CSC324 Principles of Programming Languages

Lecture 7

October 26/27, 2020

Announcement

- Lisa can't debrief ex6 because of grace day
- Test #2 is released
- Test #3 is next week
- Project grading is still ongoing, will be at least another ~week
  - The average on the unit test portion was ~76%

Today: control flow

Control flow determines the order of evaluation of a program

Racket uses left-to-right eager evaluation, but what if we want a different evaluation order?

- Use macros and thunks
- Use *continuations*

Goal for next 2 lectures

In the next 2 lectures, we will use **streams** and **continuations** to build our own mini-logic programming language inside Racket.

We'll define the **ambiguous choice operator** -< that will behave something like this:

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

Streams

Stream

- **Stream**: an abstract model of a (possibly infinite) sequence of values over time
- Implemented as a "lazy list", a list whose elements are only evaluated when necessary

Racket Lists

In Racket, `cons` is a function that **eagerly-evaluates** its arguments!

```
> (cons 4 (cons 5 (cons (first '()) '())))
; first: contract violation
; expected: (and/c list? (not/c empty?))
```

This means we can't work with infinite lists in Racket.

Haskell Lists

In Haskell, lists are streams streams due to **lazy evaluation**

```
> lst = 4:5:(head []):[]
> head lst
4
> head (tail lst)
5
> head (tail (tail lst))
*** Exception: Prelude.head: empty list
```

Haskell allows us to work with infinite lists.

Streams in Racket?

In order to implement streams in racket we need to be able to **delay evaluation**. Two things we'll need:

1. Thunks
2. Macros

Thunks

- Remember that a **thunk** is a zero-argument function, used to delay evaluation

```
> (thunk (/ 1 0))
#<procedure>
> ((thunk (/ 1 0)))
; /: division by zero
```

Thunks and Streams

- For a **non-empty stream**, we will have a value *wrapped by a thunk*, followed by a stream *also wrapped by a thunk*.
- No element of the stream is evaluated until it is used

```
> (cons (thunk 4)
        (thunk (cons (thunk 5)
                     (thunk (cons (thunk (first '()))
                                  '())))))
'(#<procedure> . #<procedure>)
```

- We will represent the **empty stream** as a special symbol

The functions car and cdr

- first and rest work on proper *lists* only
  - <list> = '() | (cons <expr> <list>)
- The general version of these functions are called car and cdr
  - They work on an arbitrary *pair* of elements, whether or not the pair of elements is part of a list

```
> (first (cons 4 (list))) ; ok, this is a list
4
> (first (cons (thunk 4) (thunk (cons (thunk 5) '()))))
; first: contract violation
> (car (cons (thunk 4) (thunk (cons (thunk 5) '()))))
#<procedure>
> (cdr (cons (thunk 4) (thunk (cons (thunk 5) '()))))
#<procedure>
```

Thunks allow us to create infinite lists:

```
> (define (repeat n)
    (cons (thunk n) (thunk (repeat n))))
> (define x (repeat 1))
> ((car x))
1
> ((car ((cdr x))))
1
> ((car ((cdr ((cdr x))))))
1
```

Use macros to build stream functions

To avoid the `(thunk ...)` boiler plate, let's write these functions

Stream Fn.	List Equiv.	Description
s-null?	null?	Test whether stream is empty
s-cons	cons	Add an element to beginning of stream
s-first	first	Get first element of stream
s-rest	rest	Get rest of the stream
make-stream	list	Build stream from some expressions

s-null?

The empty list is represented by `'()`. We need an analog for streams!

```
(define s-null 's-null)
(define (s-null? stream) (equal? stream s-null))
```

s-cons

The macro s-cons takes <first> and <rest>, wraps them in thunks, and cons's them!

```
(define-syntax s-cons
  (syntax-rules ()
    [(s-cons <first> <rest>)
     (cons (thunk <first>) (thunk <rest>))]))
```

s-first and s-rest

```
(define (s-first stream) ((car stream)))
(define (s-rest stream) ((cdr stream)))
```

Convenient Stream-Building

To build a stream, we have to use repeated application of `s-cons`.

```
> (s-cons 1
    (s-cons 2
      (s-cons 3
        (s-cons 4
          (s-cons 5 s-null)))))
```

But building a list is so much easier!

```
> (list 1 2 3 4 5)
```

make-stream

The macro `make-stream` is like the function `list`: takes one or more expressions and `cons` them all together.

```
(define-syntax make-stream
  (syntax-rules ()
    [(make-stream) s-null]
    [(make-stream <first> <rest> ...)
     (s-cons <first> (make-stream <rest> ...))]))
```

Stream Definition Summary

```
(define s-null 's-null)
(define (s-null? stream) (equal? stream s-null))

(define-syntax s-cons
  (syntax-rules ()
    [(s-cons <first> <rest>)
     (cons (thunk <first>) (thunk <rest>))]))

(define (s-first stream) ((car stream)))
(define (s-rest stream) ((cdr stream)))

(define-syntax make-stream
  (syntax-rules ()
    [(make-stream) s-null]
    [(make-stream <first> <rest> ...)
     (s-cons <first> (make-stream <rest> ...))]))
```

Breakout Group Exercise 1

Write a function `s-take` that takes the first `n` elements of a `stream` and produces a list with those `n` elements.

Exercises (for later)

- Write a function `repeat` that takes an integer `n` and returns the infinite stream where each element is `n`.
- Write a function `s-append` that appends two streams `s` and `t` (might be helpful for exercise 7)

Self-Updating Streams

Python Generator

```
>>> vals = (x for x in [1, 2, 3]) # create generator
>>> vals
<generator object <genexpr> at 0x000001BDB5E6DF68>
>>> next(vals) # access one element at a time
1
>>> next(vals)
2
>>> next(vals)
3
>>> next(vals) # no more elements
StopIteration
```

Python Generator Function

```
def gen():
    for i in [3, 2, 1, 0]:
        yield 12 / i

>>> xs = gen()
>>> next(xs)
4
>>> next(xs)
6
>>> next(xs)
12
>>> next(xs)
ZeroDivisionError
```

Infinite Iterator in Python

```
define repeat(n):
    while True:
        yield n

>>> g = repeat(3)
>>> next(g)
3
>>> next(g)
d3
```

Desired Self-Updating Stream in Racket

```
> (define s (make-stream 1 2 3))
> (next! s)
1
> (next! s)
2
> (next! s)
3
> (next! s)
'DONE
> s
's-null
```

Mutation!

The exclamation mark `!` at the end of `next!` is pronounced "bang"

The most basic mutation function in Racket is `set!`

```
>>> (define x 4)
>>> (set! x 2)
>>> x
2
```

Behaviour of `next!`

The syntax `next!` takes a stream, and if the stream is non-empty:

- Update the stream to `s-rest` of the stream
- Return the `s-first` of the stream

If the stream is empty, return 'DONE.

Behaviour of `next!` on non-empty stream

```
(let* ([tmp s])
  (begin
    (set! s (s-rest s))
    (s-first tmp)))
```

The macro next!

```
(define-syntax next!
  (syntax-rules ()
    [(next! <g>)
     (if (s-null? <g>)
         'DONE
         (let* ([tmp <g>])
           (begin
             (set! <g> (s-rest <g>))
             (s-first tmp))))]))
```

Closer to -<

Here's an example of how -< should behave:

```
> (define g (-< 1 2 (+ 3 4)))
> (next! g)
1
> (next! g)
2
> (next! g)
7
> (next! g)
'DONE
```

But isn't -< just `make-stream`?

```
> (define g (make-stream 1 2 (+ 3 4)))
> (next! g)
1
> (next! g)
2
> (next! g)
7
> (next! g)
'DONE
```

The Ambiguous Choice Operator -<

The -< operator will also *capture the surrounding computation* so that an expression like:

```
(+ 10 (-< 1 2 (+ 3 4)))
```

. . . would create a stream with the values:

- $(+\ 10\ 1)$
- $(+\ 10\ 2)$
- $(+\ 10\ (+\ 3\ 4))$

Continuations

Continuations: Definition

The **continuation** of an expression *s* representation of what has to be evaluated *after s* is evaluated

You can think of a continuation as the *rest of the stack frame*.

Continuations: Example 1

```
(+ (* 3 4) (first (list 1 2 3)))
```

- What is the continuation of `(* 3 4)`?
- i.e. what has to be evaluated after evaluating `(* 3 4)`?

Continuations: Example 1

```
(+ (* 3 4) (first (list 1 2 3)))
```

- What is the continuation of `(* 3 4)`?
- i.e. what has to be evaluated after evaluating `(* 3 4)`?

English: evaluate `(first (list 1 2 3))`, and add that to whatever we got when we evaluated `(* 3 4)`

Continuations: Example 1

`(+ (* 3 4) (first (list 1 2 3)))`

- What is the continuation of `(* 3 4)`?
- i.e. what has to be evaluated after evaluating `(* 3 4)`?

English: evaluate `(first (list 1 2 3))`, and add that to whatever we got when we evaluated `(* 3 4)`

Our Notation: `(+ _ (first (list 1 2 3)))`

The _ is the subexpression whose continuation we are representing.

Continuations: Example 2

```
(+ (* 3 4) (first (list 1 2 3)))
```

- What is the continuation of 4?

Continuations: Example 2

```
(+ (* 3 4) (first (list 1 2 3)))
```

- What is the continuation of 4?

```
(+ (* 3 _) (first (list 1 2 3)))
```

Continuations: Example 3

```
(+ (* 3 4) (first (list 1 2 3)))
```

- What is the continuation of `(first (list 1 2 3))`?

Continuations: Example 3

```
(+ (* 3 4) (first (list 1 2 3)))
```

- What is the continuation of `(first (list 1 2 3))`?

```
(+ 12 _)
```

Remember: eager left-to-right evaluation order!

Continuations: Example 4

```
(+ (* 3 4) (first (list 1 2 3)))
```

- What is the continuation of +?

Continuations: Example 4

```
(+ (* 3 4) (first (list 1 2 3)))
```

- What is the continuation of +?

```
(_ (* 3 4) (first (list 1 2 3)))
```

Continuations: Example 5

```
(+ (* 3 4) (first (list 1 2 3)))
```

- What is the continuation of `(+ (* 3 4) (first (list 1 2 3)))`?

Continuations: Example 5

```
(+ (* 3 4) (first (list 1 2 3)))
```

- What is the continuation of `(+ (* 3 4) (first (list 1 2 3)))`?

```
_
```

Continuations as Values

In Racket, continuations are **first-class** data types: they are values, just like numbers, lists, and procedures!

The syntactic form `shift` captures the continuation of an expression

```
(shift <id> <body>)
```

```
> (require racket/control)
> (shift hi 8)
8
> (shift hi (+ 5 9))
14
```

Continuations as Values

In Racket, continuations are **first-class** data types: they are values, just like numbers, lists, and procedures!

The syntactic form `shift` captures the continuation of an expression

```
(shift <id> <body>)
```

```
> (require racket/control)
> (shift hi 8)
8
> (shift hi (+ 5 9))
14
```

```
> hi
; hi: undefined;
```

Continuations as Values

In Racket, continuations are **first-class** data types: they are values, just like numbers, lists, and procedures!

The syntactic form `shift` captures the continuation of an expression

```
(shift <id> <body>)
```

```
> (require racket/control)
> (shift hi 8)
8
> (shift hi (+ 5 9))
14

> hi
; hi: undefined;

> (+ 5 (shift hi 1))
1
```

The `shift` syntactic form

```
(<shift> <id> <body>)
```

- Bind the current continuation to `<id>`
- Evaluates `<body>` in the current environment
- … with one additional binding: `<id>` is bound to the *continuation* of the `shift` expression
- … and **ignore the continuation** of the `shift` expression

Storing the Continuation

```
> (require racket/control)
> (define cont (void))
> (+ (* 3 (shift k (set! cont k))) 1)
```

The value `cont` stores the continuation (the *rest of the stack frame*).

What can we do with `cont`?

Calling a Continuation

- We can call a continuation
- Same syntax as a function call

```
> ; cont is (+ (* 3 _) 1)
> (cont 4)
13
> (cont 100)
301
> (+ 2 (cont 100))
303
```

Applying the continuation in `shift`

Note that `shift` does not automatically its own continuation!

```
> (+ 2 (shift k 3)) ; k = (+ 2 _)
3
```

If we want to apply `k`, we need to do it explicitly:

```
> (+ 2 (shift k (k 3))) ; k = (+ 2 _)
5
```

Applying the continuation multiple times

```
> (+ 2 (shift k (* (k 3) (k 4)))) ; # k = (+ 2 _)
30
```

What's happening here?

Applying the continuation multiple times

```
> (+ 2 (shift k (* (k 3) (k 4)))) ; # k = (+ 2 _)
30
```

What's happening here?

```
    (+ 2 (shift k (* (k 3) (k 4))))
==> (* (k 3) (k 4)); # k = (+ 2 _)
==> (* ((+ 2 _) 3) ((+ 2 _) 4))
==> (* 5 6)
==> 30
```

The problem with `shift`

```
> (+ 2 (shift k 3))
3
> (define n (+ 2 (shift k 3)))
3
> n
; n: undefined;
;  cannot reference an identifier before its definition
;   in module: top-level
; [,bt for context]
```

What happened?

The problem with `shift`

```
> (+ 2 (shift k 3))
3
> (define n (+ 2 (shift k 3)))
3
> n
; n: undefined;
;  cannot reference an identifier before its definition
;   in module: top-level
; [,bt for context]
```

What happened?

The problem is that `shift` captures all remaining context!!

The delimiter `reset`

```
> (define n (reset (+ 2 (shift k 3))))
> n
3
```

- `shift` will only capture the context *up to the nearest reset*

Breakout Group Exercise 2

What do these expressions evaluate to?

- `(reset (* 10 (+ 2 (shift k (* (k 3) (k 4))))))`
- `(* 10 (reset (+ 2 (shift k (* (k 3) (k 4))))))`
- `(* 10 (+ 2 (reset (shift k (* (k 3) (k 4))))))`

Using Continuations in –<

Desired Syntax of -<

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

What should -< do?

- Capture the continuation
- Apply the continuation to every argument of -<
- Put the result in a stream

First implementation of -<

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

Where `map-stream` applys `k` to every element of `lst`, and returns a stream of the result.

Exercise: Implement `map-stream`

Example

```
> (define g (reset (+ 1 (-< 10 20)))) ; `k` is (+ 1 _)
> (next! g)    ; g is (s-cons (k 10) (map-stream k '(20)))
11
> (next! g)    ; g is (s-cons (k 20) (map-stream k '()))
21
> (next! g)
'DONE
```

Multiple use of -<

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

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

Tracing through multiple use of -<

```
    (reset (+ (-< 10 20) (-< 2 3)))
==> (shift k (map-stream k '(10 20)))
    ; where k = (+ _ (-< 2 3))
==> (s-cons ((+ _ (-< 2 3)) 10) (map-stream k '(20)))
```

- When `((+ _ (-< 2 3)) 10)` is evaluated, `(+ 10 (-< 2 3))` returns a stream!
- That's why the first element of `g` was a stream

Next steps

Next class, we'll redefine -< to support multiple uses of -<