15–212: Fundamental Structures of Computer Science II

Some Notes on Continuations

Frank Pfenning
(Edited by John Lafferty)

September 23, 1998

These notes provide a brief introduction to continuations as a programming technique. Continuations have a rich history and have made important contributions to the understanding of functional programming and compilation. One can distinguish *first-class continuations* (which are not available in Standard ML, although the SML of New Jersey implementation provides them) and *functions as continuations*. In these notes, we talk only about the latter; thus we do not refer to an extension of the language, but to a particular higher-order programming technique.

1 A Tail-Recursive Append Function

Recall the definition of function `@` that appends two lists

```
(* @ : 'a list * 'a list -> 'a list *)
fun @ (nil, k) = k
  | @ (x::l, k) = x :: @(l,k);
infixr @;
```

Note that this function is not tail-recursive, since it applies the list constructor `::` to the result of the recursive call. Nonetheless, this function is quite efficient and the definition above is fully satisfactory from a pragmatic point of view.

But one may still ask if there is any way to write an append function in tail-recursive form. The answer is "yes", although it will be less efficient than the direct version above. In fact, it is a deep property of ML that *every* function can be rewritten in tail-recursive form!

Suppose that we have a function f `:` t `->` s and would like to rewrite it as a function f' in tail-recursive form. The basic idea is to give f' an additional argument, called the *continuation*, which represents the computation that should be done on the result of f. In the base case, instead of returning a result, we call the continuation. This means that f' should have type

f' `:` t `-> (`s `-> 'b) -> 'b`.

In the recursive case we add whatever computation should be done on the result to the continuation. So a continuation is like a functional accumulator argument.

When we use this function to compute f we give it the *initial continuation* which is often the identity function, indicating no further computation is done on the result.

Applying this basic idea to append yields the following definition.

```
local
   (* @' : 'a list * 'a list * ('a list -> 'b) -> 'b *)
   fun @' (nil, k, cont) = cont k
     | @' (x::l, k, cont) = @' (l, k, fn r => cont (x::r))
in
   fun @ (l, k) = @' (l, k, fn r => r)
end
```

This first line of `@'` implements the idea that instead of returning the result `k`, we apply the continuation to `k`.

The second line of `@'` implements the idea that instead of constructing

```
x :: @ (l, k)
```

we call `@'` recursively on `l` and `k`, adding the task of prepending `x` to the argument `r` of the continuation `cont`.

Consider the following sample computation, where we have renamed some instances of the variables bound in the continuation to make them easier to read.

```
    @ ([1,2], [3,4])
==> @' ([1,2], [3,4], fn r => r)
==> @' ([2], [3,4], fn r1 => (fn r => r) (1::r1))
==> @' ([], [3,4], fn r2 => (fn r1 => (fn r => r) (1::r1)) (2::r2))
==> (fn r2 => (fn r1 => (fn r => r) (1::r1)) (2::r2)) [3,4]
==> (fn r1 => (fn r => r) (1::r1)) (2::[3,4])
==> (fn r1 => (fn r => r) (1::r1)) [2,3,4]
==> (fn r => r) (1::[2,3,4])
==> (fn r => r) [1,2,3,4]
==> [1,2,3,4]
```

It should be obvious that, even though the function is now tail-recursive, we haven't saved anything since we have to build up and then call the continuation. Compare the above with the direct computation:

```
    @ ([1,2], [3,4])
==> 1 :: @ ([2], [3,4])
==> 1 :: (2 :: @ ([], [3,4]))
==> 1 :: (2 :: [3,4])
==> 1 :: [2,3,4]
==> [1,2,3,4]
```

2 Proving Correctness

The correctness statement for the continuation-passing tail-recursive version of append expresses, formally, the informal explanation above.

Theorem 1 *For any lists l and k and continuations c of appropriate type we have*

`@' (l, k, c)` $\cong$ `c (@ (l, k))`

Proof: By structural induction on l.

Base: $l =$ `nil`. Then

$$\texttt{@' (nil, } k\texttt{, } c\texttt{)} \Longrightarrow c\ \texttt{(}k\texttt{)}$$

and

$$c\ \texttt{(@ (nil, } k\texttt{))} \Longrightarrow c\ \texttt{(}k\texttt{)},$$

so both sides reduce to the same expression.

Step: Assume

$$\texttt{@' (}l\texttt{, } k\texttt{, } c\texttt{)} \cong c\ \texttt{(@ (}l\texttt{, } k\texttt{))}$$

We have to show that

$$\texttt{@' (}x\texttt{::}l\texttt{, } k\texttt{, } c\texttt{)} \cong c\ \texttt{(@ (}x\texttt{::}l\texttt{, } k\texttt{))}$$

As usual we calculate both sides. First the left:

$$\begin{array}{ll} & \texttt{@' (}x\texttt{::}l\texttt{, } k\texttt{, } c\texttt{)} \\ \Longrightarrow & \texttt{@' (}l\texttt{, } k\texttt{, fn r => } c\ \texttt{(}x\texttt{::r))} \\ \cong & \texttt{(fn r => } c\ \texttt{(}x\texttt{::r)) (@ (}l\texttt{, } k\texttt{))} \quad \text{by ind. hyp.} \\ \Longrightarrow & \texttt{(fn r => } c\ \texttt{(}x\texttt{::r))}\ lk \\ \Longrightarrow & c\ \texttt{(}x\texttt{::}lk\texttt{)} \end{array}$$

where $\texttt{@ (}l\texttt{, } k\texttt{)} \Longrightarrow lk$ always exists. For the right-hand side we obtain:

$$\begin{array}{ll} & c\ \texttt{(@ (}x\texttt{::}l\texttt{, } k\texttt{))} \\ \Longrightarrow & c\ \texttt{(}x\texttt{::@(}l\texttt{, } k\texttt{))} \\ \Longrightarrow & c\ \texttt{(}x\texttt{::}lk\texttt{)} \end{array}$$

So the left and right hand side are equivalent; that is, they reduce to the same expression (and from there to the same value, if the expression has a value, or both sides have no value).

□

3 Control with Continuations

In the function that appends two lists, continuations can be applied, but they do not help in writing simpler or more efficient code. But in functions with more complex control flow, they can often be used to advantage. In a future lecture, we will see a major application, namely the implementation of backtracking in an acceptor for regular expressions.

Here, we consider a simpler example, namely a function that traverses a list and returns the shortest prefix of the list such that a predicate p is false on the next element. If p holds on all elements, we return `NONE` to indicate no such prefix exists. The implementation takes advantage of the type `'a option`, which is pre-declared in ML with

```
datatype 'a option = NONE | SOME of 'a;
```

That is, the values of type τ `option` are either `NONE` or `SOME` v for any value v of type τ.

First, a direct implementation, using a helper function that conses an element onto an optional list.

```
(* consOpt : 'a * ('a list) option -> ('a list) option *)
fun consOpt (y, NONE) = NONE
  | consOpt (y, SOME (k)) = SOME (y::k)

(* prefix : ('a -> bool) -> 'a list -> ('a list) option *)
fun prefix p nil = NONE
  | prefix p (y::l) =
    if p(y) then consOpt (y, prefix p l)
    else SOME (nil);
```

Note an inefficiency in this function: if `p` is is true for every element in the list, `consOpt` is called on every element after the end of the list has been reached, passing along `NONE` all the way up to be returned as the overall result.

Instead, we would like to return `NONE` immediately once we have reached the end of the list and found no element on which `p` is false. Using continuations we can achieve this easily, simply by ignoring the continuation in this case.

```
local
  (* prefix' : ('a -> bool)
               -> 'a list * ('a list -> 'b option)
               -> 'b option *)
  (* Used as : ('a -> bool)
               -> ('a list * ('a list -> ('a list) option))
               -> ('a list) option *)
  fun prefix' p (nil, cont) = NONE
    | prefix' p (y::l, cont) =
      if p(y) then prefix' p (l, fn r => cont (y::r))
      else cont (nil)
in
  (* prefix2 : ('a -> bool) -> 'a list -> ('a list) option *)
  fun prefix2 p l = prefix' p (l, fn r => SOME r)
end;
```

Note that in the "success" case (we find an element where p is false) we call the continuation `cont` on the expected answer (`nil` in this program). In the case of a "failure" (we do not find a place where p is false) we discard the continuation and return directly (with `NONE` in this program). In this pattern of use we refer to the continuation `cont` as a *success continuation*.

It is instructive to construct a correctness proof for `prefix'` in a manner analogous to the tail-recursive append. We leave this an an exercise to the reader.

4 Further Examples

The pattern of control required for the `prefix` function can also be achieved by using *exceptions*, which are effects and therefore outside the pure fragment of ML. However, for more complicated control patterns, continuations are often the most natural and simplest way to program a solution. Many backtracking algorithms (such as the regular expression matcher in an upcoming lecture) are of this nature.

Another class of examples includes *co-routines*, which are functions that call each other in an egalitarian way (as distinct from *subroutines*, where one is subordinate to the other). Applications of continuations also arise when the call pattern is at cross-purposes with the natural modularization of the code. In this case we pass a continuation to the function in another module that can then "return" control by invoking the continuation. In the language of operating systems this is often referred to as an *up-call*. We will see an example of this in the guest lecture on the Foxnet project, which built the networking layer for an operating system entirely in ML.

To summarize the important principles that guide the implementation of a function passing a continuation:

- In base cases, we apply the continuation instead of directly returning a value.
- In recursive cases, the continuation acts as a functional accumulator.
- In exceptional cases, we may discard (or duplicate) the continuation to circumvent the normal control flow.