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 \(4\times5\) 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
  
  \( \text{fact}(4) + \text{fact}(2) \)?
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

\[
mult: \text{int list} \rightarrow \text{int}
\]

\[
mult: \text{int list} \times (\text{int} \rightarrow \text{ans}) \rightarrow \text{ans}
\]

(\(\text{ans}\) 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:

\[
\begin{align*}
\text{plus} & : \text{int} \times \text{int} \times (\text{int} \rightarrow \text{ans}) \rightarrow \text{ans} \\
\text{times} & : \text{int} \times \text{int} \times (\text{int} \rightarrow \text{ans}) \rightarrow \text{ans}
\end{align*}
\]

(* given a, b, c, and d, compute ab+cd *)

\[
\begin{align*}
\text{fun } f(a, b, c, d, k) & = \\
& \text{times}(a, b, \\
& \quad \quad \quad \quad \text{fn } x \Rightarrow \text{times}(c, d, \\
& \quad \quad \quad \quad \quad \quad \quad \text{fn } y \Rightarrow \text{plus}(x, y, k)))
\end{align*}
\]

- This is called continuation-passing style (CPS)

Another Example: \text{fact}

\[
\begin{align*}
\text{fun } \text{fact} \ 0 & = 1 \\
& \mid \text{fact } n = n \times \text{fact}(n-1)
\end{align*}
\]

\[
\begin{align*}
\text{fun } \text{fact'}(0, k) & = k \ 1 \\
& \mid \text{fact'}(n, k) = \text{fact'}(n-1, \text{fn } a \Rightarrow k(n*a))
\end{align*}
\]

or

\[
\text{fun } \text{fact'}(n, k) = \text{fact'}(n-1, \text{fn } a \Rightarrow k(n*a))
\]

or

\[
\text{fun } \text{fact'}(n, k) = \text{fact'}(n-1, \text{fn } a \Rightarrow \text{times}(n,a,k))
\]
Another Example: \( \text{fib} \)

\[
\begin{align*}
\text{fun} \ fib \ 0 &= 1 \\
& | \ fib \ 1 = 1 \\
& | \ fib \ n = \text{fib}(\text{n-1}) + \text{fib}(\text{n-2})
\end{align*}
\]

\[
\begin{align*}
\text{fun} \ fib'(0,k) &= k \ 1 \\
& | \ fib'(1,k) = k \ 1 \\
& | \ fib'(\text{n},k) = \text{fib}'(\text{n-1}, \\
& \quad \quad \quad \quad \text{fn} \ a => \text{fib}'(\text{n-2}, \\
& \quad \quad \quad \quad \quad \quad \text{fn} \ b => k(a+b))})
\end{align*}
\]

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

\[
\begin{align*}
\text{fun} \ f(a, b, c, d, k) &= \text{times}(a, b, \text{fn} \ x => \\
& \quad \quad \text{times}(c, d, \text{fn} \ y => \\
& \quad \quad \quad \text{plus}(x, y, k)))
\end{align*}
\]

More Advantages of CPS

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

\[
\begin{align*}
\text{fun} \ mult \ [] &= 1 \\
& | \ mult \ (0::_) = \ldots \\
& | \ mult \ (n::ns) = n * (\text{mult} \ ns)
\end{align*}
\]

\[
\begin{align*}
\text{fun} \ mult'(\_, \ k) &= k \ 1 \\
& | \ mult'(0::\_, \ k) = k \ 0 \ (* \ or \ \text{just} \ 0 \ *) \\
& | \ mult'(n::ns,k) = \text{mult}'(ns, \text{fn} \ a => k(a*n))
\end{align*}
\]

\[
\begin{align*}
\text{fun} \ mult(\_l) &= \text{mult}'(1, \text{fn} \ a => a)
\end{align*}
\]

Digression: Exceptions

- Alternate approach for \( \text{mult} \):

\[
\begin{align*}
\text{exception} \ Zero \\
\text{fun} \ mult' \ [] &= 1 \\
& | \ mult' \ (0::\_) = \text{raise} \ Zero \\
& | \ mult' \ (n::ns) = n * \text{mult}' \ ns
\end{align*}
\]

\[
\begin{align*}
\text{fun} \ mult(\_l) &= (\text{mult}' \ l) \\
& \quad \text{handle} \ Zero \Rightarrow 0
\end{align*}
\]
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
  
  \[
  \text{print(Int.toString (mult [3,2,0,6])).}
  \]

  What do the two continuations look like right before \texttt{raise Zero} is executed?

Success and Failure Continuations

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

\[
\begin{aligned}
\text{fun mult'}([], ks,kf) &= ks 1 \\
\text{mult'}(0::_, ks,kf) &= kf() \\
\text{mult'}(n::ns,ks,kf) &= \\
& \quad \text{mult}(ns, \text{fn a} \Rightarrow ks(n*a), kf) \\
\text{fun mult}(l) &= \text{mult'}(l, \text{fn a}=>a, \text{fn ()=>0)} \\
\text{fun mult}(l,ks,kf) &= \text{mult'}(l, ks, \text{fn ()=> ks 0)}
\end{aligned}
\]