Working with Procedures

CPSC 509: Programming Language Principles

Ronald Garcia*

27 February 2013

In our last class, we introduced the idea of procedural abstraction using the $\lambda(x, t)$ expression and procedure applications $\mathsf{apply}(t, t)$. This allowed us to capture common program patterns, introduce *parameters* to represent the uncommon part, and then produce particular instances by applying the procedure to a particular *argument*. We motivated these features with some example programs that we'd like to write, but we haven't quite written those programs. In fact, it seems that we need more features to be able to write them. In this lecture, we introduce some new features that correspond to what it looks like we're missing, and then we show how technically we don't need them: procedures suffice.

1 Variable Binding

In our earliest example last class, we assumed that we could define a function with a name, and then call it, as in the following repeated example:

```
f(x) = if zero?(pred(x))
       then x
       else succ(x)

f(succ(succ(z)))
```

Right now we have no way to give a function a name and use it later, we can just create lambda abstractions. To do this, we'll steal a feature from the Racket language: *let binding*. The syntax for let binding follows:

$$t := \ldots \mid \mathsf{let}\ x = t\ \mathsf{in}\ t$$

(talk about free variables)

$$FV(\mathsf{let}\ x = t_1\ \mathsf{in}\ t_2) = FV(t_1) \cup (FV(t_2) / \{ x \})$$

(talk about substitution)

$$[v/x](\mathsf{let}\ x = t_1\ \mathsf{in}\ t_2) = \mathsf{let}\ x = [v/x]t_1\ \mathsf{in}\ t_2$$
$$[v/x_1](\mathsf{let}\ x_2 = t_1\ \mathsf{in}\ t_2) = \mathsf{let}\ x_2 = [v/x_1]t_1\ \mathsf{in}\ [v/x_1]t_2 \text{ if } x_1 \neq x_2$$

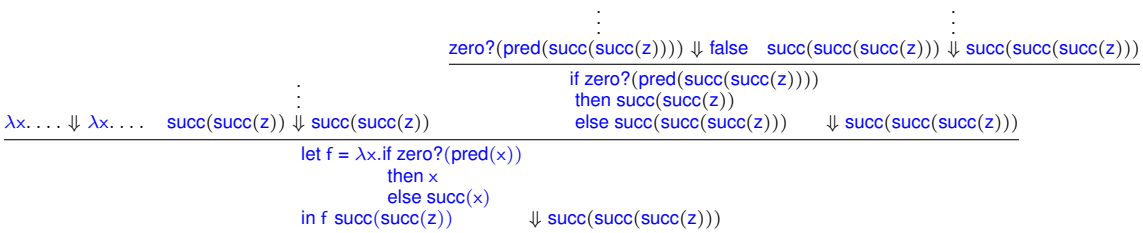
$$\text{elet}\frac{t_1 \Downarrow v_1 \qquad [v_1/x]t_2 \Downarrow v}{\mathsf{let}\ x = t_1\ \mathsf{in}\ t_2 \Downarrow v}$$

Now with let, we can write the example program from class:

```
let f = λx.if zero?(pred(x))
          then x
          else succ(x)
in f succ(succ(z))
```

*© Ronald Garcia.

Its derivation follows:



1.1 Technically, you don't need let

(talk about the translation)

(show the proof tree for left-left-lambda and compare)

$$\frac{\overline{(\lambda x.t_2) \Downarrow (\lambda x.t_2)} \quad t_1 \Downarrow v_1 \quad [v_1/x]t_2 \Downarrow v}{(\lambda x.t_2)\ t_1 \Downarrow v}$$

If we ignore the first premise, which is a closed derivation of $\overline{(\lambda x.t_2) \Downarrow (\lambda x.t_2)}$, then we end up with *exactly* the same premises as for let. That means that the resulting v is *exactly* the same value result.

What this tells us is that lambda abstraction can express the same computation as the let can. Mind you, we may still want let in our language, because it more clearly expresses what you are intending to do, but we now know that we can replace let with lambda and any immediate lambda application with a let[1]

2 Multi-parameter functions

Now that we can name functions, it would also be great if we could define functions that take more than one parameter, like

```
app(f,x) = f(x)
app(times2,6)
```

In this contrived example, the `app` function takes two arguments, a function and an argument to that function, and applies the function to the argument.

Our language only defines functions that have one parameter, so you might think that we need to somehow extend our language with support for multi-parameter functions so we can write:

$$\begin{array}{l}\text{let app} = \lambda(\text{f},\text{x}).\text{f x}\\ \text{in app}(\text{times2}, 6)\end{array}$$

It turns out that we don't need to.[2]

We can already write such a program as follows:

$$\begin{array}{l}\text{let app} = \lambda\text{f}.\lambda\text{x}.\text{f x}\\ \text{in app times2 } 6\end{array}$$

What have we done? We've replaced a function of two arguments, with a function of one argument *that returns another function of one argument*, and we can give it two arguments by applying them to the function separately. This technique for factoring function arguments is called *currying*.[3]

[1] In fact, the Chez Scheme optimizing compiler uses this correspondence to simplify compilation and generalize optimization.

[2] However, we might *want* to, and have good reasons for it!

[3] Currying is named after the American logician Haskell Curry; his first and last name has been used as the name for two separate programming languages with lambdas like we've studied here. Incidentally, currying was invented not by Curry, but by the Russian logician Moses Schönfinkel.

Here is a succinct example of currying in action:

$$\dfrac{\dfrac{\dfrac{}{(\lambda x.\ldots)\Downarrow(\lambda x.\ldots)}\quad\dfrac{}{z\Downarrow z}\quad\dfrac{}{\lambda y.\text{if true then z else y}\Downarrow\lambda y.\text{if true then z else y}}}{(\lambda x.\lambda y.\text{if true then x else y})\ z\Downarrow\lambda y.\text{if true then z else y}}\quad\dfrac{}{\text{false}\Downarrow\text{false}}\quad\dfrac{\dfrac{}{\text{true}\Downarrow\text{true}}}{\text{if true then z else false}\Downarrow z}}{(\lambda x.\lambda y.\text{if true then x else y})\ z\ \text{false}\Downarrow z}$$

3 Recursion

When we implemented our interpreters for IMP and the Boolean Language, we had to write some recursive procedures, which called themselves. Rather than consider a whole interpreter, let's consider a simpler program, though it may not do anything interesting:

$$\begin{array}{l}\text{let f} = \lambda x.\lambda y.\text{f x y}\\ \text{in f z z}\end{array}$$

Based on our earlier understanding of let, we can translate this to the following program.

$$(\lambda \text{f.f z z})\ (\lambda x.\lambda y.\text{f x y})$$

Now we should pretty easily see a problem with this program. The f in the first abstraction is bound, but the *second* f is free! It doesn't refer to any binding of f. We might have hoped that this f would refer to the function that we were defining, but that's not the case. We'll have to find another way to define a recursive function.

To do this, we introduce a more general but independent language feature that allows us to define *any* kind of object that refers to itself, not just recursive functions. The feature is the rec construct, and its syntax is as follows:

$$t := \ldots \mid \text{rec}\ x.t$$

The variable x is bound by the rec form, so we extend the free-variable function as follows:

$$FV(\text{rec}\ x.t) = FV(t)/\{x\}$$

Furthermore, we extend substitution so that it doesn't interfere with the variable bound by rec:

$$\begin{array}{rl}[v/x]\text{rec}\ x.t &= \text{rec}\ x.t\\ [v/x_1]\text{rec}\ x_2.t &= \text{rec}\ x_2.[v/x_1]t \text{ if } x_1 \neq x_2\end{array}$$

The idea behind rec is that any reference to the bound variable within its body *really* refers to the entire expression, including the rec . This behavior is captured by the following big-step rule.

$$\text{(erec)}\ \dfrac{t[\text{rec}\ x.t/x]\Downarrow v}{\text{rec}\ x.t\Downarrow v}$$

The rec form is evaluated by *substituting itself into itself!*. If you stop and think about it, this can easily lead to never-ending self-substitution. For example, the term rec x.x cannot build a derivation because it keeps repeating since x[(rec x.x)/x] = rec x.x:

$$\dfrac{\dfrac{\dfrac{\vdots}{\text{rec x.x}\Downarrow}}{\text{rec x.x}\Downarrow}}{\text{rec x.x}\Downarrow}$$

On the other hand, if we never refer to the bound variable in the body, nothing interesting happens:

$$\dfrac{\dfrac{}{z\Downarrow z}}{\text{rec x.z}\Downarrow z}$$

Things actually get interesting when we use the bound variable, but still produce something that computes. Let's look back at our earlier example with its free variable. If we bind it using rec , then we'll ultimately get a lambda abstraction that refers to itself, and that term itself evaluates to a value.

$$\frac{\lambda x.\lambda y.(\text{rec f}.\lambda x.\lambda y.\text{f x y})\ \text{x y} \Downarrow \lambda x.\lambda y.(\text{rec f}.\lambda x.\lambda y.\text{f x y})\ \text{x y}}{\text{rec f}.\lambda x.\lambda y.\text{f x y} \Downarrow \lambda x.\lambda y.(\text{rec f}.\lambda x.\lambda y.\text{f x y})\ \text{x y}}$$

Notice how the rec self-substitutes *only once*, but the body of the lambda prevents further substitution until the function is called. Typically it takes something like a lambda abstraction to suspend evaluation and keep this from running forever.

Now with this in hand, we can finally render our addition program.

```
let add = rec me.λx.λy.
            if zero?(x)
             then y
             else succ(me (pred(x)) y)
 in add succ(z) z
```

We needed to use rec to make the function self-referential (which is why we give the recursive variable binding the tongue-in-cheek name me. We could have given that variable the name add as well, if we wished.

3.1 You don't need rec either!

Now let's consider our silly recursive program again:

$$\text{rec f}.\lambda x.\lambda y.\text{f x y}$$

Well, wouldn't you know it: we can do this without rec too!

Here's the trick: We want f to somehow refer to itself, but how do we do that? Step one is to make a subtle change: make it so that instead of actually *being* itself, f is instead a function that takes itself and then acts like the body of the function itself.[4] We can write this as follows:

$$\lambda \text{f}.\lambda x.\lambda y.(\text{f f})\ \text{x y}$$

The only textual changes here are that the rec is replaced with a λ, and the reference to f is replaced with (f f) which feeds f to itself. Now if we assume that f represents a function that takes itself and then acts like the body of the function, then (f f) should act like the body of the function. The challenge now then is to find something to pass in for f. The answer is f itself!

$$\text{let fn} = \lambda \text{f}.\lambda x.\lambda y.(\text{f f})\ \text{x y in fn fn}$$

or more explicitly:

$$(\lambda \text{f}.\lambda x.\lambda y.(\text{f f})\ \text{x y})\ \lambda \text{f}.\lambda x.\lambda y.(\text{f f})\ \text{x y}$$

The result of evaluating this expression (its derivation is an exercise for the reader) is:

$$\lambda x.\lambda y.((\lambda \text{f}.\lambda x.\lambda y.(\text{f f})\ \text{x y})\ \lambda \text{f}.\lambda x.\lambda y.(\text{f f})\ \text{x y})\ \text{x y}$$

or more succinctly:

$$\lambda x.\lambda y.(\text{FN FN})\ \text{x y}$$

where FN stands for the *entire* original function.

[4] yeah, this giving me a headache too.

3.2 All you *really* need is procedures!

In this class, we've shown that a few features like multi-parameter functions, name binding, and recursion, would be useful but technically aren't needed to write programs that we would like to. Of course, for reasons of clarity, we want those features in the language we program in, but we can understand them in simpler terms.

It turns out, that we can take these simplifications much further, to an extreme! In fact, using lambda, you can encode boolean expressions like true, false, and if, and you can encode numbers and arithmetic. The language of $\lambda x.t$, x, and $t\ t$ is all you need to express *any computation whatsoever*. This is at the core of the *Turing-Church thesis*, which asserts that all of computation can be expressed using either Turing machines or *lambda calculus*, a variant of the little language of procedures that we are using. But few people have much interest in programming with Turing machines all day, and the same goes with the lambda calculus.