
         (fn k))])
```

Filtering Results

Example: Generate all triples of numbers

Use `-<` to generate triples of natural numbers, each belonging to `(range 1 n)`

```
(define (triples n)
  (let* ([nums (range 1 n)])
    (do/-< (list (apply -< nums)
                 (apply -< nums)
                 (apply -< nums)))))
```

Example: Triples whose product is n

Generate all triples of natural numbers, each less than n, whose product is n.

Use a function like `s-filter` that works like `filter` for lists.

Example: Triples whose product is n

Generate all triples of natural numbers, each less than n, whose product is n.

Use a function like `s-filter` that works like `filter` for lists.

The query operator `?-` in the notes does this!

```
(define (product? n)
  (lambda (triple)
    (equal? n (apply * triple))))
```

```
> (define g (?- (product? 6) (triples 6)))
> (next! g)
'(1 2 3)
> (next! g)
'(1 3 2)
> (next! g)
'(2 1 3)
```

But the exercise doesn't use ?-

Why? Because generating all combinations of something then discarding most values is *slow*.

First solution

This implementation is not going to make sense at first, and may seem even *worse*:

```
(define (solve-triple n lst)
  (cond
    [(equal? (length lst) 3)
     (if (equal? n (apply * lst)) lst 'FAIL)]
    [else (let* ([next-num (apply -< (range 1 n))])
            (solve-triple n (cons next-num lst)))]))
```

Problem with the first solution

```
> (define g (triples-product 4))
> (next! g)
'FAIL
> (next! g)
'FAIL
> (next! g)
'FAIL
> (next! g)
'FAIL
> (next! g)
'(2 2 1)
```

There are a lot of `'FAIL`s, which we want to ignore.

Backtracking

Backtracking: automatically continue searching in the case of failure

```
(define (solve-triple n lst)
  (cond
    [(equal? (length lst) 3)
     (if (equal? n (apply * lst))
         lst
         (shift k s-null))] ; <-- backtracking!
    [else (let* ([next-num (apply -< (range 1 n))])
            (solve-triple n (cons next-num lst)))]))
```

See notes on why this works

Backtracking Function

Backtracking: automatically continue searching in the case of failure

```
(define (fail) (shift k s-null))
```

Restricting the Search Space

Consider items already generated:

```
(define (triples-product n)
  (do/-< (solve-triple n '())))

(define (solve-triple n lst)
  (cond
    [(equal? (length lst) 3)
     (if (equal? n (apply * lst)) lst (fail))]
    [else
     (let* ([prod     (apply * lst)]
            [max-next (min n (+ (/ n prod) 1))
            [next-num (apply -< (range 1 max-next))])
       (solve-triple n (cons next-num lst)))]))
```

Exercise: Sudoku (demo in DrRacket only)