15–150: Principles of Functional Programming

Continuations

Michael Erdmann*

Spring 2020

1 Topics

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

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-Passing-Style Functions

A function of type `t1 -> t2` expects to be applied to an argument value of type `t1`, and returns a value of type `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-style* (CPS) version of such a function expects to be applied to a value of type `t1` and a continuation of type `t2 -> ans`, where `ans` is a type of "answers" or "final results". Instead of "returning" a value of type `t2`, the CPS function passes a value of type `t2` to the continuation, and thus produces an "answer". Slogan: a CPS function evaluates its argument and calls its continuation with a value.

*Adapted with small changes from a document by Stephen Brookes.

This idea is very general. Every "direct-style" function can be converted into a "continuation-passing-style" version. Writing a function in continuation-passing-style may help to emphasize the control flow, and indeed continuation-passing-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 `fact`:
[Aside: we use an `if-then-else` expression rather than function clauses, in order to focus on and simplify the discussion of continuations.]

```
(* fact : int -> int *)
fun fact n = if n=0 then 1 else n * fact(n-1)
```

As before, we know (i.e., can prove) that this function satisfies the specification given by:

> For all `n` ≥ 0, `fact(n)` returns the value `n`!.

Here is the CPS version of the factorial function, which we will call `FACT`:

```
(* FACT : int -> (int -> 'a) -> 'a *)
fun FACT n k = if n=0 then k(1) else FACT (n-1) (fn x => k(n * x))
```

Observe that:

- Base case:
  - `fact 0` returns 1.
  - `FACT 0 k` passes `1` to `k`.
- Recursive case:
  - For `n` $\neq 0$, `fact n` makes a recursive call to `fact (n-1)` and, if this returns a value `v`, returns the product of `n` and `v`.
  - For `n` $\neq 0$, `FACT n k` calls `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 function `FACT` is described formally as follows:

Theorem
For all `n` ≥ 0, all types `t`, and all continuations `k : int -> t`,
`FACT n k` $\cong$ `k(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:int -> t`, `FACT 0 k` $\cong$ `k(0!)`.

  By definition of `FACT`,

  ```
  FACT 0 k ≅ if 0=0 then k(1) else FACT(0-1)(fn x => k(0*x))
           ≅ k(1).
  ```

  Since `0!=1`, we therefore have `FACT 0 k` $\cong$ `k(0!)`, as required.

- Inductive step: Let `n>0`, and suppose as the Induction Hypothesis that

  ```
  (IH): For all types t and all k':int -> t,
        FACT (n-1) k' ≅ k'((n-1)!).
  ```

  We must show that, for all types `t` and all functions `k:int -> t`,
  `FACT n k` $\cong$ `k(n!)`.

  Let `t` be a type and `k:int -> t` be a function. Then, by the definition of `FACT`,

  ```
  FACT n k ==> if n=0 then k(1) else FACT(n-1)(fn x => k(n*x))
           ==> FACT (n-1) (fn x => k(n*x))      [since  n > 0]
  ```

  Thus `FACT n k` $\cong$ `FACT (n-1) (fn x => k(n*x))`.

  So let `k'` be `(fn x => k(n*x))`.
  Then we have

  ```
  FACT n k ≅ FACT (n-1) k'                [as we just saw]
           ≅ 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 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` $\cong$ `Int.toString (6)` $\cong$ `"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` $\cong$ `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` loops forever (i.e., has an infinite evaluation sequence). Remember that for negative values of `n`, `fact n` also loops forever.

  Is it true that for *all* integers `n`

  `FACT n k` $\cong$ `k (fact n)` ?

Continuation-Passing-Style Counting

Here is the direct-style function for adding the integers in a value of type `int tree`, using one of our standard tree datatypes:

```
(* sum : int tree -> int *)
fun sum Empty = 0
  | sum (Node(left, x, right)) = (sum left) + x + (sum right)
```

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:

```
(* sum' : int tree -> (int -> 'a) -> 'a *)
fun sum' Empty k = k 0
  | sum' (Node(left, x, right)) k =
      sum' left (fn m => sum' right (fn n => k(m+x+n)))
```

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

Observe the similarity in appearance to that of the function `flatten2` from Lecture 5.

Exercise: Prove by induction on tree structure that, for all values `T:int tree`, all types `s` and all functions `k:int -> s`,

`sum' T k` $\cong$ `k (sum T).`

(You may assume and use the fact that `sum` is total.)

Continuation-Passing-Style, Tail Recursion, and Efficiency

Recall our discussion of tail recursion from Lecture 4. We re-summarize in this section.

A recursive function definition is said to be *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.

Observe that the factorial function

```
(* fact : int -> int *)
fun fact n = if n=0 then 1 else n * fact(n-1)
```

is not tail recursive, because the recursive call `fact(n-1)` is not the last thing the function does before returning. Instead, the function waits for the result of the recursive call, then multiples that by the value of `n`.

Similarly, the Fibonacci function

```
(* fib : int -> int *)
fun fib 0 = 1
  | fib 1 = 1
  | fib n = fib(n-1) + fib(n-2)
```

is not tail recursive.

On the other hand, the continuation-passing-style functions

```
(* FACT : int -> (int -> 'a) -> 'a *)
fun FACT n k = if n=0 then k(1) else FACT (n-1) (fn x => k(n * x))

(* FIB : int -> (int -> 'a) -> 'a *)
fun FIB 0 k = k 1
  | FIB 1 k = k 1
  | FIB n k = FIB (n-1) (fn x => FIB (n-2) (fn y => k(x+y))
```

are tail recursive.

Exercise: Prove the following:

For all `n` ≥ 0, all types `t`, and all `k:int -> t`,

`FIB n k` $\cong$ `k(fib n)`.

Side comment: The definition of tail recursion gets a little tricky with functions like `flatten2` that make two recursive calls in the recursive case. Technically, such a function cannot be tail recursive, since one of the two calls cannot be the last thing the function does before returning. Instead, we spoke back in Lecture 5 of one of the recursive calls being in tail position and the other one not. Observe, however, that functions such as `sum'` and `FIB` (see previous two pages) really are tail recursive, since one of the two recursive calls is in tail position and the other recursive call is encapsulated within a continuation that is to be called later (and in this case the encapsulation further mimics tail position).

Stack Space

Tail calls are significant because they can be implemented efficiently in a manner that typically saves *space*, at least on the stack. 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:

```
fun factacc(n:int, a:int):int =
   if n=0 then a else factacc(n-1, n*a)
```

Execution of the call `factacc(3, 1)` (without doing any tail call elimination) looks like:

```
call factacc (3, 1)
  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 $\texttt{n} > 0$ uses $\texttt{O}(\texttt{n})$ 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 stack 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 $\texttt{O}(1)$ stack 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`).

Comment: Although one may save stack space by converting direct-style functions to tail recursive CPS functions, that may not save overall space, since the continuations themselves may grow in size.