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”

<table>
<thead>
<tr>
<th>globals</th>
<th>main</th>
<th>fac(3)</th>
<th>fac(2)</th>
<th>fac(1)</th>
<th>fac(0)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>3*()</td>
<td>2*()</td>
<td>1*()</td>
<td></td>
</tr>
</tbody>
</table>

Continuations

- Continuation = “the rest of a computation”
- Continuation-passing style = CPS:
  - Every function has a continuation argument \( k \)
  - Do not return \( \text{res} \); instead call \( k(\text{res}) \)
  - The continuation “knows what to do with \( \text{res} \)”

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

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 \texttt{facr}

```ml
let rec facr n =  
  if n=0 then 1  
  else n \times \texttt{facr}(n-1)

let id = fun v \rightarrow v

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

facc n k = k(\texttt{facr} n)

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

faccr n = facc n id

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 \rightarrow k(n \times 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)
==> (fun 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

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

• Find the continuation-passing version, form:

```ml
let rec prodc xs k =  
  match xs with  
  | []    -> ...  
  | x::xr -> ...
```
Tail-recursive functions, iteration

• Rewrite facr with accumulating parameter r

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

let facr n = faci n 1

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

facr n = faci n 1
```

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

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

Uses no stack space!
Continuations and 
accumulating parameters

• Both \( (\text{facc } n \ k) \) and \( (\text{faci } n \ r) \) are tail-recursive
• What relation between \( k \) and \( r \)?
• In fact, \( k \) always has form \( \text{fun } u \rightarrow r*u \)
  – To begin with, \( k = (\text{fun } u \rightarrow u) = (\text{fun } u \rightarrow 1*u) \)
  – If, inductively, \( k \) has form \( k = \text{fun } u \rightarrow r*u \),
    then the new continuation
      \[
      \text{fun } v \rightarrow k(n*v)
      = \text{fun } v \rightarrow (\text{fun } u \rightarrow r*u)(n*v)
      = \text{fun } v \rightarrow r*(n*v)
      = \text{fun } v \rightarrow (r*n)*v
      \]
• 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);
}

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);
                        }
                    });
}
```

To represent functions int->int

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:

```ocaml
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):

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

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

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result seq.</th>
<th>Print</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5</td>
<td></td>
<td>Constant</td>
</tr>
<tr>
<td>write 5</td>
<td>5</td>
<td>5</td>
<td>Constant, side effect</td>
</tr>
<tr>
<td>(1 to 3)</td>
<td>1 2 3</td>
<td></td>
<td>Range, 3 results</td>
</tr>
<tr>
<td>write (1 to 3)</td>
<td>1 2 3</td>
<td>1</td>
<td>Side effect</td>
</tr>
<tr>
<td>every (write (1 to 3))</td>
<td>0</td>
<td>1 2 3</td>
<td>Force all results</td>
</tr>
<tr>
<td>(1 to 0)</td>
<td></td>
<td></td>
<td>Empty range, no res.</td>
</tr>
<tr>
<td>&amp;fail</td>
<td></td>
<td></td>
<td>No results</td>
</tr>
<tr>
<td>(1 to 3)+(4 to 6)</td>
<td>5 6 7 6 7 8 7 8 9</td>
<td></td>
<td>All combinations</td>
</tr>
<tr>
<td>3 &lt; 4</td>
<td>4</td>
<td></td>
<td>Comparison success</td>
</tr>
<tr>
<td>4 &lt; 3</td>
<td></td>
<td></td>
<td>Comparison fails</td>
</tr>
<tr>
<td>3 &lt; (1 to 5)</td>
<td>4 5</td>
<td></td>
<td>Success twice</td>
</tr>
<tr>
<td>(1 to 3)</td>
<td>(4 to 6)</td>
<td>1 2 3 4 5 6</td>
<td>Each left, each right</td>
</tr>
<tr>
<td>(1 to 3) &amp; (4 to 6)</td>
<td>4 5 6 4 5 6 4 5 6</td>
<td></td>
<td>Each right for each left</td>
</tr>
<tr>
<td>(1 to 3) ; (4 to 6)</td>
<td>4 5 6</td>
<td></td>
<td>No backtracking to left</td>
</tr>
</tbody>
</table>
Micro-Icon interpreter

• The interpreter takes two continuations:
  – A failure continuation
    \[ \text{econt} : \text{unit} \rightarrow \text{answer} \]
    called when there are no (more) results
  – A success continuation
    \[ \text{cont} : \text{value} \rightarrow \text{fcont} \rightarrow \text{answer} \]
    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 ()
```
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
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)
  | ... |
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)

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