15-312 Foundations of Programming Languages
Recitation 6: Continuations

Daniel Spoonhower
spoons+@cs

October 1, 2003

1 Homework Solution

Review the solution to Assignment 3. Be sure you understand (in general) what sorts of programs constitute counterexamples to progress and preservation. Furthermore, be sure you understand the counterexample in the case of lazy MinML.

(In recitation, we will also touch (ever so briefly) on another issue raised by the alternative *CaseTyp'* rule, the decidability of typechecking. Finally, we will discuss the differences between call-by-value and call-by-name semantics for our pure functional language.)

2 Continuations

We began discussion of continuations last week in lecture; we will continue today with a pair of more detailed examples, both borrowed from Harper's notes.

2.1 Review

Recall the static semantics of our constructs for manipulating continuations.

$$\frac{\Gamma, x{:}\tau \vdash e : \tau}{\Gamma \vdash \mathtt{letcc}\, x\, \mathtt{in}\, e\, \mathtt{end} : \tau} \qquad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_1\ \mathsf{cont}}{\Gamma \vdash \mathtt{throw}[\tau]\, e_1\, \mathtt{to}\, e_2 : \tau}$$

Remember that `letcc` binds the current continuation to a variable and then continues evaluation, while `throw` evaluations an expression and then continues at the point of evaluation marked by the second expression.

2.2 Short Circuiting Evaluation

Consider the following function that computes the product of the elements in an integer list.

```
fun mul(l : int list) : int =>
  case l
    of nil => 1
     | x::l => x * mul l
```

Note that if the list contains the element 0, this function might perform a significant amount of extra work, particularly if the 0 appears near the beginning of the list. We might consider a *short-circuited* version of multiplication, one where we stop inspecting the remainder of the list once we encounter a 0.

```
fun mul(l : int list) : int =>
  case l
    of nil => 1
     | x::l => if x = 0 then 0
               else x * mul l fi
```

If we expect a list of length n to have exactly one 0 (distributed uniformly), we've reduced the (expected) number of recursive calls by $n/2$. We are still, however, performing $n/2$ multiplications (again, expected case), each of whose result will be 0. We'd like to jump out the entire sequence of recursive calls, not just the current one.

For reasons that will become clear in a moment, we first transform the function by η-expansion:

```
fn l : int list =>
  let mul = fun mul(l : int list) : int =>
    case l
      of nil => 1
       | x::l => if x = 0 then 0
                 else x * mul l fi
  in
    mul l
  end
```

Now, using the `letcc` and `throw` constructs from above, we can write

```
fn l : int list =>
  letcc ret in
    let mul = fun mul(l : int list) : int =>
      case l
        of nil => 1
         | x::l => if x = 0 then throw 0 to ret
                   else x * mul l fi
    in
      mul l
    end
  end
```

In this example, we are throwing a value *backward* to a previous point in evaluation, and moreover, we don't really use `ret` for anything particularly interesting. We could have easily written a similar short-circuiting function using exceptions. Next, we'll see an example where that is definitely *not* the case.

2.3 Composition

Remember that continuations are *values*: even though we can't write a value of type τ `cont` in the concrete syntax, they may be manipulated just like any other value.

We'd like to write a function `compose` that combines a function with a continuation, resulting in a new continuation. Specifically, a function with the following type. (Why does this make sense?)

compose : (τ' -> τ) -> τ cont -> τ' cont

We begin as follows:

fn f : τ' -> τ => fn k : τ cont =>

Now what do we do? Let's inspect the types and see what we *can* do.

`throw[?]` (something of type τ) `to k`
`f` (applied to something of type τ')

Finally, we know we want to return a value of type τ' `cont`, and there is only one way to create such a value:

`letcc k' in` (something of type τ') `end`

Let's start from the end and work backwards. The `k'` above holds the value we'd like to return, but we can't simply write

```
letcc k' in k' end
```

(Remember from lecture that such an expression is not well-typed.)

So how else can we save the continuation (and return it later)? Well, the only other thing we can *do* with a continuation is to `throw` it!

```
letcc k' in throw[τ'] k' to ? end
```

(Why do we give the `throw` expression type τ'?) Of course, we need somewhere to throw this continuation, so let's use another `letcc`.

```
letcc ret in ... letcc k' in throw[τ'] k' to ret end end
```

Now that we have captured the continuation we want, let's go back and consider what we'd do if someone actually threw to it. First we'd apply `f`:

```
letcc ret in ... f (letcc k' in throw[τ'] k' to ret end) end
```

What remains? We have only to `throw` some value of type τ to `k`. The result of the application of `f` is just such value.

```
fn f : τ' -> τ => fn k : τ cont =>
  letcc ret in
    throw[τ' cont] f (letcc k' in throw[τ'] k' to ret end) to k
  end
```

(Convince yourself that this function typechecks. What's the type of `ret`? Finally, does `compose` ever return? Did we ever expect it to?)

Clearly, we could not accomplish a feat such as `compose` with exceptions!