Lecture Notes on Recursion

15-814: Types and Programming Languages
Frank Pfenning

Lecture 2
Thu Aug 28, 2025

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* and finally the fully general scheme of *recursion*. With the first two we always define total functions from total functions, while with arbitrary recursion we can define partial functions. This is necessary in order to capture all partial recursive functions, which are the same as can be computed by Turing machines.

2 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.

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

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

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):

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

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)

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

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.

$$plus = \lambda n.\,\lambda k.\,n\ succ\ k$$

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

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

3 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.

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

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.

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

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*.

4 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).

4.1 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

$$pred_2\, n = \langle n, n \dot{-} 1 \rangle$$

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:

$$pred_2\, 0 = \langle 0, 0 \rangle$$

For $n + 1$ we need to use the value of $pred_2\, n$. For this purpose we assume we have a function *split* where

$$split\, \langle e_1, e_2 \rangle\, k = k\, e_1\, e_2$$

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

$$pred_2\, (n + 1) = split\, (pred_2\, n)\, (\lambda x.\, \lambda y.\, \ldots)$$

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

$$\begin{array}{lcl} pred_2\, 0 & = & \langle 0, 0 \rangle \\ pred_2\, (n+1) & = & split\, (pred_2\, n)\, (\lambda x.\, \lambda y.\, \langle x+1, x \rangle) \\ \\ pred\, n & = & split\, (pred_2\, n)\, (\lambda x.\, \lambda y.\, y) \end{array}$$

4.2 Defining Pairs

The next question is how to define pairs and *split*. The idea is to simply abstract over the continuation itself! Then *split* 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.

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

4.3 Proving the Correctness of the Predecessor Function

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

$$\begin{array}{l} pred_2 = \lambda n.\, n\, (\lambda p.\, p\, (\lambda x.\, \lambda y.\, pair\, (succ\, x)\, x))\, (pair\, zero\, zero) \\ pred = \lambda n.\, pred_2\, n\, (\lambda x.\, \lambda y.\, y) \end{array}$$

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

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

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

Lemma 1 $pred_2\,\overline{n} =_\beta \overline{\langle n, n \dot{-} 1\rangle}$

Proof: By mathematical induction on n.

Base: $n = 0$. Then

$$\begin{array}{lll} pred_2\,\overline{0} & =_\beta \overline{0}\,(\ldots)\,(pair\,zero\,zero) & \\ & =_\beta pair\,zero\,zero & \text{By repn. of } 0 \\ & =_\beta \overline{\langle 0,0\rangle} = \overline{\langle 0, 0 \dot{-} 1\rangle} & \text{By repn. of 0 and pairs} \end{array}$$

Step: $n = m + 1$. Then

$$\begin{array}{lll} pred_2\,\overline{m+1} & =_\beta \overline{m+1}\,(\lambda p.\,p\,(\lambda x.\,\lambda y.\,pair\,(succ\,x)\,x))\,(pair\,zero\,zero) & \\ & =_\beta (\lambda p.\,p\,(\lambda x.\,\lambda y.\,pair\,(succ\,x)\,x))\,(\overline{m}\,(\lambda p.\,\ldots)\,(\ldots)) & \text{By repn. of } m+1 \\ & =_\beta (\lambda p.\,p\,(\lambda x.\,\lambda y.\,pair\,(succ\,x)\,x))\,(pred_2\,\overline{m}) & \text{By defn. of } pred_2 \\ & =_\beta (\lambda p.\,p\,(\lambda x.\,\lambda y.\,pair\,(succ\,x)\,x))\,\overline{\langle m, m \dot{-} 1\rangle} & \text{By ind. hyp. on } m \\ & =_\beta \overline{\langle m, m \dot{-} 1\rangle}\,(\lambda x.\,\lambda y.\,pair\,(succ\,x)\,x) & \\ & =_\beta pair\,(succ\,\overline{m})\,\overline{m} & \text{By repn. of pairs} \\ & =_\beta \overline{\langle m+1, m\rangle} & \text{By repn. of successor and pairs} \\ & = \overline{\langle m+1, (m+1) \dot{-} 1\rangle} & \text{By defn. of } \dot{-} \end{array}$$

□

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

Proof: Direct, from Lemma 1.

$$\begin{array}{lll} pred\,\overline{n} & = (\lambda n.\,pred_2\,n\,(\lambda x.\,\lambda y.\,y))\,\overline{n} & \\ & =_\beta pred_2\overline{n}\,(\lambda x.\,\lambda y.\,y) & \\ & =_\beta \overline{\langle n, n \dot{-} 1\rangle}\,(\lambda x.\,\lambda y.\,y) & \text{By Lemma 1} \\ & =_\beta (\lambda k.\,k\,\overline{n}, \overline{n \dot{-} 1})\,(\lambda x.\,\lambda y.\,y) & \text{By repn. of pairs} \\ & =_\beta \overline{n \dot{-} 1} & \end{array}$$

□

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

4.4 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.

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

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:

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

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

5 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*.

6 General Recursion

Recall the schemas of iteration and primitive recursion:

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

We have already seen how functions defined by iteration and primitive recursion can be represented in the λ-calculus. We can also see that functions defined in this manner are total as long as c, g, and h are.

But there are many functions that do not fit such of schema, for two reasons: (1) their natural presentation differs from the rigid schema (even if there actually is one that fits it), and (2) they simply fall out of the class of functions. An example of (1) is below; an example of (2) would be a function simulating a Turning machine. Since setting up a representation of Turing machines is tedious, we just show simple examples of (1).

Let's consider the subtraction-based specification of a *gcd* function for the greatest common divisor of strictly positive natural numbers $a, b > 0$.

$$\begin{array}{lcll} gcd\ a\ a & = & a & \\ gcd\ a\ b & = & gcd\ (a-b)\ b & \text{if } a > b \\ gcd\ a\ b & = & gcd\ a\ (b-a) & \text{if } b > a \end{array}$$

Why is this correct? First, the result of $gcd\ a\ b$ is a divisor of both a and b. This is clearly true in the first clause. For the second clause, assume c is a common divisor of a and b. Then there are n and k such that $a = n \times c$ and $b = k \times c$. Then $a - b = (n-k) \times c$ (defined because $a > b$ and therefore $n > k$) so c still divides both $a - b$ and b. In the last clause the argument is symmetric. It remains to show that the function terminates, but this holds because the sum of the arguments to *gcd* becomes strictly smaller in each recursive call because $a, b > 0$.

While this function looks simple and elegant, it does not fit the schema of iteration or primitive recursion. The problem is that the recursive calls are not just on the immediate predecessor of an argument, but on the results of subtraction. So it might look like

$$f\ n = h\ n\ (f\ (g\ n))$$

but that doesn't fit exactly, either, because the recursive calls to *gcd* are on different functions in the second and third clauses.

So, let's be bold! The most general schema we might think of is

$$f = h\ f$$

which means that in the right-hand side we can make arbitrary recursive calls to f. For the *gcd*, the function h might look something like this:

$$\begin{array}{l} h = \lambda g.\ \lambda a.\ \lambda b.\ \mathit{if}\ \ (a = b)\ a \\ \qquad\qquad\qquad\quad (\mathit{if}\ \ (a > b)\ (g\ (a-b)\ b) \\ \qquad\qquad\qquad\qquad\ (g\ a\ (b-a))) \end{array}$$

Here, we assume functions for testing $x = y$ and $x > y$ on natural numbers, for subtraction $x - y$ (assuming $x > y$) and for conditionals.

The interesting question now is if we can in fact define an f explicitly when given h so that it satisfies $f = h\ f$. We say that f is a *fixed point* of h, because when we apply h to f we get f back. Since our solution should be in the λ-calculus, it would be $f =_\beta h\ f$. A function f satisfying such an equation may *not* be uniquely determined. For example, the equation $f = f$ (so, $h = \lambda x.x$) is satisfied by every function f. On the other hand, if h is a constant function such as $\lambda x.I$ then $f =_\beta (\lambda x.\ I)\ f =_\beta I$ has a simple unique solution. For the purpose of this lecture, any function that satisfies the given equation is acceptable.

If we believe in the Church-Turing thesis, then any partial recursive function should be representable on Church numerals in the λ-calculus, so there is reason to hope there are explicit representations for such f. The answer is given by the so-called Y combinator.[2] Before we write it out, let's reflect on which laws Y should satisfy? We want that if $f = Y\ h$ and we specified that $f = h\ f$, so we get $Y\ h = h\ (Y\ h)$. We can iterate this reasoning indefinitely:

$$Y\ h = h\ (Y\ h) = h\ (h\ (Y\ h)) = h\ (h\ (h\ (Y\ h))) = \ldots$$

[2]For our purposes, a *combinator* is simply a λ-expression without any free variables.

In other words, Y must iterate its argument arbitrarily many times.

The ingenious solution deposits one copy of h and the replicates $Y\,h$.

$$Y = \lambda h.\,(\lambda x.\,h\,(x\,x))\,(\lambda x.\,h\,(x\,x))$$

Here, the application $x\,x$ takes care of replicating $Y\,h$, and the outer application of h in $h\,(x\,x)$ leaves a copy of h behind. Formally, we calculate

$$\begin{array}{lll} Y\,h & =_\beta & (\lambda x.\,h\,(x\,x))\,(\lambda x.\,h\,(x\,x)) \\ & =_\beta & h\,((\lambda x.\,h\,(x\,x))\,(\lambda x.\,h\,(x\,x))) \\ & =_\beta & h\,(Y\,h) \end{array}$$

In the first step, we just unwrap the definition of Y. In the second step we perform a β-reduction, substituting $[(\lambda x.\,h\,(x\,x))/x]\,h\,(x\,x)$. In the third step we recognize that this substitution recreated a copy of $Y\,h$.

You might wonder how we could ever get an answer since

$$Y\,h =_\beta h\,(Y\,h) =_\beta h\,(h\,(Y\,h)) =_\beta h\,(h\,(h\,(Y\,h))) = \ldots$$

Well, we sometimes don't! Actually, this is important if we are to represent *partial recursive functions* which include functions that are undefined (have no normal form) on some arguments. Reconsider the specification $f = f$ as a recursion schema. Then $h = \lambda g.\,g$ and

$$Y\,h = Y\,(\lambda g.\,g) =_\beta (\lambda x.\,(\lambda g.\,g)\,(x\,x))\,(\lambda x.\,(\lambda g.\,g)\,(x\,x)) =_\beta (\lambda x.\,x\,x)\,(\lambda x.\,x\,x)$$

The term on the right-hand side here (called Ω) has the remarkable property that it only reduces to itself! It therefore does not have a normal form. In other words, the function $f = Y\,(\lambda g.\,g) = \Omega$ solves the equation $f = f$ by giving us a result which always diverges.

We do, however, sometimes get an answer. Consider, for example, a case where f does not call itself recursively at all: $f = \lambda n.\,succ\;n$. Then $h_0 = \lambda g.\,\lambda n.\,succ\;n$. And we calculate further

$$\begin{array}{lll} Y\,h_0 & = & Y\,(\lambda g.\,\lambda n.\,succ\;n) \\ & =_\beta & (\lambda x.\,(\lambda g.\,\lambda n.\,succ\;n)\,(x\,x))\,(\lambda x.\,(\lambda g.\,\lambda n.\,succ\;n)\,(x\,x)) \\ & =_\beta & (\lambda x.\,(\lambda n.\,succ\;n))\,(\lambda x.\,(\lambda n.\,succ\;n)) \\ & =_\beta & \lambda n.\,succ\;n \end{array}$$

So, fortunately, we obtain just the successor function *if we apply β-reduction from the outside in*. It is however also the case that there is an infinite reduction sequence starting at $Y\,h_0$. By the Church-Rosser Theorem (Theorem 3) this means that at any point during such an infinite reduction sequence we could still also reduce to $\lambda n.\,succ\;n$. A remarkable and nontrivial theorem about the λ-calculus is that if we always reduce the left-most/outer-most redex (which is the first expression of the form $(\lambda x.\,e_1)\,e_2$ we come to when reading an expression from left to right) then we will definitely arrive at a normal form when one exists. And by the Church-Rosser theorem such a normal form is unique (up to renaming of bound variables, as usual).

7 Defining Functions by Recursion

As a simpler example than *gcd*, consider the factorial function, which we deliberately write using general recursion rather than primitive recursion.

$\mathsf{fact}\ n = \mathbf{if}\ n = 0\ \mathbf{then}\ 1\ \mathbf{else}\ n * \mathsf{fact}(n-1)$

To write this in the λ-calculus we first define a zero test *if0* satisfying

$$\begin{aligned} \mathit{if0}\ \overline{0}\ c\ d &= c \\ \mathit{if0}\ \overline{n+1}\ c\ d &= d \end{aligned}$$

which is a special case of if iteration and can be written, for example, as

$\mathit{if0} = \lambda n.\, \lambda c.\, \lambda d.\, n\ (K\ d)\ c$

Eliminating the mathematical notation from the recursive definition of fact get the equation

$\mathsf{fact} = \lambda n.\, \mathit{if0}\ n\ (\mathit{succ}\ \mathit{zero})\ (\mathit{times}\ n\ (\mathsf{fact}\ (\mathit{pred}\ n)))$

where we have already defined *succ*, *zero*, *times*, and *pred*. Of course, this is not directly allowed in the λ-calculus since the right-hand side mentions fact which we are just trying to define. The function h_{fact} which will be the argument to the Y combinator is then

$h_{\mathsf{fact}} = \lambda f.\, \lambda n.\, \mathit{if0}\ n\ (\mathit{succ}\ \mathit{zero})\ (\mathit{times}\ n\ (f\ (\mathit{pred}\ n)))$

and

$\mathit{fact} = Y\ h_{\mathsf{fact}}$

We can write and execute this now in LAMBDA notation (see file rec.lam)

```
1  defn I = \x. x
2  defn K = \x. \y. x
3  defn Y = \h. (\x. h (x x)) (\x. h (x x))
4
5  defn if0 = \n. \c. \d. n (K d) c
6
7  defn h_fact = \f. \n. if0 n (succ zero) (times n (f (pred n)))
8  defn fact = Y h_fact
9
10 norm _120 = fact _5
11 norm _720 = fact (succ _5)
```

Listing 1: Recursive factorial in LAMBDA

8 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.

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

$\lambda x.\, e$ *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

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

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* [Harper, 2016, 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 and Rosser [1936]) *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 $B\ I\ f \stackrel{?}{=} f$ 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.

Exercise 8 Give an implementation of the factorial function in the λ-calculus as it arises from the schema of primitive recursion. How many β-reduction steps are required for factorial of 0, 1, 2, 3, 4, 5 in each of the two implementations?

Exercise 9 The Fibonacci function is defined by

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

Give two implementations of the Fibonacci function in the λ-calculus (using the LAMBDA implementation). You may use the functions in (see file rec.lam).

(i) Exploit the idea behind the encoding of primitive recursion using pairs to give a direct implementation of fib without using the Y combinator.

(ii) Give an implementation of fib using the Y combinator.

Test your implementation on inputs 0, 1, 9, and 11, expecting results 0, 1, 34, and 89. Which of the two is more "efficient" (in the sense of number of β-reductions)?

Exercise 10 Recall the specification of the greatest common divisor (*gcd*) from this lecture for natural numbers $a, b > 0$:

$$\begin{array}{lcll} gcd\ a\ a & = & a & \\ gcd\ a\ b & = & gcd\ (a-b)\ b & \text{if } a > b \\ gcd\ a\ b & = & gcd\ a\ (b-a) & \text{if } b > a \end{array}$$

We don't care how the function behaves if $a = 0$ or $b = 0$.

Define *gcd* as a closed expression in the λ-calculus over Church numerals. You may use the Y combinator we defined, and any other functions like *succ*, *pred*, and you should define other functions you may need such as subtraction or arithmetic comparisons.

Also analyze how your function behaves when one or both of the arguments a and b are $\overline{0}$.

References

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

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