

CSC324 Lecture 13

9 July 2020

Announcements

Assignment 1 grading is in progress. The ones I've eyeballed so far look good.

Assignment 2 will be out today! Due by Friday, 7 August (Still waiting on feedback on it from the TAs, but it should be good to go by now.)

Last time...

We saw how an expression that is in the process of being evaluated contains:

- the **redex**, which is what will be reduced in the next reduction step;
- the **continuation**, which is everything that remains to be reduced

We saw how **shift** and **reset** are special forms that allow us to capture continuations out of an expression.

Today: Let's implement something interesting with them!

Determinism and non-determinism

Up to this point, we have seen how the output of pure functions is entirely dependent on its inputs.

A referentially-transparent pure function called with the same arguments will always produce the same value, no matter the context in which it is called.

Determinism and non-determinism

Definition: We say that an expression is deterministic if it only evaluates to one value.

Definition: We say that an expression is non-deterministic if it *may* evaluate to more than one possible value.

-< : the mysterious ambiguous operator

```
; The -< operator (pronounced "amb", for ambiguous)  
; consumes one or more expressions, and returns an  
; "ambiguously-chosen" one of them.
```

-< : the mysterious ambiguous operator

```
; The -< operator (pronounced "amb", for ambiguous)
; consumes one or more expressions, and returns an
; "ambiguously-chosen" one of them.

; The following expression has one of three possible
; values: 1, 2, or 3.
(-< 1 2 3)
```

-< : the mysterious ambiguous operator

```
; The -< operator (pronounced "amb", for ambiguous)
; consumes one or more expressions, and returns an
; "ambiguously-chosen" one of them.

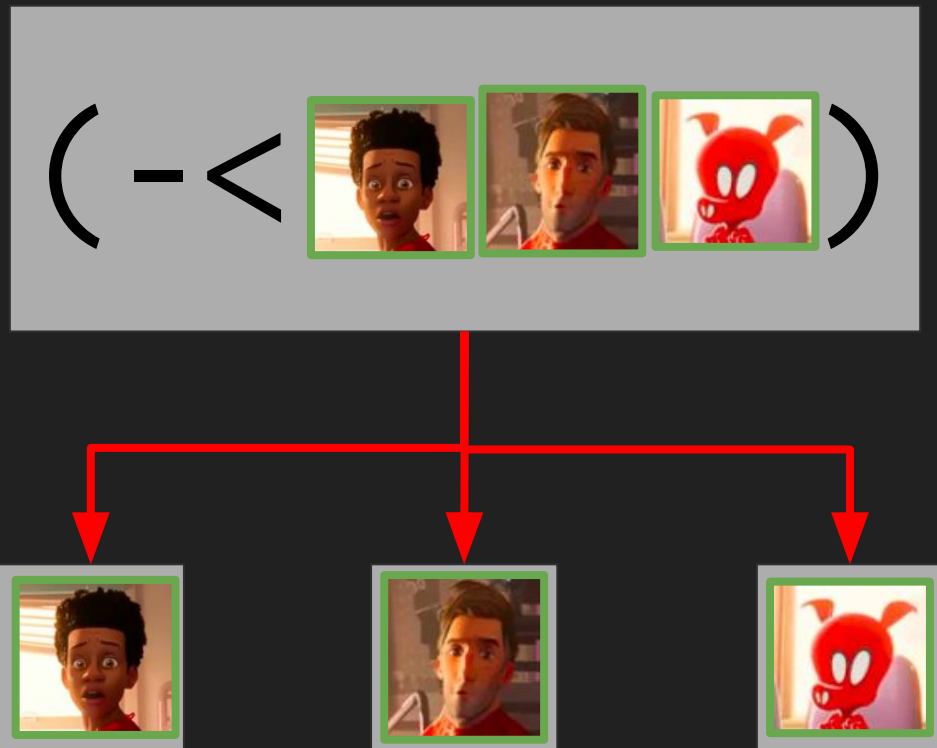
; The following expression has one of three possible
; values: 1, 2, or 3.
(-< 1 2 3)

; The following expression has one of six values:
; '(1 a), '(1 b), '(2 a), '(2 b), '(3 a), '(3 b).
(list (-< 1 2 3) (-< 'a 'b))
```


Choice points

One can imagine that `amb` causes time to split into branches, where the computation continues on each branch with one of the possible values of the expression.

Definition: Such an expression is said to be a non-deterministic choice point.



Hopping between universes with next!

Calling `(next!)` will evaluate the surrounding expression but with a different chosen choice point.

```
> (list (-< 1 2) (-< 'a 'b))  
'(1 a)
```

Hopping between universes with next!

Calling `(next!)` will evaluate the surrounding expression but with a different chosen choice point.

```
> (list (-< 1 2) (-< 'a 'b))  
'(1 a)  
> (next!)  
'(1 b)
```

Hopping between universes with next!

Calling `(next!)` will evaluate the surrounding expression but with a different chosen choice point.

```
> (list (-< 1 2) (-< 'a 'b))  
'(1 a)  
> (next!)  
'(1 b)  
> (next!)  
'(2 a)
```

Hopping between universes with next!

Calling `(next!)` will evaluate the surrounding expression but with a different chosen choice point.

Once we've called `(next!)`, we can't return to the old choice point!

```
> (list (-< 1 2) (-< 'a 'b))  
'(1 a)  
> (next!)  
'(1 b)  
> (next!)  
'(2 a)  
> (next!)  
'(2 b)
```

Hopping between universes with next!

Calling `(next!)` will evaluate the surrounding expression but with a different chosen choice point.

Once we've called `(next!)`, we can't return to the old choice point!

The symbol `'done` will be produced when all choice points are exhausted.

```
> (list (-< 1 2) (-< 'a 'b))  
'(1 a)  
> (next!)  
'(1 b)  
> (next!)  
'(2 a)  
> (next!)  
'(2 b)  
> (next!)  
'done  
>
```

Today:

- Let's go implement amb!

An initial implementation...

```
#lang racket

(define choices (box (void)))
(define (set-choices! x) (set-box! choices x))

(define-syntax -<
  (syntax-rules ()
    [(-< expr) ; base case: one option
     (begin (set-choices! (void))
             expr)]

    ; if there is more than one choice, return the
    ; first one and store the rest in a thunk.
    [(-< expr1 expr2 ...)
     (begin (set-choices! (thunk (-< expr2 ...)))
             expr1))])

(define (next!)
  (if (void? (unbox choices)) 'done ((unbox choices))))
```


Mutation!?!?!?!?!?!?!?

Nathan! Are you feeling ok?

Yes! We are going to be mutating the choices identifier when we grab a new choice.

We said how a disadvantage of mutation is that it breaks referential transparency, but here explicitly want it broken; that's how we get amb to be non-deterministic! We better hope that this choice opens up new design possibilities...let's see!

So far, we haven't used `-<` as a subexpression in a larger one

```
> (+ 100 (-< 1 2 3))
```

```
101
```

```
> ; ok, makes sense, we get back 1 and add 100 to it.
```

```
  (next!)
```

```
2
```

```
> ; intuitively, we should have gotten back 2 and added 100 to it again..?
```

```

#lang racket

(require racket/control)

(define choices (void))
(define (set-choices! x) (set! choices x))

(define-syntax -<
  (syntax-rules ()
    [(-< expr) ; base case: one option
     (begin (set-choices! (void))
             expr)]

    ; if there is more than one choice, return the
    ; first one and store the rest in a thunk.
    [(-< expr1 expr2 ...)
     (shift k
            (begin (set-choices! (thunk (k (-< expr2 ...))))
                    (k expr1))))]))

(define (next!)
  (if (void? (unbox choices)) 'done ((unbox choices))))

```

(next!) in larger expressions

Let's play the same game with (next!) -
does it behave correctly in larger
expressions?

We can see that it does not.

```
> (-< 1 2 3)
1
> (* 2 (next!))
4
> (* 2 (next!))
12
> ; where did the extra (* 2) come from|
```

$(* \ 2 \ (-< \ 1 \ 2 \ 3))$

```
(* 2 (-< 1 2 3))
```

```
(* 2 (shift k1 ; k1 is (lambda (x) (* 2 x)  
  (begin  
    (set-choices! (thunk (k1 (-< 2 3))))  
    (k1 1))))
```

```
(* 2 (-< 1 2 3))
```

```
(* 2 (shift k1 ; k1 is (lambda (x) (* 2 x)  
            (k1 1))))
```

`(* 2 (-< 1 2 3))`

; k1 is (lambda (x) (2 x))*

`(k1 1)` ; remember that evaluating a shift
; drops the surrounding expression!

`(* 2 (-< 1 2 3))`

`(* 2 1)` ; remember that evaluating a shift
; absorbs the surrounding expr!

$(* \ 2 \ (-< \ 1 \ 2 \ 3))$

2

What about (next!) in a larger expression?

```
(* 3 (next!))
```

```
; k1 is ( $\lambda$  (x) (* 2 x))
```

```
; Recall: choices was set to
```

```
; (thunk (k1 (-< 2 3)))
```

```
(* 3 (next!))
```

```
; k1 is ( $\lambda$  (x) (* 2 x))
```

```
(* 3 (unbox...)) ; Recall: choices was set to  
; (thunk (k1 (-< 2 3)))
```

```
(* 3 (next!))
```

```
; k1 is ( $\lambda$  (x) (* 2 x))
```

```
(* 3 (k1 (-< 2 3))))
```

```
(* 3 (next!))
```

```
; k1 is ( $\lambda$  (x) (* 2 x))
```

```
(* 3 (k1 (shift k2
```

```
(begin
```

```
(set-choices! (thunk (k2 (-< 3)))))
```

```
(k2 1))))))
```

We ate too much of the enclosing expression! Is there a way to not capture everything in the outer expression (highlighted in red)?

```
                                ; k1 is ( $\lambda$  (x) (* 2 x))  
(* 3 (k1 (shift k2  
          (begin  
            (set-choices! (thunk (k2 (-< 3))))  
            (k2 1)))))
```

reset to the rescue!

Shift is a special form of one argument

(reset <expr-with-a-shift>)

Reset **limits where the continuation extends to** in the expression; anything outside the reset will not be bound to *k*, but evaluated normally as if there was no shift.

(* 2 (reset (+ 1 (shift k (+ 1 2)))))




```

(define-syntax -<
  (syntax-rules ()
    [(-< expr) ; base case: one option
      (begin (set-choices! (void))
              expr)]

    ; if there is more than one choice, return the
    ; first one and store the rest in a thunk.
    [(-< expr1 expr2 ...)
      (shift k
              (begin (set-choices! (thunk (k (-< expr2 ...))))
                      (k expr1))))))

(define (next!)
  (if (void? choices)
      'done
      (reset (choices))))

```

```
(* 3 (next!))
```

```
(* 3 (reset (choices)))
```

```
(* 3 (next!))
```

```
(* 3 (reset (k1 (-< 2 3))))
```

```
(* 3 (next!))
```

The continuation *would have been* everything outside the shift, but...

```
(* 3 (reset (k1 (shift k2
```

```
(begin
```

```
(s-c! (thunk (k2 (-< 3)))
```

```
(k2 3))))))
```

```
(* 3 (next!))
```

The reset blocks us from absorbing * 3
into the continuation!

```
(* 3 (reset (k1 (shift k2
```

```
(begin
```

```
(s-c! (thunk (k2 (-< 3)))
```

```
(k2 3))))))
```

```
(* 3 (next!))
```

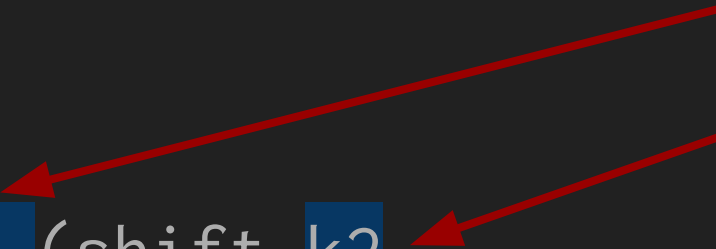
```
(* 3 (reset (k1 (shift k2
```

```
(begin
```

```
(s-c! (thunk (k2 (-< 3)))
```

```
(k2 3))))))
```

Notice that k1 and k2
close over the same
expression!



```
(* 3 (next!))
```

```
(* 3 (reset (k1 (shift k2
```

```
(begin
```

```
(s-c! (thunk (k2 (-< 3)))
```

```
(k2 3))))))
```

Notice that k1 and k2
close over the same
expression!

And, notice how k2 is
in tail position...

Continuations and tail-calls

```
(define (slow-add m n)
  (cond [(equal? m 0) n]
        [else (slow-add (- m 1) (+ n 1))]))
```

A recursive call is in tail position if

- whenever the call becomes a redex,
- **its continuation** is the same as was **the enclosing expression's continuation**

Continuations and tail-calls

```
(define (slow-add m n)
  (cond [(equal? m 0) n]
        [else (slow-add (- m 1) (+ n 1))]))
```

A recursive call is in tail position if

- whenever the call becomes a redex,
- **its continuation** is the same as was **the enclosing expression's continuation**

Next time:

- We will continue our discussion of amb and introduce **backtracking search**.