Lecture Notes on Primitive Recursion

15-814: Types and Programming Languages
Frank Pfenning

Lecture 2
Thursday, September 2, 2021

1 Introduction

In this lecture we continue our exploration of the λ-calculus and the representation of data and functions on them. We give schematic forms to define functions on natural numbers and give uniform ways to represent them in the λ-calculus. We begin with the *schema of iteration* and then proceed the more complex *schema of primitive recursion*. In the next lecture we will arrive at the fully general scheme of *recursion*.

2 Function Composition Revisited

The unit of composition should the identity function, as defined by $I = \lambda x.\, x$. Composing any other function f with I should just yield f. In other words, we expect

$$B\ f\ I \overset{?}{=} f \overset{?}{=} B\, I\, f$$

Let's calculate:

$$\begin{array}{rcl} B\ f\ I & = & (\lambda f.\, \lambda g.\, \lambda x.\, f\,(g\,x))\ f\ I \\ & \longrightarrow_\beta & (\lambda g.\, \lambda x.\, f\,(g\,x))\ I \\ & \longrightarrow_\beta & \lambda x.\, f\,(I\,x) \\ & \longrightarrow_\beta & \lambda x.\, f\,x \\ & \overset{?}{=} & f \end{array}$$

We see the result is not exactly f as we expected, but $\lambda x.\, f\,x$. However, these two expressions always behave the same when applied to an arbitrary argument so they are *extensionally equal*. To capture this we add one more rule to the λ-calculus:

$$\eta\text{-conversion} \quad \lambda x.\, e\, x \quad =_\eta \quad e \quad \text{provided } x \notin \mathsf{FV}(e)$$

The proviso that x not be among the *free variables* of e is needed, because $\lambda x.\, x\, x \neq \lambda x.\, y\, x$. The first applies the argument to itself, the second applies y to the given argument.

It is possible to orient this equation and investigate the notion of $\beta\eta$-reduction. However, it turns out this is somewhat artificial because extensionality is a reasoning principle for equality and not a priori a computational principle. Interestingly, in the setting of typed λ-calculi it makes more sense to use the equation from right to left, called η-expansion, but some discipline has to be imposed or expansion does not terminate.

We should remember that this form of extensionality does not extend to functions defined over specific representations. For example, we saw there are (at least) two formulations of negation on Booleans which are not equal, even if we throw in the rule of η-conversion.

3 Representing Natural Numbers

Finite types such as Booleans are not particularly interesting. When we think about the computational power of a calculus we generally consider the *natural numbers* $0, 1, 2, \ldots$. We would like a representation $\overline{n}$ such that they are all distinct. We obtain this by thinking of the natural numbers as generated from zero by repeated application of the successor function. Since we want our representations to be closed we start with two abstractions: one (z) that stands for zero, and one (s) that stands for the successor function.

$$\begin{array}{lcl}
\overline{0} & = & \lambda s.\, \lambda z.\, z \\
\overline{1} & = & \lambda s.\, \lambda z.\, s\, z \\
\overline{2} & = & \lambda s.\, \lambda z.\, s\, (s\, z) \\
\overline{3} & = & \lambda s.\, \lambda z.\, s\, (s\, (s\, z)) \\
\ldots & & \\
\overline{n} & = & \lambda s.\, \lambda z.\, \underbrace{s\, (\ldots (s}_{n \text{ times}}\; z))
\end{array}$$

In other words, the representation $\overline{n}$ iterates its first argument n times over its second argument

$$\overline{n}\, f\, x = f^n(x)$$

where $f^n(x) = \underbrace{f(\ldots(f}_{n \text{ times}}(x)))$

The first order of business now is to define a successor function that satisfies *succ* $\overline{n} = \overline{n+1}$. As usual, there is more than one way to define it, here is one (throwing in the definition of *zero* for uniformity):

$$\begin{array}{lclcl} \mathit{zero} & = & \overline{0} & = & \lambda s.\, \lambda z.\, z \\ \mathit{succ} & = & \lambda n.\, \overline{n+1} & = & \lambda n.\, \lambda s.\, \lambda z.\, s\,(n\, s\, z) \end{array}$$

We cannot carry out the correctness proof in closed form as we did for the Booleans since there would be infinitely many cases to consider. Instead we calculate generically (using mathmetical notation and properties)

$$\begin{array}{ll} & \mathit{succ}\ \overline{n} \\ = & \lambda s.\, \lambda z.\, s\,(\overline{n}\, z\, s) \\ = & \lambda s.\, \lambda z.\, s\,(s^n(z)) \\ = & \lambda s.\, \lambda z.\, s^{n+1}(z) \\ = & \overline{n+1} \end{array}$$

A more formal argument might use mathematical induction over n.

Using the iteration property we can now define other mathematical functions over the natural numbers. For example, addition of n and k iterates the successor function n times on k.

$$\mathit{plus} = \lambda n.\, \lambda k.\, n\ \mathit{succ}\ k$$

You are invited to verify the correctness of this definition by calculation. Similarly:

$$\begin{array}{lcl} \mathit{times} & = & \lambda n.\, \lambda k.\, n\ (\mathit{plus}\ k)\ \mathit{zero} \\ \mathit{exp} & = & \lambda b.\, \lambda e.\, e\ (\mathit{times}\ b)\ (\mathit{succ}\ \mathit{zero}) \end{array}$$

4 The Schema of Iteration

As we saw in the first lecture, a natural number n is represented by a function $\overline{n}$ that iterates its first argument n times applied to the second: $\overline{n}\, g\, c = \underbrace{g\,(\ldots(g}_{n \text{ times}}\ c))$. Another way to specify such a function schematically is

$$\begin{array}{lcl} f\, 0 & = & c \\ f\,(n+1) & = & g\,(f\, n) \end{array}$$

If a function satisfies such a *schema of iteration* then it can be defined in the λ-calculus on Church numerals as

$$f = \lambda n.\, n\, g\, c$$

which is easy to verify. The class of function definable this way is *total* (that is, defined on all natural numbers if c and g are), which can easily be proved by induction on n. Returning to examples from the last lecture, let's consider multiplication again.

$$\begin{array}{lcl} \mathit{times}\ 0\, k & = & 0 \\ \mathit{times}\ (n+1)\, k & = & k + \mathit{times}\, n\, k \end{array}$$

This doesn't exactly fit our schema because k is an additional parameter. That's usually allowed for iteration, but to avoid generalizing our schema the *times* function can just return a *function* by abstracting over k.

$$\begin{array}{lcl} \mathit{times}\ 0 & = & \lambda k.\, 0 \\ \mathit{times}\ (n+1) & = & \lambda k.\, k + \mathit{times}\, n\, k \end{array}$$

We can read off the constant c and the function g from this schema

$$\begin{array}{lcl} c & = & \lambda k.\, \mathit{zero} \\ g & = & \lambda r.\, \lambda k.\, \mathit{plus}\, k\, (r\, k) \end{array}$$

and we obtain

$$\mathit{times} = \lambda n.\, n\, (\lambda r.\, \lambda k.\, \mathit{plus}\, k\, (r\, k))\, (\lambda k.\, \mathit{zero})$$

which is more complicated than the solution we constructed by hand

$$\begin{array}{lcl} \mathit{plus} & = & \lambda n.\, \lambda k.\, n\ \mathit{succ}\ k \\ \mathit{times}' & = & \lambda n.\, \lambda k.\, n\ (\mathit{plus}\ k)\ \mathit{zero} \end{array}$$

The difference in the latter solution is that it takes advantage of the fact that k (the second argument to *times*) never changes during the iteration. We have repeated here the definition of *plus*, for which there is a similar choice between two versions as for *times*.

5 The Schema of Primitive Recursion

It is easy to define very fast-growing functions by iteration, such as the exponential function, or the "stack" function iterating the exponential.

$$\begin{array}{lcl} \mathit{exp} & = & \lambda b.\, \lambda e.\, e\ (\mathit{times}\ b)\ (\mathit{succ}\ \mathit{zero}) \\ \mathit{stack} & = & \lambda b.\, \lambda n.\, n\ (\mathit{exp}\ b)\ (\mathit{succ}\ \mathit{zero}) \end{array}$$

Everything appears to be going swimmingly until we think of a very simple function, namely the predecessor function defined by

$$\begin{array}{lcl} \mathit{pred}\, 0 & = & 0 \\ \mathit{pred}\, (n+1) & = & n \end{array}$$

You may try for a while to see if you can define the predecessor function, but it is difficult. The problem is that we have to go from $\lambda s.\, \lambda z.\, s\, (\ldots (s\, z))$ to $\lambda s.\, \lambda z.\, s\, (\ldots z)$, that is, we have to *remove* an s rather than add an s as was required for the successor. One possible way out is to change representation and define $\overline{n}$ differently so that predecessor becomes easy (see Exercise 3). We run the risk that other functions then become more difficult to define, or that the representation is larger than the already inefficient unary representation already is. We follow a different path, keeping the representation the same and defining the function directly.

We can start by assessing why the schema of iteration does not immediately apply. The problem is that in

$$\begin{array}{lcl} f\; 0 & = & c \\ f\; (n+1) & = & g\; (f\; n) \end{array}$$

the function g only has access to the result of the recursive call of f on n, but not to the number n itself. What we would need is the *schema of primitive recursion*:

$$\begin{array}{lcl} f\; 0 & = & c \\ f\; (n+1) & = & h\; n\; (f\; n) \end{array}$$

where n is passed to h. For example, for the predecessor function we have $c = 0$ and $h = \lambda x.\, \lambda y.\, x$ (we do not need the result of the recursive call, just n which is the first argument to h).

5.1 Defining the Predecessor Function

Instead of trying to solve the general problem of how to implement primitive recursion, let's define the predecessor directly. Mathematically, we write $n \dot{-} 1$ for the predecessor (that is, $0 \dot{-} 1 = 0$ and $n + 1 \dot{-} 1 = n$). The key idea is to gain access to n in the schema of primitive recursion by *rebuilding it* during the iteration. This requires *pairs*, a representation of which we will construct shortly.

Our specification then is

$$\mathit{pred}_2\, n = \langle n, n \dot{-} 1 \rangle$$

and the key step in its implementation in the λ-calculus is to express the definition by a schema of *iteration* rather than *primitive recursion*. The start is easy:

$$\mathit{pred}_2\, 0 = \langle 0, 0\rangle$$

For $n+1$ we need to use the value of $\mathit{pred}_2\, n$. For this purpose we assume we have a function *letpair* where

$$\mathit{letpair}\,\langle e_1, e_2\rangle\, k = k\, e_1\, e_2$$

In other words, *letpair* passes the elements of the pair to a "continuation" k. Using *letpair* we start as

$$\mathit{pred}_2\,(n+1) = \mathit{letpair}\,(\mathit{pred}_2\, n)\,(\lambda x.\, \lambda y.\, \ldots)$$

If pred_2 satisfies it specification then reduction will substitute n for x and $n \dot{-} 1$ for y. From these we need to construct the pair $\langle n+1, n\rangle$ which we can do, for example, with $\langle x+1, x\rangle$. This gives us

$$\begin{array}{lcl}
\mathit{pred}_2\, 0 & = & \langle 0, 0\rangle \\
\mathit{pred}_2\,(n+1) & = & \mathit{letpair}\,(\mathit{pred}_2\, n)\,(\lambda x.\, \lambda y.\, \langle x+1, x\rangle) \\
 & & \\
\mathit{pred}\, n & = & \mathit{letpair}\,(\mathit{pred}_2\, n)\,(\lambda x.\, \lambda y.\, y)
\end{array}$$

5.2 Defining Pairs

The next question is how to define pairs and *letpair*. The idea is to simply abstract over the continuation itself! Then *letpair* isn't really needed because the functional representation of the pair itself will apply its argument to the two components of the pair, but if want to write it out it would be the identity.

$$\begin{array}{lcl}
\overline{\langle x, y\rangle} & = & \lambda k.\, k\, x\, y \\
\mathit{pair} & = & \lambda x.\, \lambda y.\, \lambda k.\, k\, x\, y \\
\mathit{letpair} & = & \lambda p.\, p
\end{array}$$

5.3 Proving the Correctness of the Predecessor Function

Summarizing the above and expanding the definition of *letpair* we obtain

$$\begin{array}{l}
\mathit{pred}_2 = \lambda n.\, n\,(\lambda p.\, p\,(\lambda x.\, \lambda y.\, \mathit{pair}\,(\mathit{succ}\, x)\, x))\,(\mathit{pair}\, \mathit{zero}\, \mathit{zero}) \\
\mathit{pred} = \lambda n.\, \mathit{pred}_2\, n\,(\lambda x.\, \lambda y.\, y)
\end{array}$$

Let's do a rigorous proof of correctness of *pred*.[1] For the representation of natural numbers, it is convenient to assume its correctness in the form

$$\begin{array}{lll} \overline{0}\, g\, c & =_\beta & c \\ \overline{n+1}\, g\, c & =_\beta & g\,(\overline{n}\, g\, c) \end{array}$$

Lemma 1 $\mathit{pred}_2\, \overline{n} =_\beta \overline{\langle n, n \dot{-} 1\rangle}$

Proof: By mathematical induction on n.

Base: $n = 0$. Then

$$\begin{array}{llr} \mathit{pred}_2\, \overline{0} & =_\beta \overline{0}\,(\ldots)\,(\mathit{pair\, zero\, zero}) & \\ & =_\beta \mathit{pair\, zero\, zero} & \text{By repn. of } 0 \\ & =_\beta \overline{\langle 0, 0\rangle} = \overline{\langle 0, 0 \dot{-} 1\rangle} & \text{By repn. of } 0 \text{ and pairs} \end{array}$$

Step: $n = m + 1$. Then

$$\begin{array}{llr} \mathit{pred}_2\, \overline{m+1} & =_\beta \overline{m+1}\,(\lambda p.\, p\,(\lambda x.\, \lambda y.\, \mathit{pair}\,(\mathit{succ}\, x)\, x))\,(\mathit{pair\, zero\, zero}) & \\ & =_\beta (\lambda p.\, p\,(\lambda x.\, \lambda y.\, \mathit{pair}\,(\mathit{succ}\, x)\, x))\,(\overline{m}\,(\lambda p.\, \ldots)\,(\ldots)) & \text{By repn. of } m+1 \\ & =_\beta (\lambda p.\, p\,(\lambda x.\, \lambda y.\, \mathit{pair}\,(\mathit{succ}\, x)\, x))\,(\mathit{pred}_2\, \overline{m}) & \text{By defn. of } \mathit{pred}_2 \\ & =_\beta (\lambda p.\, p\,(\lambda x.\, \lambda y.\, \mathit{pair}\,(\mathit{succ}\, x)\, x))\, \overline{\langle m, m \dot{-} 1\rangle} & \text{By ind. hyp. on } m \\ & =_\beta \overline{\langle m, m \dot{-} 1\rangle}\,(\lambda x.\, \lambda y.\, \mathit{pair}\,(\mathit{succ}\, x)\, x) & \\ & =_\beta \mathit{pair}\,(\mathit{succ}\, \overline{m})\, \overline{m} & \text{By repn. of pairs} \\ & =_\beta \overline{\langle m+1, m\rangle} & \text{By repn. of successor and pairs} \\ & = \overline{\langle m+1, (m+1) \dot{-} 1\rangle} & \text{By defn. of } \dot{-} \end{array}$$

□

Theorem 2 $\mathit{pred}\, \overline{n} =_\beta \overline{n \dot{-} 1}$

Proof: Direct, from Lemma 1.

$$\begin{array}{llr} \mathit{pred}\, \overline{n} & = (\lambda n.\, \mathit{pred}_2\, n\,(\lambda x.\, \lambda y.\, y))\, \overline{n} & \\ & =_\beta \mathit{pred}_2 \overline{n}\,(\lambda x.\, \lambda y.\, y) & \\ & =_\beta \overline{\langle n, n \dot{-} 1\rangle}\,(\lambda x.\, \lambda y.\, y) & \text{By Lemma 1} \\ & =_\beta (\lambda k.\, k\, \overline{n}, \overline{n \dot{-} 1})\,(\lambda x.\, \lambda y.\, y) & \text{By repn. of pairs} \\ & =_\beta \overline{n \dot{-} 1} & \end{array}$$

□

An interesting consequence of the Church-Rosser Theorem is that if $e =_\beta e'$ where e' is in normal form, then $e \longrightarrow_\beta^* e'$.

[1] We did not carry out this proof in lecture relying on intuition and testing instead.

5.4 General Primitive Recursion

The general case of primitive recursion follows by a similar argument. Recall

$$\begin{array}{lcl} f\;0 & = & c \\ f\;(n+1) & = & h\;n\;(f\;n) \end{array}$$

We begin by defining a function f_2 specified with

$$f_2\,n = \langle n, f\,n\rangle$$

We can define f_2 using the schema of iteration.

$$\begin{array}{lcl} f_2\,0 & = & \langle 0, c\rangle \\ f_2\,(n+1) & = & \mathit{letpair}\,(f_2\,n)\,(\lambda x.\,\lambda y.\,\langle x+1, h\,x\,y\rangle) \\ f\,n & = & \mathit{letpair}\,(f_2\,n)\,(\lambda x.\,\lambda y.\,x) \end{array}$$

To put this all together, we implement a function specified with

$$\begin{array}{lcl} f\;0 & = & c \\ f\;(n+1) & = & h\;n\;(f\;n) \end{array}$$

with the following definition in terms of c and h:

$$\begin{array}{lcl} \mathit{pair} & = & \lambda x.\,\lambda y.\,\lambda g.\,g\,x\,y \\ f_2 & = & \lambda n.\,n\,(\lambda r.\,r\;(\lambda x.\,\lambda y.\,\mathit{pair}\;(\mathit{succ}\;x)\;(h\;x\;y)))\,(\mathit{pair}\;\mathit{zero}\;c) \\ f & = & \lambda n.\,f_2\,n\;(\lambda x.\,\lambda y.\,y) \end{array}$$

Recall that for the concrete case of the predecessor function we have $c = 0$ and $h = \lambda x.\,\lambda y.\,x$.

6 The Significance of Primitive Recursion

We have used primitive recursion here only as an aid to see how we can define functions in the pure λ-calculus. However, when computing over natural numbers we can restrict the functions that can be formed in schematic ways to obtain a language in which all functions terminate. Primitive recursion plays a central role in this because if c and g are terminating then so is f formed from them by primitive recursion. This is easy to see by induction on n.

In this ways we obtain a very rich set of functions but we couldn't use them to fully simulate Turing machines, for example.

Furthermore, if we give a so-called *constructive* proof of a statement in certain formulations of arithmetic with mathematical induction, we can extract a function that is defined by primitive recursion. We will probably not have an opportunity to discuss this observation further in this course, but it is an important topic in the course 15-317/15-657 *Constructive Logic*.

7 A Few Somewhat More Rigorous Definitions

We write out some definitions for notions from the first two lectures a little more rigorously.

λ-Expressions. First, the abstract syntax.

$$\begin{array}{lll} \text{Variables} & x & \\ \text{Expressions} & e & ::= \quad \lambda x.\, e \mid e_1\, e_2 \mid x \end{array}$$

$\lambda x.\, e$ *binds* x with scope e. In the concrete syntax, the scope of a binder λx is as large as possible while remaining consistent with the given parentheses so $y\,(\lambda x.\, x\, x)$ stands for $y\,(\lambda x.\,(x\, x))$. Juxtaposition $e_1\, e_2$ is left-associative so $e_1\, e_2\, e_3$ stands for $(e_1\, e_2)\, e_3$.

We define $\mathsf{FV}(e)$, the *free variables* of e with

$$\begin{array}{lcl} \mathsf{FV}(x) & = & \{x\} \\ \mathsf{FV}(\lambda x.\, e) & = & \mathsf{FV}(e) \backslash \{x\} \\ \mathsf{FV}(e_1\, e_2) & = & \mathsf{FV}(e_1) \cup \mathsf{FV}(e_2) \end{array}$$

Renaming. Proper treatment of names in the λ-calculus is notoriously difficult to get right, and even more difficult when one *reasons about* the λ-calculus. A key convention is that "*variable names do not matter*", that is, we actually *identify expressions that differ only in the names of their bound variables*. So, for example, $\lambda x.\, \lambda y.\, x\, z = \lambda y.\, \lambda x.\, y\, z = \lambda u.\, \lambda w.\, u\, z$. The textbook defines *fresh renamings* [Har16, pp. 8–9] as bijections between sequences of variables and then α-conversion based on fresh renamings. Let's take this notion for granted right now and write $e =_\alpha e'$ if e and e' differ only in the choice of names for their bound variables and this observation is important. From now on we identify e and e' if they differ only in the names of their bound variables, which means that other operations such as substitution and β-conversion are defined on α-equivalence classes of expressions.

Substitution. We can now define *substitution of e' for x in e*, written $[e'/x]e$, following the structure of e.

$$\begin{array}{lcll}
[e'/x]x & = & e' & \\
[e'/x]y & = & y & \text{for } y \neq x \\
[e'/x](\lambda y.\, e) & = & \lambda y.[e'/x]e & \text{provided } y \notin \mathsf{FV}(e') \\
[e'/x](e_1\, e_2) & = & ([e'/x]e_1)\, ([e'/x]e_2) &
\end{array}$$

This looks like a partial operation, but since we identify terms up to α-conversion we can always rename the bound variable y in $[e'/x](\lambda y.\, e)$ to another variable that is not free in e' or e. Therefore, substitution is a *total function* on α-equivalence classes of expressions.

Now that we have substitution, we also characterize α-conversion as $\lambda x.\, e =_\alpha \lambda y.\, [y/x]e$ provided $y \notin \mathsf{FV}(e)$ but as a definition it would be circular because we already required renaming to define substitution.

Equality. We can now define β- and η-conversion. We understand these conversion rules as defining a *congruence*, that is, we can apply an equation anywhere in an expression that matches the left-hand side of the equality. Moreover, we extend them to be reflexive, symmetric, and transitive so we can write $e =_\beta e'$ if we can go between e and e' by multiple steps of β-conversion.

$$\begin{array}{llcll}
\beta\text{-conversion} & (\lambda x.\, e)\, e' & =_\beta & [e'/x]e & \\
\eta\text{-conversion} & \lambda x.\, e\, x & =_\eta & e & \text{provided } x \notin \mathsf{FV}(e)
\end{array}$$

Reduction. Computation is based on reduction, which applies β-conversion in the left-to-right direction. In the pure calculus we also treat it as a congruence, that is, it can be applied anywhere in an expression.

$$\beta\text{-reduction} \quad (\lambda x.\, e)\, e' \quad \longrightarrow_\beta \quad [e'/x]e$$

Sometimes we like to keep track of length of reduction sequences so we write $e \longrightarrow_\beta^n e'$ if we can go from e to e' with n steps of β-reduction, and $e \longrightarrow_\beta^* e'$ for an arbitrary n (including 0).

Confluence. The Church-Rosser property (also called confluence) guarantees that the normal form of a λ-expression is unique, if it exists.

Theorem 3 (Church-Rosser [CR36]) *If $e \longrightarrow_\beta^* e_1$ and $e \longrightarrow_\beta^* e_2$ then there exists an e' such that $e_1 \longrightarrow_\beta^* e'$ and $e_2 \longrightarrow_\beta^* e'$.*

Exercises

Exercise 1 Analyze whether $B\ I\ f \overset{?}{=} f$ and, if so, whether it requires only β-conversion or $\beta\eta$-conversion.

Exercise 2 *Once we can define each individual instance of the schemas of iteration and primitive recursion, we can also define them explicitly as combinators.*

Define combinators iter and primrec such that

(i) The function iter $g\ c$ satisfies the schema of iteration

(ii) The function primrec $h\ c$ satisfies the schema of primitive recursion

You do not need to prove the correctness of your definitions.

Exercise 3 One approach to representing functions defined by the schema of primitive recursion is to change the representation so that $\overline{n}$ is not an iterator but a *primitive recursor*.

$$\begin{array}{lcl} \overline{0} & = & \lambda s.\, \lambda z.\, z \\ \overline{n+1} & = & \lambda s.\, \lambda z.\, s\, \overline{n}\, (\overline{n}\, s\, z) \end{array}$$

1. Define the successor function *succ* (if possible) and show its correctness.
2. Define the predecessor function *pred* (if possible) and show its correctness.
3. Explore if it is possible to directly represent any function f specified by a schema of primitive recursion, ideally without constructing and destructing pairs.

Exercise 4 The unary representation of natural numbers requires tedious and error-prone counting to check whether your functions (such a factorial, Fibonacci, or greatest common divisor in the exercises below) behave correctly on some inputs with large answers. Fortunately, you can exploit that the LAMBDA implementation counts the number or reduction steps for you and prints it in decimal form!

(i) We have

$$\overline{n}\ \mathit{succ}\ \mathit{zero} \longrightarrow_{\beta}^{*} \overline{n}$$

because $\overline{n}$ iterates the successor function n times on 0. Run some experiments in LAMBDA and conjecture how many leftmost-outermost reduction steps are required as a function of n. Note that only β-reductions are counted, and *not* replacing a definition (for example, *zero* by $\lambda s.\, \lambda z.\, z$). We justify this because we think of the definitions as taking place at the metalevel, in our mathematical domain of discourse.

(ii) Prove your conjecture from part (i), using induction on n. It may be helpful to use the mathematical notation $f^k c$ to describe a λ-expression generated by $f^0\, c = c$ and $f^{k+1}\, c = f\,(f^k\, c)$ where f and c are λ-expressions. For example, $\overline{n} = \lambda s.\, \lambda z.\, s^n\, z$ or $\mathit{succ}^3\ \mathit{zero} = \mathit{succ}\ (\mathit{succ}\ (\mathit{succ}\ \mathit{zero}))$.

Exercise 5 Define the following functions in the λ-calculus using the LAMBDA implementation. Here we take "=" to mean $=_\beta$, that is, β-conversion.

You may use all the functions in nat.lam as helper functions. Your functions should evidently reflect iteration, primitive recursion and pairs. In particular, you should avoid the use of the Y combinator which will be introduced in Lecture 3.

Provide at least 3 test cases for each function.

(i) if0 (definition by cases) satisfying the specification

$$\begin{array}{lcl} \mathsf{if0}\ \overline{0}\ x\ y & = & x \\ \mathsf{if0}\ \overline{k+1}\ x\ y & = & y \end{array}$$

(ii) even satisfying the specification

$$\begin{array}{lcl} \mathsf{even}\ \overline{2k} & = & \mathsf{true} \\ \mathsf{even}\ \overline{2k+1} & = & \mathsf{false} \end{array}$$

(iii) half satisfying the specification

$$\begin{array}{lcl} \mathsf{half}\ \overline{2k} & = & k \\ \mathsf{half}\ \overline{2k+1} & = & k \end{array}$$

Exercise 6 The Lucas function (a variant on the Fibonacci function) is defined mathematically by

$$\begin{array}{lcl} \mathsf{lucas}\ 0 & = & 2 \\ \mathsf{lucas}\ 1 & = & 1 \\ \mathsf{lucas}\ (n+2) & = & \mathsf{lucas}\ n + \mathsf{lucas}\ (n+1) \end{array}$$

Give an implementation of the Lucas function in the λ-calculus via the LAMBDA implementation.

You may use the functions from nat.lam as helper functions, as well as those from Exercise 5. Your functions should evidently reflect iteration, primitive recursion and pairs. In particular, you should avoid the use of the Y combinator which will be introduced in Lecture 3.

Test your implementation on inputs 0, 1, 9, and 11, expecting results 2, 1, 76, and 199. Include these tests in your code submission, and record the number of β-reductions used by your function.

Exercise 7 We can define binomial coefficients $\mathsf{bin}\ n\ k$ by the following recurrence:

$$\begin{array}{lcl} \mathsf{bin}\ 0\ k & = & 1 \\ \mathsf{bin}\ (n+1)\ 0 & = & 1 \\ \mathsf{bin}\ (n+1)\ (k+1) & = & \mathsf{bin}\ n\ k + \mathsf{bin}\ n\ (k+1) \end{array}$$

Give an implementation of the bin function in the λ-calculus via the LAMBDA implementation.

You may use the functions from nat.lam as helper functions, as well as those from Exercise 5. Your functions should evidently reflect iteration, primitive recursion and pairs. In particular, you should avoid the use of the Y combinator which will be introduced in Lecture 3.

Provide at least 5 test cases.

References

[CR36] Alonzo Church and J.B. Rosser. Some properties of conversion. *Transactions of the American Mathematical Society*, 39(3):472–482, May 1936.

[Har16] Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, second edition, April 2016.