

Bride of Continuations

Echo Recitation • № 7 • 28 February 2007

- Here's how to think about continuations:

- ~~Here's how to think about continuations:~~

Fat Chance

- ~~Here's how to think about continuations:~~

Fat Chance

- Here's what you **NEED** to think about continuations:
 - A keen ability to envision the computation.
 - "If I'm here, what comes next?"
 - Inductive thinking:
"How do I express 'the rest' of the problem"?
- Why?

- Three examples:
 - Act 1: Early stopping
 - Act 2: Failure
 - Act 3: More search and backtrack

- Artificial problem:

```
(* val pi : int list -> int
   pi [a1,...,an] a1*...*an or 1 if any n=0.
   NOTE: Could return immediately on discovery of 0!
*)
```

- Artificial problem:

```
(* val pi : int list -> int
   pi [a1,...,an] a1*...*an or 1 if any n=0.
   NOTE: Could return immediately on discovery of 0!
*)
```

- Regular recursive strategy:

```
(* val pi : int list -> int
   fun recPi nil      = 1
       | recPi (0::_) = 0
       | recPi (a::a1) = a * recPi a1
```

- Artificial problem:

```
(* val pi : int list -> int
   pi [a1,...,an] a1*...*an or 1 if any n=0.
   NOTE: Could return immediately on discovery of 0!
   *)
```


- Regular recursive strategy:

```
(* val pi : int list -> int
   fun recPi nil      = 1
       | recPi (0::_) = 0
       | recPi (a::a1) = a * recPi a1
```

- Can't escape the stack, so... continuations idea:
 - Build our own stack out of functions and pass it around.

- Tail recursive
- Instead of accumulator, there's a "stack" of pending computations.
- Stack? Yeah, similar—first in, last out.

- Tail recursive
- Instead of accumulator, there's a "stack" of pending computations.
 - Stack? Yeah, similar—first in, last out.
- Pending computations begin to be executed when you hit a "base case"—just like the stack unrolls at the "bottom" of a recursion.
- Pending computations are tossed if convenient.

- Tail recursive
- Instead of accumulator, there's a "stack" of pending computations.
- Stack? Yeah, similar—first in, last out.
- Pending computations begin to be executed when you hit a "base case"—just like the stack unrolls at the "bottom" of a recursion.
- Pending computations are tossed if convenient.
-  It's over there.

- Two examples:

```
pi' [4,2] (fn x0=>x0)
```

```
pi' [2] (fn x1=>(fn x0=>x0) (4 * x1))
```

```
pi' nil (fn x2=>(fn x1=>(fn x0=>x0) (4 * x1)) (2 * x2))
```

```
(fn x2=>(fn x1=>(fn x0=>x0) (4 * x1)) (2 * x2)) 1
```

```
(fn x1=>(fn x0=>x0) (4 * x1)) (2 * 1)
```

```
(fn x0=>x0) (4 * 2)
```

8

```
pi' [4,0] (fn x0=>x0)
```

```
pi' [0] (fn x1=>(fn x0=>x0) (4 * x1))
```

0

- Envision searching for something in a tree.
- The behavior of pending computations differs depending on whether the search succeeds or fails in a subtree.
- Two behaviors means you have to pass around two stacks, two continuations.
 - Success continuation and failure continuation.

```
val t1 = Node (Node (Leaf 1, Leaf 2),  
              Node (Leaf 3, Leaf 4))
```



```
- findPath t1 1;  
val it = SOME (L(L(H)))
```

```
- findPath t1 5;  
val it = NONE
```



```
finalPath t1 3
```

```
finalPath t1 3  
fp t1 SOME retNONE
```

```
finalPath t1 3
fp t1 SOME retNONE
fp (Node (Leaf 1, Leaf 2)) (SOME o L)
                             (fn () => fp (Node (Leaf 3, Leaf 4))
                             (SOME o R) retNONE)
```

```
finalPath t1 3
```

```
fp t1 SOME retNONE
```

```
fp (Node (Leaf 1, Leaf 2)) (SOME o L)
```

```
(fn () => fp (Node (Leaf 3, Leaf 4))  
          (SOME o R) retNONE)
```

FC1

```
finalPath t1 3
fp t1 SOME retNONE
fp (Node (Leaf 1, Leaf 2)) (SOME o L)
                             (fn () => fp (Node (Leaf 3, Leaf 4))
                             (SOME o R) retNONE)
fp (Leaf 1) (SOME o L o L) FC1
              (fn () => fp (Leaf 2) (SOME o L o R) FC1)
```

```
finalPath t1 3
fp t1 SOME retNONE
fp (Node (Leaf 1, Leaf 2)) (SOME o L)
                             (fn () => fp (Node (Leaf 3, Leaf 4))
                                       (SOME o R) retNONE)
fp (Leaf 1) (SOME o L o L) FC1
              (fn () => fp (Leaf 2) (SOME o L o R) FC1)
if 1=3 then (SOME o L o L) H
           else (fn () => fp (Leaf 2) (SOME o L o R) FC1) ()
```

```
finalPath t1 3
fp t1 SOME retNONE
fp (Node (Leaf 1, Leaf 2)) (SOME o L)
                             (fn () => fp (Node (Leaf 3, Leaf 4))
                                           (SOME o R) retNONE)
fp (Leaf 1) (SOME o L o L) FC1
              (fn () => fp (Leaf 2) (SOME o L o R) FC1)
if 1=3 then (SOME o L o L) H
            else (fn () => fp (Leaf 2) (SOME o L o R) FC1) ()
fp (Leaf 2) (SOME o L o R) FC1      (* BACKTRACKING! *)
```

```
finalPath t1 3
fp t1 SOME retNONE
fp (Node (Leaf 1, Leaf 2)) (SOME o L)
                             (fn () => fp (Node (Leaf 3, Leaf 4))
                                           (SOME o R) retNONE)
fp (Leaf 1) (SOME o L o L) FC1
              (fn () => fp (Leaf 2) (SOME o L o R) FC1)
if 1=3 then (SOME o L o L) H
            else (fn () => fp (Leaf 2) (SOME o L o R) FC1) ()
fp (Leaf 2) (SOME o L o R) FC1 (* BACKTRACKING! *)
if 2=3 then ... else FC1()
```

```
finalPath t1 3
fp t1 SOME retNONE
fp (Node (Leaf 1, Leaf 2)) (SOME o L)
                             (fn () => fp (Node (Leaf 3, Leaf 4))
                                       (SOME o R) retNONE)
fp (Leaf 1) (SOME o L o L) FC1
              (fn () => fp (Leaf 2) (SOME o L o R) FC1)
if 1=3 then (SOME o L o L) H
            else (fn () => fp (Leaf 2) (SOME o L o R) FC1) ()
fp (Leaf 2) (SOME o L o R) FC1 (* BACKTRACKING! *)
if 2=3 then ... else FC1()
FC1()
```

```
finalPath t1 3
fp t1 SOME retNONE
fp (Node (Leaf 1, Leaf 2)) (SOME o L)
                             (fn () => fp (Node (Leaf 3, Leaf 4))
                                       (SOME o R) retNONE)
fp (Leaf 1) (SOME o L o L) FC1
              (fn () => fp (Leaf 2) (SOME o L o R) FC1)
if 1=3 then (SOME o L o L) H
            else (fn () => fp (Leaf 2) (SOME o L o R) FC1) ()
fp (Leaf 2) (SOME o L o R) FC1 (* BACKTRACKING! *)
if 2=3 then ... else FC1()
FC1()
fp (Node (Leaf 3, Leaf 4)) (SOME o R) retNONE
```

```
finalPath t1 3
fp t1 SOME retNONE
fp (Node (Leaf 1, Leaf 2)) (SOME o L)
                             (fn () => fp (Node (Leaf 3, Leaf 4))
                                           (SOME o R) retNONE)
fp (Leaf 1) (SOME o L o L) FC1
              (fn () => fp (Leaf 2) (SOME o L o R) FC1)
if 1=3 then (SOME o L o L) H
            else (fn () => fp (Leaf 2) (SOME o L o R) FC1) ()
fp (Leaf 2) (SOME o L o R) FC1 (* BACKTRACKING! *)
if 2=3 then ... else FC1()
FC1()
fp (Node (Leaf 3, Leaf 4)) (SOME o R) retNONE
fp (Leaf 3) (SOME o R o L)
              (fn () => fn (Leaf 4) (SOME o R o R) retNONE)
```

```
finalPath t1 3
fp t1 SOME retNONE
fp (Node (Leaf 1, Leaf 2)) (SOME o L)
                             (fn () => fp (Node (Leaf 3, Leaf 4))
                                           (SOME o R) retNONE)
fp (Leaf 1) (SOME o L o L) FC1
              (fn () => fp (Leaf 2) (SOME o L o R) FC1)
if 1=3 then (SOME o L o L) H
            else (fn () => fp (Leaf 2) (SOME o L o R) FC1) ()
fp (Leaf 2) (SOME o L o R) FC1 (* BACKTRACKING! *)
if 2=3 then ... else FC1()
FC1()
fp (Node (Leaf 3, Leaf 4)) (SOME o R) retNONE
fp (Leaf 3) (SOME o R o L)
              (fn () => fn (Leaf 4) (SOME o R o R) retNONE)
if 3=3 then (SOME o R o L) H else ...
```

```
finalPath t1 3
fp t1 SOME retNONE
fp (Node (Leaf 1, Leaf 2)) (SOME o L)
                             (fn () => fp (Node (Leaf 3, Leaf 4))
                                           (SOME o R) retNONE)
fp (Leaf 1) (SOME o L o L) FC1
              (fn () => fp (Leaf 2) (SOME o L o R) FC1)
if 1=3 then (SOME o L o L) H
            else (fn () => fp (Leaf 2) (SOME o L o R) FC1) ()
fp (Leaf 2) (SOME o L o R) FC1 (* BACKTRACKING! *)
if 2=3 then ... else FC1()
FC1()
fp (Node (Leaf 3, Leaf 4)) (SOME o R) retNONE
fp (Leaf 3) (SOME o R o L)
              (fn () => fn (Leaf 4) (SOME o R o R) retNONE)
if 3=3 then (SOME o R o L) H else ...
(SOME o R o L) H
```

```
finalPath t1 3
fp t1 SOME retNONE
fp (Node (Leaf 1, Leaf 2)) (SOME o L)
                             (fn () => fp (Node (Leaf 3, Leaf 4))
                                           (SOME o R) retNONE)
fp (Leaf 1) (SOME o L o L) FC1
              (fn () => fp (Leaf 2) (SOME o L o R) FC1)
if 1=3 then (SOME o L o L) H
            else (fn () => fp (Leaf 2) (SOME o L o R) FC1) ()
fp (Leaf 2) (SOME o L o R) FC1 (* BACKTRACKING! *)
if 2=3 then ... else FC1()
FC1()
fp (Node (Leaf 3, Leaf 4)) (SOME o R) retNONE
fp (Leaf 3) (SOME o R o L)
              (fn () => fn (Leaf 4) (SOME o R o R) retNONE)
if 3=3 then (SOME o R o L) H else ...
            (SOME o R o L) H
            SOME (R (L (H)))
```

- Siamese Strings
 - e.g. "bobo", "haha", "mikeerdmannmikeerdmann"
$$S = \{ s \mid s = xx \text{ for any string } x \}$$
 - Includes the empty string.
- What's the strategy to use for an acceptor for Siamese Strings?
- How can we use continuations?

`twin' bobo (fn (s0, t0) => s0=t0)`

twin' bobo (fn (s₀, t₀) => s₀=t₀)

twin' obo (fn (s₁, t₁) => (fn (s₀, t₀) => s₀=t₀) (b :: s₁, t₁))

```
twin' bobo (fn (s0, t0) => s0=t0)
twin' obo (fn (s1, t1) => (fn (s0, t0) => s0=t0) (b::s1, t1))
twin' bo (fn (s2, t2) =>
          (fn (s1, t1) =>
            (fn (s0, t0) => s0=t0) (b::s1, t1)) (o::s2, t2))
```

```
twin' bobo (fn (s0, t0) => s0=t0)
twin' obo (fn (s1, t1) => (fn (s0, t0) => s0=t0) (b::s1, t1))
twin' bo (fn (s2, t2) =>
          (fn (s1, t1) =>
            (fn (s0, t0) => s0=t0) (b::s1, t1)) (o::s2, t2))
(fn (s2, t2) =>
  (fn (s1, t1) =>
    (fn (s0, t0) => s0=t0) (b::s1, t1)) (o::s2, t2)) (nil, bo)
```

```
twin' bobo (fn (s0, t0) => s0=t0)
twin' obo (fn (s1, t1) => (fn (s0, t0) => s0=t0) (b::s1, t1))
twin' bo (fn (s2, t2) =>
      (fn (s1, t1) =>
        (fn (s0, t0) => s0=t0) (b::s1, t1)) (o::s2, t2))
(fn (s2, t2) =>
  (fn (s1, t1) =>
    (fn (s0, t0) => s0=t0) (b::s1, t1)) (o::s2, t2)) (nil, bo)
(fn (s1, t1) =>
  (fn (s0, t0) => s0=t0) (b::s1, t1)) (o, bo)
```

```
twin' bobo (fn (s0, t0) => s0=t0)
twin' obo (fn (s1, t1) => (fn (s0, t0) => s0=t0) (b::s1, t1))
twin' bo (fn (s2, t2) =>
      (fn (s1, t1) =>
        (fn (s0, t0) => s0=t0) (b::s1, t1)) (o::s2, t2))
(fn (s2, t2) =>
  (fn (s1, t1) =>
    (fn (s0, t0) => s0=t0) (b::s1, t1)) (o::s2, t2)) (nil, bo)
(fn (s1, t1) =>
  (fn (s0, t0) => s0=t0) (b::s1, t1)) (o, bo)
(fn (s0, t0) => s0=t0) (bo, bo)
```

```
twin' bobo (fn (s0, t0) => s0=t0)
twin' obo (fn (s1, t1) => (fn (s0, t0) => s0=t0) (b::s1, t1))
twin' bo (fn (s2, t2) =>
          (fn (s1, t1) =>
            (fn (s0, t0) => s0=t0) (b::s1, t1)) (o::s2, t2))
(fn (s2, t2) =>
  (fn (s1, t1) =>
    (fn (s0, t0) => s0=t0) (b::s1, t1)) (o::s2, t2)) (nil, bo)
(fn (s1, t1) =>
  (fn (s0, t0) => s0=t0) (b::s1, t1)) (o, bo)
(fn (s0, t0) => s0=t0) (bo, bo)
bo = bo
```

```
twin' bobo (fn(s0, t0) => s0=t0)
twin' obo (fn(s1, t1) => (fn(s0, t0) => s0=t0) (b::s1, t1))
twin' bo (fn(s2, t2) =>
          (fn(s1, t1) =>
            (fn(s0, t0) => s0=t0) (b::s1, t1)) (o::s2, t2))
(fn(s2, t2) =>
  (fn(s1, t1) =>
    (fn(s0, t0) => s0=t0) (b::s1, t1)) (o::s2, t2)) (nil, bo)
(fn(s1, t1) =>
  (fn(s0, t0) => s0=t0) (b::s1, t1)) (o, bo)
(fn(s0, t0) => s0=t0) (bo, bo)
bo = bo
true
```