# Programs as Data
## Continuations: exceptions, backtracking, micro-Icon

Peter Sestoft

Monday 2013-10-28

# Today

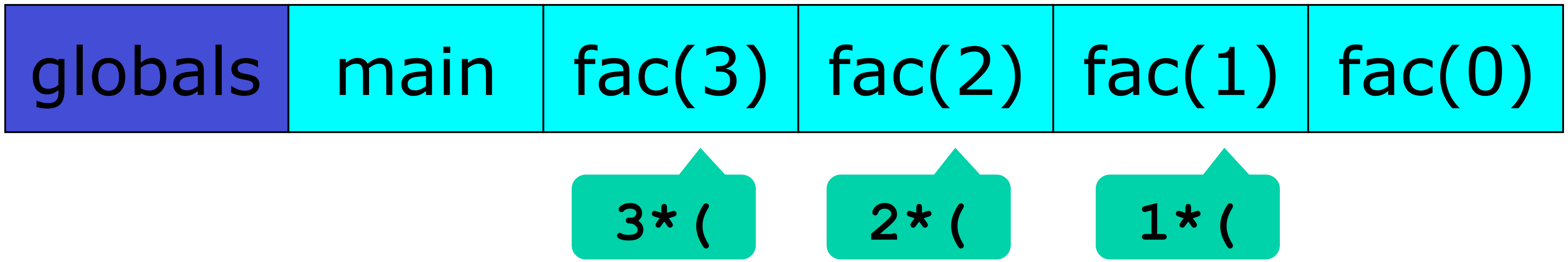
- Continuations
  - Tail-calls and accumulating parameters
  - Continuations and continuation-passing style
  - Continuation-based interpreters and exceptions
  - Continuations and backtracking

# Recursive functions
# Stack represents "continuation"

```
let rec facr n =
    if n=0 then 1
    else n * facr(n-1)
```

```
facr 3
==> 3 * facr 2
==> 3 * (2 * facr 1)
==> 3 * (2 * (1 * facr 0))
==> 3 * (2 * (1 * 1))
==> 3 * (2 * 1)
==> 3 * 2
==> 6
```

- One stack frame per recursive call
- Each stack frame represents “remember to *multiply result by n*”
- Together the stack frames represent the current *continuation*: “what must be done to the result of this expression”



# Continuations

- Continuation = "the rest of a computation"
- Continuation-passing style = CPS:
  - Every function has a continuation argument **`k`**
  - Do not return **`res`**; instead call **`k(res)`**
  - The continuation "knows what to do with **`res`**"

Normal

```
let rec facr n =
    if n=0 then 1
    else n * facr(n-1)
```

Continuation parameter

Don't return, instead call **`k`**

New continuation

CPS

```
let rec facc n k =
    if n=0 then k 1
    else facc (n-1) (fun v -> k(n * v))
```

# Uses of continuations

- A function in CPS can sometimes be rewritten to use an accumulating parameter, saving memory
- A function in CPS can sometimes stop the computation early, saving time
- An interpreter in CPS can model exceptions and exception handling such as **`try-catch`**
- Continuations can implement expressions with multiple results, as in Icon and Prolog
- Continuation-thinking helps on-the-fly optimization in the micro-C compiler (next week's lecture)
- Continuations can be used to structure web dialogs
- Continuations have many other more magical uses

CPS = continuation-passing style

# Deriving a CPS version of facr

```
let rec facr n =
    if n=0 then 1
    else n * facr(n-1)
```

```
let id = fun v -> v
```

facc n k = k(facr n)

facr n = facc n id

```
let rec facc n k =
    if n=0 then ...
    else ...
```

```
let rec facc n k =
    if n=0 then k 1
    else ...
```

```
let rec facc n k =
    if n=0 then k 1
    else facc (n-1) <new continuation>
```

```
let rec facc n k =
    if n=0 then k 1
    else facc (n-1) (fun v -> k(n * v))
```

# Evaluating facc n id

```
let rec facc n k =
    if n=0 then k 1
    else facc (n-1) (fun v -> k(n * v))

let id = fun v -> v
```

```
facc 3 id
==> facc 2 (fun v -> id(3 * v))
==> facc 1 (fun w -> (fun v -> id(3 * v)) (2 * w))
==> facc 0 (fun u -> (fun w -> (fun v -> id(3 * v)) (2 * w)) (1 * u))
==> (fun u -> (fun w -> (fun v -> id(3 * v)) (2 * w)) (1 * u)) 1
==> (fun w -> (fun v -> id(3 * v)) (2 * w)) (1 * 1)
==> (fun w -> (fun v -> id(3 * v)) (2 * w)) 1
==> (fun v -> id(3 * v)) (2 * 1)
==> (fu v -> id(3 * v)) 2
==> id(3 * 2)
==> id 6
==> 6
```

Uses no stack space!

Uses heap for contin's

# Joint exercise

- Given this function

```
let rec prod xs =
    match xs with
    | []     -> 1
    | x::xr -> x * prod xr
```

- Find the continuation-passing version, form:

```
let rec prodc xs k =
    match xs with
    | []     -> ...
    | x::xr -> ...
```

# Tail-recursive functions, iteration

- Rewrite facr with accumulating parameter r

```
let rec faci n r =
    if n=0 then r
    else faci (n-1) (r * n)
```

faci n r = r * (facr n)

facr n = faci n 1

```
faci 3 1
==> faci 2 3
==> faci 1 6
==> faci 0 6
==> 6
```

Uses no stack space!

# Continuations and accumulating parameters

- Both `(facc n k)` and `(faci n r)` are tail-recursive
- What relation between `k` and `r`?
- In fact, `k` always has form `fun u -> r*u`
  - To begin with, `k = (fun u -> u) = (fun u -> 1*u)`
  - If, inductively, `k` has form `k = fun u -> r*u`, then the new continuation

```
fun v -> k(n*v)
= fun v -> (fun u -> r*u)(n*v)
= fun v -> r*(n*v)
= fun v -> (r*n)*v
```

- So integer `r` is a simple way to represent function `k`
- All functions can be made tail-recursive
- Only some continuations can be represented simply

# Continuation-passing style in Java

- A Java function could be written in CPS also

```
interface Cont {
  int k(int v);
}
```

To represent functions `int->int`

```
static int facc(final int n, final Cont cont) {
  if (n == 0)
    return cont.k(1);
  else
    return facc(n-1,
                new Cont() {
                    public int k(int v) {
                      return cont.k(n * v);
                    }
                  });
}
```

The new continuation

Very ugly

# Why make continuations explicit?

- In normal code, the continuation is implicit:
  - The surrounding expressions
  - The next statement
  - The activation records on the stack
- By making the continuation explicit
  - We can ignore it, thus “avoid returning”
  - We can have two continuations, thus “choose how to return”
- Ignoring continuation = throwing exception
- Choosing a continuation is good for
  - handling exceptions, and
  - producing multiple results from an expression

# A simple functional language with exceptions

- Let's add exceptions to our small functional language:

```
type expr =
  | ...
  | Raise of exn                          // raise exn
  | TryWith of expr * exn * expr          // try e1 with exn -> e2
```

- Evaluation of an expression now either gives an integer result, or fails (aborts):

```
type answer =
  | Result of int
  | Abort of string
```

```
let rec coEval1 e env (cont : int -> answer) : answer =
```

# Interpreter with continuation, for *throwing* exceptions, part 1

```
let rec coEval1 e env (cont : int -> answer) : answer =
    match e with
    | CstI i -> cont i
    | Var x  ->
      match lookup env x with
        | Int i -> cont i
        | _      -> Abort "coEval1 Var"
    | Prim(ope, e1, e2) ->
      coEval1 e1 env
       (fun i1 ->
          coEval1 e2 env
            (fun i2 ->
             match ope with
               | "*" -> cont(i1 * i2)
               | "+" -> cont(i1 + i2)
               | ... ))
    | Raise (Exn s) -> Abort s
```

Compare lecture 4

# Interpreter with continuation, for *throwing* exceptions, part 2

```
let rec coEval1 e env (cont : int -> answer) : answer =
    match e with
    | ...
    | If(e1, e2, e3) ->
      coEval1 e1 env
              (fun b -> if b<>0 then
                          coEval1 e2 env cont
                        else
                          coEval1 e3 env cont)
    | ...
```

# Interpretation of exception *handling*

- Add an error continuation to interpreter:
  econt : exn -> answer
- To throw exception, call error continuation instead of normal continuation
- The error continuation looks at the exception and decides whether it wants to handle it

# Interpreter with two continuations for throwing and handling (part)

```
let rec coEval2 e env (cont : int -> answer)
                      (econt : exn -> answer) : answer =
    match e with
    | CstI i -> cont i
    | If(e1, e2, e3) ->
      coEval2 e1 env (fun b ->
                       if b<>0 then
                         coEval2 e2 env cont econt
                       else
                         coEval2 e3 env cont econt)
                      econt
    | ...
    | Raise exn -> econt exn
    | TryWith (e1, exn, e2) ->
      let econt1 thrown =
          if thrown = exn then coEval2 e2 env cont econt
                               else econt thrown
      in coEval2 e1 env cont econt1
```

# Expressions that give multiple results; the Icon language

| Expression | Result seq. | Print | Comment |
|---|---|---|---|
| 5 | 5 | | Constant |
| write 5 | 5 | 5 | Constant, side effect |
| (1 to 3) | 1 2 3 | | Range, 3 results |
| write (1 to 3) | 1 2 3 | 1 | Side effect |
| every (write (1 to 3)) | 0 | 1 2 3 | Force all results |
| (1 to 0) | | | Empty range, no res. |
| &fail | | | No results |
| (1 to 3)+(4 to 6) | 5 6 7 6 7 8 7 8 9 | | All combinations |
| 3 < 4 | 4 | | Comparison success |
| 4 < 3 | | | Comparison fails |
| 3 < (1 to 5) | 4 5 | | Success twice |
| (1 to 3) \| (4 to 6) | 1 2 3 4 5 6 | | Each left, each right |
| (1 to 3) & (4 to 6) | 4 5 6 4 5 6 4 5 6 | | Each right for each left |
| (1 to 3) ; (4 to 6) | 4 5 6 | | No backtracking to left |

# Micro-Icon interpreter

- The interpreter takes two continuations:
  - A *failure continuation*

    **`econt : unit -> answer`**

    called when there are no (more) results
  - A *success continuation*

    **`cont : value -> fcont -> answer`**

    called when there is one (more) result
- The econt argument to cont can be called by cont to ask for more results

```
let rec eval (e : expr) (cont : cont) (econt : econt) =
    match e with
    | CstI i -> cont (Int i) econt
    | ...
    | Fail -> econt ()
```

# Micro-Icon interpreter, part 1

```
let rec eval (e : expr) (cont : cont) (econt : econt) =
    match e with
    | CstI i -> cont (Int i) econt
    | CstS s -> cont (Str s) econt
    | Prim(ope, e1, e2) ->
      eval e1 (fun v1 -> fun econt1 ->
          eval e2 (fun v2 -> fun econt2 ->
              match (ope, v1, v2) with
                | ("+", Int i1, Int i2) ->
                  cont (Int(i1+i2)) econt2
                | ("*", Int i1, Int i2) ->
                  cont (Int(i1*i2)) econt2
                | ("<", Int i1, Int i2) ->
                  if i1<i2 then
                      cont (Int i2) econt2
                  else
                      econt2 ()
                | _ -> Str "unknown prim2")
              econt1)
          econt
    | ...
```

5

2+2

# Micro-Icon interpreter, part 2

```
let rec eval (e : expr) (cont : cont) (econt : econt) =
    match e with
    | ...
    | FromTo(i1, i2) ->
      let rec loop i =
          if i <= i2 then
              cont (Int i) (fun () -> loop (i+1))
          else
              econt ()
      in loop i1
    | Write e ->
      eval e (fun v ->
              fun econt1 -> (write v; cont v econt1))
             econt
    | If(e1, e2, e3) ->
      eval e1 (fun _ -> fun _ -> eval e2 cont econt)
              (fun () -> eval e3 cont econt)
    | ...
```

(1 to 3)

write 5

if 3<4 ...

# Micro-Icon interpreter, part 3

```
let rec eval (e : expr) (cont : cont) (econt : econt) =
    match e with
    | ...
    | And(e1, e2) ->
      eval e1 (fun _ -> fun econt1 -> eval e2 cont econt1) econt

    | Or(e1, e2) ->
      eval e1 cont (fun () -> eval e2 cont econt)

    | Seq(e1, e2) ->
      eval e1 (fun _ -> fun econt1 -> eval e2 cont econt)
              (fun () -> eval e2 cont econt)

    | Every e ->
      eval e (fun _ -> fun econt1 -> econt1 ())
             (fun () -> cont (Int 0) econt)
```

e1&e2

e1|e2

e1;e2

every e

Take the result, ignore it, and ask for one more

When no more results, succeed with result 0

# Reading and homework

- This week's lecture:
  - PLCSD chapter 11
  - Exercises 11.1, 11.2, 11.3, 11.4, 11.8

- Next week:
  - PLCSD chapter 12