15-312 Foundations of Programming Languages
Recitation 5: Closures

Daniel Spoonhower
spoons+@cs

September 24, 2003

1 Closures in Action

To see why closures are necessary, consider evaluation of the following expression in detail (using the original set of evaluation rules given in lecture).

$$
\begin{array}{rl}
 & \cdot \mid \cdot > \texttt{apply}(\texttt{apply}(\texttt{fn}(\texttt{int}, x.\texttt{fn}(\texttt{int}, y.x)), 7), 3) \\
\mapsto_{\mathsf{e}} & \texttt{apply}(\square, 3) \mid \cdot > \texttt{apply}(\texttt{fn}(\texttt{int}, x.\texttt{fn}(\texttt{int}, y.x)), 7) \\
\mapsto_{\mathsf{e}} & \texttt{apply}(\square, 3) \triangleright \texttt{apply}(\square, 7) \mid \cdot > \texttt{fn}(\texttt{int}, x.\texttt{fn}(\texttt{int}, y.x)) \\
\mapsto_{\mathsf{e}} & \texttt{apply}(\square, 3) \triangleright \texttt{apply}(\square, 7) \mid \cdot > \texttt{fn}(\texttt{int}, x.\texttt{fn}(\texttt{int}, y.x)) \\
\mapsto_{\mathsf{e}} & \texttt{apply}(\square, 3) \triangleright \texttt{apply}(\square, 7) \mid \cdot < \texttt{fn}(\texttt{int}, x.\texttt{fn}(\texttt{int}, y.x)) \\
\mapsto_{\mathsf{e}} & \texttt{apply}(\square, 3) \triangleright \texttt{apply}(\texttt{fn}(\texttt{int}, x.\texttt{fn}(\texttt{int}, y.x)), \square) \mid \cdot > 7 \\
\mapsto_{\mathsf{e}} & \texttt{apply}(\square, 3) \triangleright \texttt{apply}(\texttt{fn}(\texttt{int}, x.\texttt{fn}(\texttt{int}, y.x)), \square) \mid \cdot < 7 \\
\mapsto_{\mathsf{e}} & \texttt{apply}(\square, 3) \triangleright \cdot \mid \cdot, x = 7 > \texttt{fn}(\texttt{int}, y.x) \\
\mapsto_{\mathsf{e}} & \texttt{apply}(\square, 3) \triangleright \cdot \mid \cdot, x = 7 < \texttt{fn}(\texttt{int}, y.x) \\
\mapsto_{\mathsf{e}} & \texttt{apply}(\square, 3) \mid \cdot < \texttt{fn}(\texttt{int}, y.x) \\
 & \text{(You should be quite skeptical at this point!)} \\
\mapsto_{\mathsf{e}} & \texttt{apply}(\texttt{fn}(\texttt{int}, y.x), \square) \mid \cdot > 3 \\
\mapsto_{\mathsf{e}} & \texttt{apply}(\texttt{fn}(\texttt{int}, y.x), \square) \mid \cdot < 3 \\
\mapsto_{\mathsf{e}} & \cdot \mid \cdot, y = 3 > x
\end{array}
$$

With even a very simple example, we are able to show how things go wrong.

(Do you see how to modify the above evaluation sequence to include closures for the functions?)

2 Suspensions

The look-up rule for ordinary variables was

$$k \mid \eta > x \mapsto_{\mathsf{e}} k \mid \eta < v$$

when $(x = v) \in \eta$.

Think about how this rule would be applied in expression such as

$$\texttt{rec}(\texttt{int} \to \texttt{int}, f.\texttt{fn}(\texttt{int}, x.\texttt{apply}(f, x)))$$

Using an ordinary binding and the above look-up rule, we would arrive a machine state such as

$$k \mid \eta < \texttt{rec}(\texttt{int} \to \texttt{int}, f.\texttt{fn}(\texttt{int}, x.\texttt{apply}(f, x)))$$

However, `rec` is *not* a value! Pushing this expression back into an evaluation frame will not maintain the semantics we have previously presented. How do we go about fixing our machine semantics to preserve the meaning of our program?

(An alternative semantics of `rec` in a call-by-value language is to bind the recursive variable to a *thunk*. That is

$$k \mid \eta > \texttt{rec}(\tau, x.e) \mapsto k \triangleright \eta \mid \eta, x = \langle\!\langle \eta; \texttt{fn}(\texttt{unit}, \_.e) \rangle\!\rangle > e$$

with all occurrences of x in e rewritten as $\texttt{apply}(x, \texttt{unitel})$. How does this compare to our semantics?)

3 Closure Conversion

Rather than build closures as part of the dynamic semantics, we can ensure that all values are closed statically, using a process known as *closure conversion*. The general idea is to determine the free variables of an expression and then build a record mapping each of the variables to a value. Then we wrap the expression in a new function abstraction (a function whose parameter is this record) and replace uses of free variables with projections of the record.[1] Let's consider this in a bit more detail.

Take our example from above:

$$\texttt{fn}(\texttt{int}, x.\texttt{fn}(\texttt{int}, y.x))$$

Call this expression `f`. Note that `g` itself is closed, though it contains a subexpression with a free variable. We will take that subexpression and add an explicit environment parameter.

$$\texttt{fn}(?, env.\texttt{fn}(\texttt{int}, y.x))$$

Then we replace occurrences of the free variable with projections from env

$$\texttt{fn}(?, env.\texttt{fn}(\texttt{int}, y.\texttt{proj}(x, env)))$$

and pair this new (closed) function with a record that binds exactly those records that are necessary.

$$\texttt{pair}(\texttt{fn}(?, env.\texttt{fn}(\texttt{int}, y.\texttt{proj}(x, env))), \{x = x\})$$

We then could do the same for g itself, though in this case the environment record would be empty. Applications $g4$ would be written as follows.

$$\texttt{apply}(\texttt{apply}(\texttt{fst}(g), \texttt{snd}(g)), 4)$$

[1] That said, we'll discuss closure conversion in a language with pairs and records.

3.1 Typing

Consider the following example before

```
let w = 5 in
  let x = true in
    if true then
      fn y : int => w
    else
      fn z : int => if x then z else 7 fi
    fi
  end
end : int -> int
```

and after closure conversion:

```
let w = 5 in
  let x = true in
    if true then
      (fn env : ? => fn y : int => #w env, {w = w})
    else
     (fn env : ? => fn z : int => if #x env then z else 7 fi, {x = x})
    fi
  end
end : ?
```

What type can we assign to the `if` statement – perhaps something such as $? \rightarrow \texttt{int} \rightarrow \texttt{int}$? Think about how we would use the result of this expression. Call the entire (closure converted) expression above `g`. `g` has a product type $(? \rightarrow \texttt{int} \rightarrow \texttt{int}) * ?$ where the two "?" are the same. How can we enforce this constraint? As we will see in an upcoming lecture, *existential types* are exactly what we need.

3.2 A Look Ahead

Another way to describe what we've done here would be to say that we've bound up the code of our program together with the data relevant to that code. In our discussion above, we only considered pairing a single function with its closure, but there's no reason not to consider functions that *share* data. In fact, we will do exactly that when we add *objects* to MinML a few weeks from now.