CSC324 Principles of Programming Languages

Lecture 7

October 30, 2019

Midterm

- Marks and solutions will be released today
- **LEC0101** midterm was harder than **LEC0102** by a significant amount, and will be adjusted

```
(define (midterm-percent-grade-LEC0101 raw-score)
    (/ (+ raw-score 1.5) 23.5))
(define (midterm-percent-grade-LEC0102 raw-score)
    (/ raw-score 25))
```

Midterm Takeup

Lab 7: Midterm Takeup

Attend the takeup session that you are interested in:

- LEC0101 midterm in room DH2020
- LEC0102 midterm in room DH2026

Assignment 1

Grades are posted on Markus

- 45% Part 1
- 35% Parts 2 and 3
- (Of which 5% is on currying + etc)
- 20% TA grading

Exercises

Exercise 6

- Task 1 and 2 were good review for the midterm, should be straightforward
- We'll talk more aebout task 3 next week

Exercises 7 and 8

- Both of these assignments will be in Racket
- I decided against adding a Haskell component
- Start early!

Exercise 10

- This exercise will be in Haskell, and will be related to today's material
- This exercise leads up to assignment 2

Last time...

We introduced `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))))]))
```

Today

Today, we'll create the *ambiguous choice operator* `-<` (pronounced "amb") that behaves like this:

```
> (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))

Why -<?

- Can be used to generate a lot of options
- Then we can search the result for ones that satisfy a given constraint

Example:

- Use the syntax `(list (-< 0 1) (-< 0 1) (-< 0 1) (-< 0 1))` to create a stream of all four digit binary values
- We can then filter through the stream for something we're looking for (e.g. same numbers of 0s and 1s)

First goal

Have . . .

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

. . . create a stream with the values:

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

Today

- Continuations: capturing the surrounding computation
- Implementing a first version of -<

Continuations

Continuations: Definition

- **Continuation** of s: representation of what has to be evaluated *after* s is evaluated
  - Does *not* include the evaluation of s itself

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)`

Racket: `(+ _ (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>)
```

```
> (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>)
```

```
> (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>)
```

```
> (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*

Examples of `reset`

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 -<