15-150 Fall 2019
Continuations
Stephen Brookes

1 Topics

• Direct- and continuation-style versions of familiar functions.

• This material expands on some basic concepts that you should have been introduced to in lab.

• The lecture slides will cover different – and new – material about how to use continuations to implement backtracking algorithms. There will be a separate chapter of lecture notes to accompany that material.

• We showed in lecture last week how the use of “success” and “failure” continuations arose naturally when we worked to generalize a solution to a specific problem, aiming for greater generality.

• Continuation-style program design may seem harder to grasp at first sight than “direct” style, but it’s important to see and we hope you will come to appreciate the elegance with which we can solve problems in this style.
2 What is a continuation?

A function, used as a parameter by another function, and typically used to abstract away from “the rest of the computation”, or “what to do to finish a task”.

3 Direct- and continuation-style functions

A function of type \( \text{t1} \rightarrow \text{t2} \) expects to be applied to an argument value of type \( \text{t1} \), and returns a value of type \( \text{t2} \). We will refer to this as a direct-style function. Slogan: a direct-style function evaluates its argument and returns a value.

A continuation-passing version of such a function expects to be applied to a value of type \( \text{t1} \) and a continuation of type \( \text{t2} \rightarrow \text{ans} \), where \( \text{ans} \) is a type of “answers” or “final results”. Instead of “returning” a value of type \( \text{t2} \), the cps-function passes a value of type \( \text{t2} \) to the continuation, and thus produces an “answer”. Slogan: a continuation-style function evaluates its argument and calls its continuation with a value.

This idea is very general. Every “direct-style” function can be converted into a “continuation-style” version. Writing a function in continuation-style may help to emphasize the control flow, and indeed continuation-style can be a good way to implement a complex pattern of control flow.

To illustrate, let’s revisit the factorial function.

Factorial, revisited

Here is the direct-style factorial function \( \text{fact} \):

[Aside: we use an if-then-else expression rather than function clauses, in order to focus on and simplify the discussion of continuations.]

\[
(* \text{fact} : \text{int} \rightarrow \text{int} *)
\]

\[
\text{fun fact} \ n = \text{if} \ n=0 \ \text{then} \ 1 \ \text{else} \ n \ * \ \text{fact}(n-1)
\]

As before, we know that this function satisfies the specification given by:

For all \( n \geq 0 \), \( \text{fact}(n) \) returns the value of \( n! \).

Here is the cps-version of the factorial function, which we will call \( \text{FACT} \):

\[
(* \text{FACT} : \text{int} \rightarrow (\text{int} \rightarrow 'a) \rightarrow 'a *)
\]

\[
\text{fun FACT} \ n \ k = \text{if} \ n=0 \ \text{then} \ k(1) \ \text{else} \ \text{FACT}(n-1) \ (\text{fn} \ x => k(n \ * \ x))
\]
Observe that:

- \texttt{fact 0} returns 1
- \texttt{FACT 0 k} passes 1 to \( k \)
- For \( n \neq 0 \), \texttt{fact n} makes a recursive call to \texttt{fact(n-1)} and, if this returns a value \( v \), returns the product of \( n \) and \( v \)
- For \( n \neq 0 \), \texttt{FACT n k} calls \texttt{FACT (n-1)} with a continuation that, if passed a value \( v \), multiplies it by \( n \) and passes the product to \( k \).

The behavior of the cps-style function \texttt{FACT} is given by:

**Theorem**

For all \( n \geq 0 \), all types \( t \), and all continuations \( k:\textit{int} \to t \),
\[
 \texttt{FACT } n \ k = k(\textit{n}!).
\]

We can prove that this is true, by induction on the value of \( n \). Since this is the first time we've seen an example like this, here is the proof.

- **Base case**: For \( n = 0 \). We need to show that for all types \( t \) and all functions \( k:\textit{int} \to t \), \( \texttt{FACT 0 k} = k(0!) \). By definition of \texttt{FACT},

  \[
  \texttt{FACT 0 k} = \begin{cases} 
  k(1) & \text{if } 0=0 \\ 
  \texttt{FACT(0-1)(fn x => k(0*x))} & \text{else} 
  \end{cases}
  = k(1).
  \]

  Since \( 0!=1 \), we therefore have \( \texttt{FACT 0 k} = k(0!) \), as required.

- **Inductive step**: Let \( n>0 \), and suppose as the Induction Hypothesis that

  \[
  \text{(IH)}: \text{For all types } t \text{ and all } k':\textit{int} \to t, \\
  \texttt{FACT (n-1) k'} = k'((n-1)!).
  \]

  We must show that, for all types \( t \) and all functions \( k:\textit{int} \to t \), \( \texttt{FACT n k} = k(n!) \).
  Let \( t \) be a type and \( k:\textit{int} \to t \) be a function. Then, since \( n>0 \),

  \[
  \texttt{FACT n k} = \texttt{FACT (n-1) (fn x => k(n*x)) \ by def of FACT}
  \]

  So let \( k' \) be \( \texttt{(fn x => k(n*x))} \). Then we have
FACT n k
  = FACT (n-1) k' by def of FACT
  = k'((n-1)!) by IH
  = (fn x => k(n*x)) ((n-1)!) by def of k'
  = k(n*(n-1)!) by value-substitution
  = k(n!) by def of n!

- That completes the inductive proof.

Comments
(a) The type of FACT indicates that we can choose any type of answers that we like. For example, the SML function

Int.toString : int -> string

converts an integer value into a string. If we want to get a string as an answer, we can use this function as a continuation for FACT:

FACT 3 Int.toString = Int.toString (6) = "6"

(b) Since we know that the direct-style fact function satisfies its specification, we can conclude from the above proof for FACT that

For all n ≥ 0, and all types t, and all functions k:int -> t,
FACT n k = k(fact n).

Question
- What happens when n < 0? We don’t have a definition for n! when n is negative. Use stepping and evaluational reasoning, show that for all types t and all functions k:int -> t, FACT n k fails to terminate (has an infinite evaluation sequence). Remember that for negative values of n, fact n also fails to terminate. Is it true that for all integers n,

FACT n k = k (fact n)?
Continuation-style counting

Here is the direct-style function for adding the integers in a value of type \texttt{int tree}.

\begin{verbatim}
(* count : int tree -> int *)
fun count Empty = 0 
   | count (Node(left, x, right)) = 
       (count left) + x + (count right)
\end{verbatim}

Notice the control flow in the non-empty case: first make a recursive call on the left subtree, then make a recursive call on the right subtree, then do the arithmetic.

Here is the continuation-passing style version, in which this control flow is made explicit, and also the roles played by the results of these recursive calls is made explicit:

\begin{verbatim}
(* count' : int tree -> (int -> 'a) -> 'a *)
fun count' Empty k = k 0 
   | count' (Node(left, x, right)) = 
       count' left (fn m => count' right (fn n => k(m+x+n)))
\end{verbatim}

In the non-empty case the continuation on the right-hand side contains an embedded call to \texttt{count'} on the right subtree, with a continuation that does the arithmetic before passing the total to the original continuation.

**Exercise:**
Prove by induction on tree structure that, for all values \texttt{T:int tree}, all types \texttt{a} and all functions \texttt{k:int -> a},

\[
\texttt{count'} \ T \ k = k(\texttt{count } T).
\]

You can use without proof the fact that for all values \texttt{T:int tree}, \texttt{count } T returns a value.

Continuation-style, tail recursion, and efficiency

A recursive function definition is said to be \textit{tail recursive} if each recursive call made by the function is in “tail position”, which means that it is the last thing the function does before returning.

For example, the factorial function

\begin{verbatim}
(* fact : int -> int *)
fun fact n = if n=0 then 1 else n * fact(n-1)
\end{verbatim}
is not tail recursive, because the recursive call \( \text{fact}(n-1) \) is not in tail position: its result gets multiplied by the value of \( n \).

Similarly, the Fibonacci function

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

is not tail recursive, because the call to \( \text{fib}(n-2) \) is not in tail position.

On the other hand, the continuation-style functions

\[
\begin{align*}
(* \text{FACT} : \text{int} \rightarrow (\text{int} \rightarrow \text{a}) \rightarrow \text{a} *) \\
\text{fun} \quad \text{FACT} \ n \ k \ &= \ \text{if} \ n=0 \ \text{then} \ k(1) \ \text{else} \ \text{FACT}(n-1) \ (\text{fn} \ x \ => \ k(n * x))
\end{align*}
\]

\[
\begin{align*}
(* \text{FIB} : \text{int} \rightarrow (\text{int} \rightarrow \text{a}) \rightarrow \text{a} *) \\
\text{fun} \quad \text{FIB} \ 0 \ k \ &= \ k \ 1 \\
| \quad \text{FIB} \ 1 \ k \ &= \ k \ 1 \\
| \quad \text{FIB} \ n \ k \ &= \ \text{FIB} \ (n-1) \ (\text{fn} \ x \ => \ \text{FIB} \ (n-2) \ (\text{fn} \ y \ => \ k(x+y))))
\end{align*}
\]

are tail recursive.

Tail calls are significant because they can be implemented efficiently in a manner that typically saves space. Here is an abbreviated (and simplified) overview of the main ideas. When a function is called, the computer “pushes” the values of the function call’s arguments onto a stack, along with a “return address” that indicates the place it was called from, which is where the result of the call needs to be “returned” to. For tail calls, there is no need to remember the place we are calling from. Instead, one can perform tail call elimination by re-using the same stack frame for the new argument values but leaving the return address unchanged; the result of the tail call needs to be “returned” to the original caller.

Here is an illustration of these ideas, based on evaluating a call of the following (tail recursive) function:

\[
\begin{align*}
\text{fun} \quad \text{factacc} \ (n:\text{int}, \ a:\text{int}) : \text{int} = \\
\quad \text{if} \ n=0 \ \text{then} \ a \ \text{else} \ \text{factacc}(n-1, \ n*a)
\end{align*}
\]

Execution of the call \( \text{factacc}(3, 1) \) (without doing any tail call elimination) looks like:

\[
\begin{align*}
\text{call factacc} \ (3, 1) \\
\quad \text{call factacc} \ (2, 3)
\end{align*}
\]

\text{call factacc} \ (2, 3)
call factacc (1, 6)
call factacc (0, 6)
return 6
return 6
return 6
return 6

In the above display, indentation indicates the growth and shrinking of the stack: the call to factacc (3,1) involves pushing the argument pair (2,3) onto the stack along with a return address, and so on. There are several return addresses, one for each recursive call, but we end with a series of trivial returns that essentially just pass the value 6 along each time until we reach the return address for the original call. Executing factacc(n, a) for some n > 0 uses O(n) stack space.

If the implementation does tail call elimination, we would end up with an execution that looks like:

call factacc (3, 1)
  replace arguments with (2, 3) [same return address]
  replace arguments with (1, 6) [same return address]
  replace arguments with (0, 6) [same return address]
  return 6

This reorganization saves space because only the calling function’s address needs to be saved, and the stack frame for factacc is reused for the recursive calls. Doing things this way takes O(1) space.

In a language implementation that does tail call elimination automatically, the programmer need not worry about running out of stack space for extremely deep recursions. Also, the tail recursive variant of a function may be faster than a non-tail recursive variant, but typically only by a constant factor (which would imply the same O-class).

Some programmers working in functional languages will rewrite recursive code to be tail-recursive so they can take advantage of this feature. This often requires addition of an “accumulator” argument (a in factacc).