

**Programming Languages, F2003**  
**Lecture 8, Wednesday 26 March 2003**

- Continuations: describing 'the rest of the computation'
- Functions in continuation-passing style
- From continuation to accumulating parameter
- Continuation-passing interpreter for a functional language
- Continuation-based implementation of exceptions
- Continuation-passing interpreter for an imperative language
- Continuation-passing interpreter for a language with backtracking

**Continuations**

A *continuation* is an explicit representation of the rest of the computation.

Continuations have many uses, practical and conceptual:

- rewriting a function in continuation-passing style makes its evaluation order explicit;
- a function in continuation-passing style can sometimes be rewritten in tail-recursive form, saving space;
- a function in continuation-passing style can sometimes stop the computation early, saving time;
- the evaluation stack (in micro-C, for example) can be considered a representation of a continuation;
- an interpreter in continuation-passing style helps understand the notion of tail-call;
- an interpreter in continuation-passing style can model exceptions and exception handling (try-catch);
- continuations are used to implement backtracking, as in the languages Prolog and Icon;
- continuations can be used to understand on-the-fly optimization in the micro-C compiler (lecture 9);
- continuations have many other and more magical uses ...

Continuations were invented independently by several people around 1970; see Reynolds' 1993. The name is due to Christopher Wadsworth, a student of Christopher Strachey (values, CPL, etc).

**Evaluation of the recursive factorial function**

The function `factr n` computes  $n! = n \cdot (n-1) \cdot \dots \cdot 2 \cdot 1$  by recursion:

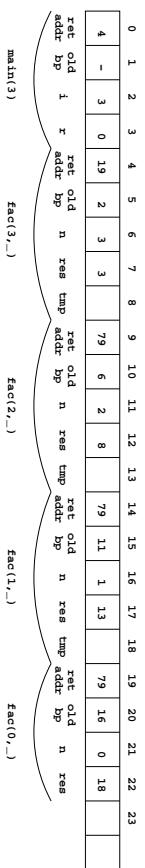
```
fun factr n =
  if n=0 then 1 else n * factr(n-1);
```

After the recursive call `factr (n-1)`, we must multiply the result by `n`. Computing `factr 3`:

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

Some space is needed to remember the work (the multiplications) that must be done after the recursive call.

Just like micro-C evaluation:



**The factorial function in continuation-passing style**

We can make 'the work that must be done after the recursive call' explicit as a function, called a *continuation*.

The continuation `k : int -> int` represents the rest of the computation.

The factorial function in continuation-passing style:

```
fun factc n k =
  if n=0 then k 1 else factc (n-1) (fn v => k(n * v))
```

We have `factc n k = k(factr n)`; in particular `factc n id = factr n`, where `fun id v = v`.

Computing `factc 3 id` goes like this:

```
factc 3 id
==> factc 2 (fn v => id(3 * v))
==> factc 1 (fn w => (fn v => id(3 * v)) (2 * w))
==> factc 0 (fn u => (fn w => (fn v => id(3 * v)) (2 * w)) (2 * u)) (1 * u)) 1
==> (fn w => (fn v => id(3 * v)) (2 * w)) (1 * 1)
==> (fn w => (fn v => id(3 * v)) (2 * w)) 1
==> (fn v => id(3 * v)) (2 * 1)
==> (fn v => id(3 * v)) 2
==> id(3 * 2)
==> id 6
==> 6
```

### Continuation-passing style in Java (file `cont/Factorial1.java`)

Continuation-passing style is not particular to Standard ML.

But the Java version of continuation-passing style `FactCC` is a little complicated at first sight.

A continuation is an instance of a local inner class that implements `Cont`:

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

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

This looks strange, but the idea is well-known: an `ActionListener` is a kind of continuation.

### The iterative factorial function

```
fun fact1 n r =
  if n=0 then r else fact1 (n-1) (r * n);
```

The evaluation of `fact1 3 1` goes like this:

```
fact1 3 1
==> fact1 2 3
==> fact1 1 6
==> fact1 0 6
==> 6
```

This computation can be done in constant space: there are no pending multiplications to remember.

The reason is that the recursive call to `fact1` is a *tail call*: it is the last action of the function.

Consequence: no space is needed to remember the work that must be done after the recursive call.

Transformation to continuation-passing style makes all calls into tail calls. (Why?)

This in itself saves nothing: the continuations must be stored.

In some cases a continuation can be represented by a value (integer, list, ...), saving space or time.

### From continuation to accumulating parameter

Apparently little is gained by writing factorial in continuation-passing style.

However, in the special case of factorial we can represent the continuation by a simple number.

**Observation:** In `FactCC`, the continuation always has the form `fn u => r * u` for some integer `r`:

- The initial continuation `id` can be written as `fn u => 1 * u`;
- If the old continuation `k` can be written as `fn u => r * u` then the new continuation `(fn v => k(n * v))` can be written as `fn v => (r * n) * v`. This is because `k(n * v) = r * (n * v) = (r * n) * v`.

If we replace `k` by `r` and replace `k(e)` by `r*e` in `FactCC` we get an iterative version of factorial:

```
fun fact1 n r =
  if n=0 then r * 1 else fact1 (n-1) (r * n)
```

It holds that `fact1 n 1 = factc n id = factr n`.

The parameter `r` is called an *accumulating parameter*: it accumulates the result of the function.

### Tail calls

A function call is a *tail call* if it is the last action of the calling function.

An expression is in *tail position* if the evaluation of that expression is the last action of the enclosing function.

Expression	Status of subexpressions
Let $x=e_1$ in $e_2$ end	$e_2$ is in tail position, $e_1$ is not
$e_1+e_2$	neither $e_1$ nor $e_2$ is in tail position
if $e_1$ then $e_2$ else $e_3$	$e_2$ and $e_3$ are in tail position, $e_1$ is not
Let $f x=e_1$ in $e_2$ end	$e_2$ is in tail position, $e_1$ is not
$f e$	$e$ is not in tail position

A tail call is a call that is in tail position. Which of these calls are tail calls:

```
F 1
f(F 1)
F 1 + F 2
if 1=2 then F 3 else f(F 4)
let x = f 1 in f x end
let x = f 1 in if x=2 then f x else f 3 end
```

### The importance of tail calls

Most functional language implementations execute tail-recursive functions in constant space.

Most other language implementations (Pascal, C, C++, Java, C#, ...) do not.

Typically they use stack space proportional to the depth of recursive calls.

Using the lecture 7 compiler, this micro-C program (file `imp/ex12.c`) runs out of stack space for `n > 350`:

```
int main(int n) {
    if (n)
        return main(n-1);
    else
        return 17;
}
```

Next week we shall improve the micro-C compiler so that tail-recursive functions execute in constant space.

In Java it is difficult to compile tail calls properly because of its security model:

The security manager checks the frame stack to see that all calling methods have sufficient permissions.

If the stack calling method's stack frame were discarded in a tail-call, then a low-permission method could cheat the security manager by calling a high-permission method by a tail call ...

Microsoft .Net CLR has adopted the same security model, but makes some effort to be tailcall-friendly.

F7C

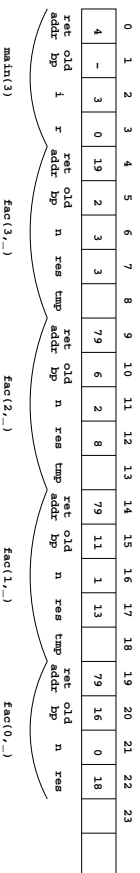
### The evaluation stack represents a continuation

The stack frame on the stack top binds the variables of the currently executing function.

The stack frames below it represent function calls that are not yet finished:

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

The pending multiplications (by 3, 2, and 1) are represented by the stack frames below the top-most one:



The stack frames remember what must be done when the current function returns.

F7C

### Continuation-passing interpreters

A interpreter is a function:

```
fun eval (e : expr) (env : vFenv) : int = ...
```

Every function can be rewritten in continuation-passing style.

So an interpreter can be written in continuation-passing style.

This is interesting:

- if the interpreter is allowed to ignore its continuation — similar to throwing of an exception, or
- if the interpreter is given two continuations and can choose which one to use — similar to handling an exception.

F7C

### Continuation-passing interpreter for a functional language

Recall our simple functional language:

```
datatype expr =
  CstI of int
| GcstB of bool
| Var of string
| Let of string * expr * expr
| Prim of string * expr list
| If of expr * expr * expr
| Letfun of string * string * expr * expr (* (f, x, body, ebody) *)
| Call of string * expr
```

An interpreter for this language was a function:

```
fun eval (e : expr) (env : vFenv) : int = ...
```

This function can be rewritten in continuation-passing style, with a continuation `cont : int -> answer`.

The continuation takes the value of a subexpression and produces the final result of the interpretation:

```
fun ccallv1 (e : expr) (env : vFenv) (cont : int -> answer) : answer = ...
```

An answer is `Success` (normal termination) or `Failure` (exceptional termination):

```
datatype answer =
  Success of int
| Failure of string
```

F7C

### The interpreter, part 1 (file contc/Fun.sml)

```
fun coEval1 (e : expr) (env : vFenv) (cont : int -> answer) : answer =
  case e of
  CstI i => cont i
| CstB b => cont (if b then 1 else 0)
| Var x  => (case lookup env x of
             | _ => Failure "coEval1 Var")
             | _ => cont i
             Int i => cont i)
| Let(x, exhs, ebody) =>
  coEval1 exhs env (fn xval =>
    let val env1 = bind1 env (x, Int xval)
    in coEval1 ebody env1 cont end)
| Prim(ope, [e1, e2]) =>
  coEval1 e1 env
  (fn i1 =>
   coEval1 e2 env
   (fn i2 =>
    case ope of
    "*" => cont(i1 * i2)
    "+" => cont(i1 + i2)
    "-" => cont(i1 - i2)
    "=" => cont(if i1 = i2 then 1 else 0)
    "<" => cont(if i1 < i2 then 1 else 0)
    _   => Failure "unknown primitive"))
| Prim(ope, _) => Failure "primitive arity"
| ...
```

FC Programming Languages, F2003

Page 8-13

### The interpreter, part 2 (file contc/Fun.sml)

```
fun coEval1 (e : expr) (env : vFenv) (cont : int -> answer) : answer =
  case e of
  ...
| LetFun(f, x, fbody, ebody) =>
  let val env1 = bind1 env (f, RClO(f, x, fbody, env))
  in coEval1 ebody env1 cont end
| Call(f, args) =>
  (case lookup env f of
   fclOure as RClO (f, x, fbody, env1) =>
     coEval1 args env
     (fn argv =>
      let val env2 = bind1 env1 (f, fclOure)
          val env3 = bind1 env2 (x, Int argv)
          in coEval1 fbody env3 cont end)
   | _ => Failure "coEval1 Call")
| If(e1, e2, e3) =>
  coEval1 e1 env
  (fn b => if b <> 0 then coEval1 e2 env cont
           else coEval1 e3 env cont)
fun eval1 e env = coEval1 e env (fn v => Success v)
```

When the interpreter fails (in Var, Prim or Call) it simply ignores the continuation, and returns Failure.

It terminates abruptly without using meta-language (SML) exceptions!

A major advantage of continuation-passing style:

When the continuation cont is explicit, we can terminate the computation abruptly just by ignoring it.

FC Programming Languages, F2003

Page 8-14

### Throwing exceptions in a functional language

Now we can add exceptions and a raise expression to our functional language.

```
datatype exn =
  Exn of string
datatype expr =
  ...
  | Raise of exn
  | Handle of expr * exn * expr
  (* e1 handle exn => e2 *)
```

The expression Raise (Exn s) should simply terminate the interpreter.

It does so by ignoring the exception and returning Failure s as the answer:

```
fun coEval1 (e : expr) (env : vFenv) (cont : int -> answer) : answer =
  case e of
  ...
  | Raise (Exn s) => Failure s
```

FC

Programming Languages, F2003

Page 8-15

### How can we model the handling of exceptions?

In SML, exceptions thrown in e1 can be handled by (e1 handle exn => e2).

- If e1 terminates successfully, then its result is the result of the entire expression.
  - If e1 throws an exception, then
    - if the exception matches exn then the exception is discarded and e2 is evaluated;
    - otherwise the exception propagates out, either to a previous handler or all the way to the top-level.
- This is just as in Java's and C#'s try stmt1 catch (exn) stmt2.

To model this, we need two continuations:

- the normal (success) continuation cont : int -> answer;
- an error continuation econt : exn -> answer.

The error continuation looks at the exception given to it, and either handles it or propagates it.

The new exception-throwing and exception-handling interpreter has type:

```
fun coEval2 (e : expr) (env : vFenv)
  (cont : int -> answer) (econt : exn -> answer) : answer = ...
```

FC

Programming Languages, F2003

Page 8-16

### Exception-handling functional interpreter, part 1 (file contc/Fun.sml)

```
fun coEval2 (e : expr) (env : vFenv)
  (cont : int -> answer) (econt : exn -> answer) : answer =
  case e of
  CstI i => cont i
| CstB b => cont (if b then 1 else 0)
| Var x => (case Lookup env x of
  Int i => cont i
  | _ => Failure "coEval2 Var")
| Let(x, exhs, ebody) =>
  coEval2 exhs env (fn xval =>
    let val env1 = bind1 env (x, Int xval)
    in coEval2 ebody env1 cont econt end)
  econt
| Prim(ope, [e1, e2]) =>
  coEval2 e1 env
  coEval2 e2 env
  (fn i2 =>
    case ope of
    "*" => cont(i1 * i2)
    "+" => cont(i1 + i2)
    "-" => cont(i1 - i2)
    "=" => cont(if i1 = i2 then 1 else 0)
    "<" => cont(if i1 < i2 then 1 else 0)
    _ => Failure "unknown primitive") econt) econt
| Prim(ope, _) => Failure "primitive arity"
| ...
```

### Expressions in tail position, and the continuation-passing interpreter

The continuation-passing interpreter shows which expressions are in tail position.

A subexpression is in tail position if it is evaluated with the same continuations as the enclosing expression.

In particular, in (e1 handle exn => e2), the subexpression e1 is *not* in tail position.

Subexpression e1 has a different error continuation than the enclosing expression.

### Why can tail calls be evaluated without a new stack frame?

A function call Call(F, earg) evaluates the body of F with the same continuations as the call itself:

```
fun coEval2 (e : expr) (env : vFenv) (cont : int -> answer) econt =
  case e of
  ...
  | Call(F, earg) => ... coEval2 fbody env3 cont econt ...
```

Hence if the call to F is a tail call, evaluated with the same continuations as the enclosing function

```
let g x = ... f e ...
```

then the body of the called F is evaluated with the same continuations as the calling G also.

So the evaluation of F can reuse the continuations of G.

In a stack machine, the evaluation of F can use the same stack as G: no new stack frame is needed.

### Exception-handling functional interpreter, part 1 (file contc/Fun.sml)

```
fun coEval2 (e : expr) (env : vFenv)
  (cont : int -> answer) (econt : exn -> answer) : answer =
  ...
| LetFun(F, x, fbody, ebody) =>
  let val env1 = bind1 env (F, RClO(F, x, fbody, env))
  in coEval2 ebody env1 cont econt end
| Call(F, earg) =>
  (case Lookup env f of
  fclosure as RClO (f, x, fbody, env1) =>
    coEval2 earg env
    (fn argv =>
      let val env2 = bind1 env1 (f, fclosure)
      val env3 = bind1 env2 (x, Int argv)
      in coEval2 fbody env3 cont econt end) econt)
  | _ => Failure "coEval2 Call")
| If(e1, e2, e3) =>
  coEval2 e1 env (fn b =>
    if b > 0 then coEval2 e2 env cont econt
    else coEval2 e3 env cont econt) econt
| Raise exn => econt exn
| Handle(e1, exn, e2) =>
  let fun econt1 exn1 =
    if exn1 = exn then coEval2 e2 env cont econt
    else econt exn1
  in coEval2 e1 env cont econt1 end
fun eval2 e env =
  in coEval2 e1 env cont econt1 end
  (fn v => Success v)
  (fn (Exn s) => Failure ("Uncaught exception: " ^ s))
```

### Continuation-passing interpreter for an imperative language

A continuation-passing interpreter can be defined for a small imperative language:

```
datatype stmt =
  | Assign of string * expr
  | If of expr * stmt * stmt
  | Block of stmt list
  | For of string * expr * expr * stmt
  | While of expr * stmt
  | Print of expr
  | Throw of exn
  | TryCatch of stmt * exn * stmt
```

The statement interpreter will have type:

```
fun coExec1 stmt sto (cont : sto -> answer) : answer = ...
```

A continuation cont takes the store resulting from the execution of a statement, and produces an answer:

```
datatype answer =
  | Success
  | Failure of string
```

It also permits exception-throwing (and handling, not shown here).

### Continuation-passing Interpreter, part 1 (file cont/imp.sml)

```
fun coExec1 stmt sto (cont : sto -> answer) : answer =
  case stmt of
  | Asgn(x, e) =>
    cont (set sto (x, eval e sto))
  | If(e1, stmt1, stmt2) =>
    if eval e1 sto <= 0 then
      coExec1 stmt1 sto cont
    else
      coExec1 stmt2 sto cont
  | Block stmts =>
    let fun loop []      sto = cont sto
        | loop (s1::sr) sto = coExec1 s1 sto (fn sto => loop sr sto)
    in loop stmts sto end
  | For(x, estart, estop, body) =>
    let val start = eval estart sto
        val stop = eval estop sto
    in fun loop i sto =
        if i <= stop then
          coExec1 body (set sto (x, i)) (fn sto => loop (i+1) sto)
        else
          cont sto
    in loop start sto end
  | ...
  in loop start sto end
```

No continuation is passed to the expression evaluator.

This limitation of the model means that expression evaluation cannot throw exceptions.

FIG

### Continuation-passing Interpreter, part 2 (file cont/imp.sml)

```
fun coExec1 stmt sto (cont : sto -> answer) : answer =
  ...
  | While(e, body) =>
    let fun loop sto =
        if eval e sto <= 0 then
          coExec1 body sto loop
        else
          cont sto
    in loop sto end
  | Print e =>
    (print (Int.toString (eval e sto)); print "\n"; cont sto)
  | Throw (Exn s) =>
    Failure ("Uncaught exception: " ^ s)
  fun run1 stmt : answer =
    coExec1 stmt empty (fn sto => Success)
```

An exception-handling interpreter coExec2 can be defined just as for a functional language.

An error continuation eCont : exn \* sto -> answer must be passed around.

It takes both a throw exception and a possibly updated store as argument

The throwing of an exception does not discard updates to variables that have already been made.

FIG

### A Backtracking Interpreter (cont/backtrack.sml)

In the language Icon, an expression may fail, produce one result, or produce many results:

Expression	Results	Comment
5	5	Integer constant
(1 to 3)	1 2 3	Range: succeeds 3 times
(1 to 0)		Empty range: fails
write (1 to 3)	1 2 3	Print the values
every(write (1 to 3))	0	Force all results to be generated and printed
(1 to 3) + (4 to 6)	5 6 7 6 7 8 7 8 9	All argument combinations
(1 to 3)   (4 to 6)	1 2 3 4 5 6	Results from left, and results from right
(1 to 3) & (4 to 6)	4 5 6 4 5 6 4 5 6	All results from right for every result from left
(1 to 3) ; (4 to 6)	4 5 6	All results from right, no backtracking left

This can be implemented using two continuations:

- A failure continuation fCont : unit -> answer
- A success continuation cont : int -> fCont -> answer

The success continuation's failure continuation is used for backtracking: go back and ask for more results.

FIG

```
fun eval (e : expr) (cont : cont) (fcont : fcont) =
  case e of
  | CstI i => cont i fcont (* i1 to i2 *)
  | FromTo(i1, i2) =>
    let fun loop i =
        if i <= i2 then
          cont i (fn () => loop (i+1))
        else
          fcont ()
    in loop i1 end (* write e *)
  | Write e =>
    eval e (fn v => fn fcont1 => (write v; cont v fcont1))
  | Prim2(ope, e1, e2) => (* e1 + e2 *)
    eval e1 (fn v1 => fn fcont1 =>
      eval e2 (fn v2 => fn fcont2 =>
        case "ope" of
        | "+" => cont (v1+v2) fcont2
        | "*" => cont (v1*v2) fcont2
        | _ => Failure "unknown prim2")
      fcont1)
  | And(e1, e2) => fcont (* e1 & e2 *)
    eval e1 (fn _ => fn fcont1 => eval e2 cont fcont1)
  | Or(e1, e2) => fcont (* e1 | e2 *)
    eval e1 cont
  | Seq(e1, e2) => (* e1 ; e2 *)
    eval e1 (fn _ => fn fcont1 => eval e2 cont fcont)
  | ...
```

FIG