

Programming Languages, F2002
Lecture 8, Wednesday 3 April 2002

- 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

TF-C

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 Prolog and Icon languages;
- 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).

TF-C

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)` has been computed, we must multiply the result by `n`.

Thus computing `factr 3` goes like this:

```
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.

This is just like the evaluation of factorial in micro-C (`!mp/ex9.c`) by a stack machine.

TF-C

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

TF-C

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 `FactC`, 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 `FactC` we get an iterative version of factorial:

```
fun facti n r =  
  if n=0 then r * 1 else facti (n-1) (r * n)  
It holds that facti n 1 = factc n id = factr n.
```

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

FC

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 $x=e_r$ in e_b end	e_b is in tail position, e_r 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 $f x=e_r$ in e_b end	e_b is in tail position, e_r 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
```

FC

The iterative factorial function

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

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

```
facti 3 1  
==> facti 2 3  
==> facti 1 6  
==> facti 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 `facti` 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.

FC

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 and C# it is difficult to compile tail calls properly because of their (flawed) security model.

FC

The evaluation stack as 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
==> 6
```

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

The purpose of those stack frames is to remember what must be done when the current function returns.

The evaluation stack can be understood as a representation of the continuation.

FC

Continuation-passing interpreter for a functional language

Recall our simple functional language:

```
datatype expr =
  CStrI of int
| CStrB 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, fbody, 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 coEval1 (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
```

FC

The interpreter, part 1 (file `cont/fun.sml1`)

```
fun coEval1 (e : expr) (env : vFenv) (cont : int -> answer) : answer =
  case e of
  CStrI i => cont i
| CStrB b => cont (bool2int b)
| Var x => (case lookup env x of
  Int i => cont i
  | _ => Failure "coEval1 Var")
| Let(x, ehns, ebody) =>
  coEval1 ehns env (fn xvval =>
    let val env1 = bind1 env (x, Int xvval)
    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(bool2int (i1 = i2))
      "<" => cont(bool2int (i1 < i2))
      _ => Failure "unknown primitive")
    )
  )
| Prim(ope, _) => Failure "primitive arity"
  ...
```

FC

The interpreter, part 2 (file `cont/fun.sml1`)

```
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, env1))
    in coEval1 ebody env1 cont end
  | Call(F, args) =>
    (case lookup env F of
    fclosure as RClO (F, x, fbody, env1) =>
      coEval1 args env
      (fn argv =>
        let val env2 = bind1 env1 (F, fclosure)
        val env3 = bind1 env2 (x, Int argv)
        in coEval1 fbody env3 cont end)
    )
  | If(e1, e2, e3) =>
    coEval1 e1 env
    (fn b => if Int2bool b 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

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, F2002

Page 8-13

How can we model the handling of exceptions?

In SML exceptions 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, 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 usual (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, F2002

Page 8-14

Exception-handling functional interpreter, part 1 (file `cont/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 (bool2int b)  
  | Var x => (case lookup env x of  
    Int i => cont i  
    | _ => Failure "coEval2 Var")  
  | Let(x, ehns, ebody) =>  
    coEval2 ehns env (fn xv1 =>  
      let val env1 = bind1 env (x, Int xv1)  
        in coEval2 ebody env1 cont econt end)  
    econt  
  | Prim(ope, [e1, e2]) =>  
    coEval2 e1 env  
    (fn i1 =>  
      coEval2 e2 env  
      (fn i2 =>  
        case ope of  
        "*" => cont(i1 * i2)  
        | "+" => cont(i1 + i2)  
        | "-" => cont(i1 - i2)  
        | "=" => cont(bool2int (i1 = i2))  
        | "<" => cont(bool2int (i1 < i2))  
        | _ => Failure "unknown primitive") econt) econt  
  | Prim(ope, _) => Failure "primitive arity"  
  ...
```

FC

Programming Languages, F2002

Page 8-15

Exception-handling functional interpreter, part 1 (file `cont/fun.sml`)

```
fun coEval2 (e : expr) (env : vFenv) (cont : int -> answer) econt =  
  case e of  
  ...  
  | 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, args) =>  
    (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 Int2bool b 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 =  
    coEval2 e env (fn v => Success v)  
    (fn (Exn s) => Failure ("Uncaught exception: " ^ s))
```

FC

Programming Languages, F2002

Page 8-16

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 `(e1 handle expr => 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)` case evaluates the body of `F` with the same continuations as the call itself:

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

Hence if the call 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.

TC

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 expr
  | 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).

TC

Continuation-passing interpreter, part 1 (file `cont/imp.sml`)

```
fun coExec1 stmt sto (cont : sto -> answer) : answer =
  case stmt of
  | Assign(x, e) =>
      cont (set sto (x, eval e sto))
  | If(e1, stmt1, stmt2) =>
      if Int2bool(eval e1 sto) 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
          fun loop i sto =
              if i <= stop then
                coExec1 body (set sto (x, i)) (loop (i+1))
              else
                cont sto
          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.

TC

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 Int2bool(eval e sto) 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.

TC