Continuations (continued)

November 8, 2001

CS 131: Programming Languages

Continuations

- Generally, the *continuation* of an piece of code being computed is "everything that will happen after this code completes"
- The idea of continuation is very abstract, but has turned out to be very useful (particularly in the theory of programming languages)
  - Reynolds [*The Discoveries of Continuations*, 1993] lists 7 computer scientists who independently came up with this idea (between 1964 and 1973)!

Continuations Example

- For example, consider the following code

```
3 + ( 4*5 - 7)
```

- What is the continuation of `4*5` when this code is executed?

Continuations Example

- Continuation is a *run-time* concept

```
let
   fun fact 0 = 1
     | fact n = n * fact(n-1)
in
   fact 4
end
```

- What if the body of the `let` were
  `fact(4) + fact(2)` ?

Representing Continuations

- Normally the continuation is implicit:
  - The CPU's program counter/instruction pointer
  - The contents of the run-time stack
    - Especially the return-addresses on the stack!
- But, it is occasionally useful to make the continuation explicit.
  - By representing it as a continuation function
  - As a primitive concept

Review: Continuation Functions

- A *continuation function* is an extra argument that says what is to be done once we have a result

```
mult :  int list -> int
```

↓

```
mult :  int list * (int -> ans) -> ans
```

(ans is the result type of the entire program)

- Represents "the rest of the computation"
  - What we're supposed to do after the current step

Continuation Arguments

- Can write code so that *every* function takes a continuation argument:

```
plus  : int * int * (int -> ans) -> ans
times : int * int * (int -> ans) -> ans

(* given a,b,c, and d, compute ab+cd *)
fun f(a, b, c, d, k) =
   times(a, b,
         fn x => times(c, d,
                       fn y => plus(x, y, k)))
```

- This is called *continuation-passing style* (CPS)

Another Example: `fact`

```
fun fact 0 = 1
  | fact n = n * fact(n-1)
```

↓

```
fun fact'(0, k) = k 1
  | fact'(n, k) = fact'(n-1, fn a => k(n*a))
```

or

```
k o (fn a => n * a)
```

or

```
fn a => times(n,a,k)
```

Another Example: `fib`

```
fun fib 0 = 1
  | fib 1 = 1
  | fib n = fib(n-1) + fib(n-2)
```

↓

```
fun fib'(0,k) = k 1
  | fib'(1,k) = k 1
  | fib'(n,k) = fib'(n-1,
                  fn a => fib'(n-2,
                               fn b => k(a+b)))
```

or

`k o (fn b => b+a)`

or

`fn b => plus(a,b,k)`

Advantages of CPS Form

- Every function call is a tail call (a `goto`)
  - No stack required!
- Order of operations is explicit in the code
- Every intermediate quantity has a name

```
fun f(a, b, c, d, k) =
   times(a, b, fn x =>
   times(c, d, fn y =>
   plus (x, y, k)))
```

- Many compilers for functional programming languages automatically convert every program into CPS...including SML/NJ.

More Advantages of CPS

- Flexibility in handling control flow
  - Have the option of *not* invoking the continuation, but doing something else instead.

```
fun mult [] = 1
  | mult (0::_) = ...
  | mult (n::ns) = n * (mult ns)
```

```
fun mult'([],    k) = k 1
  | mult'(0::_, k) = k 0 (* or just 0 *)
  | mult'(n::ns,k) = mult'(ns, fn a => k(n*a))

fun mult(l) = mult'(l, fn a => a)
```

Digression: Exceptions

- Alternate approach for `mult`:

```
exception Zero

fun mult' []      = 1
  | mult' (0::_) = raise Zero
  | mult' (n::ns) = n * mult' ns

fun mult(l) = (mult' l)
                 handle Zero => 0
```

Observation on Exceptions

- Expressions being evaluated really must keep track of "two" continuations (possible futures)
  - What to do if the current expression returns a value
  - What to do if this expression raises an exception.
- Suppose we run the code

  ```
  print(Int.toString (mult [3,2,0,6])).
  ```

  What do the two continuations look like right before `raise Zero` is executed?

Success and Failure Continuations

- Exceptions can always be avoided (if desired) by making both continuations explicit.

```
fun mult'([],   ks,kf) = ks 1
  | mult'(0::_, ks,kf) = kf()
  | mult'(n::ns,ks,kf) =
       mult(ns, fn a => ks(n*a), kf)

fun mult(l) = mult'(l, fn a=>a, fn ()=>0)
```

```
fun mult(l,ks,kf) = mult'(l, ks, fn () => ks 0)
```