Lecture Notes on Recursion

15-814: Types and Programming Languages
Frank Pfenning

Lecture 2
Thursday, September 5, 2019

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 just plan *recursion*.

2 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*.

In this latter solution, we exploit that $(\mathit{plus}\ k)$ is a function because *plus* starts with two λ-abstractions. We could also make the second argument to *plus* explicit:

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

We observe that $\mathit{plus}\ \ k$ and $\lambda u.\, \mathit{plus}\ \ k\ \ u$ always behave the same when applied to any argument e because

$$\mathit{plus}\ k\ e =_\beta (\lambda u.\, \mathit{plus}\ k\ u)\ e$$

More generally, the behavior of e and $\lambda u.\, e\ u$ is the same when applied to any argument e' as long as $u \notin \mathsf{FV}(e)$ (that is, u is not among the free variables of e, see Section 5). This law is called *η-conversion* and represents a weak form of an *extensionality principle*: two functions should be equal if they return equals results when applied to the equal arguments.

3 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 1). 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).

At first glance it seems at least plausible that we should be able to define any primitive recursive function using only the schema of iteration. Certainly, all functions in these two classes are total, as long as the component functions c, g, and h are total.

The basic idea in the representation of primitive recursion by iteration is that we need the recursive call to return not only $f\,n$ but also n itself! In other words, in order to eventually get f, we first define a function f' satisfying

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

where $\langle -, -\rangle$ forms a pair. From such a pair we can extract both n and $f\,n$ in order to pass them to h. In more detail:

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

What is *letpair*[1] supposed to do? We specify that

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

In other words, *letpair* applies the continuation k to the components of its first argument (which should be a pair). If we can define pairs and *letpair* then f' is correctly defined. Formally, we would prove this by induction on n. First, if $n = 0$ then

$$f'\,0 \quad = \quad \langle 0, c\rangle = \langle 0, f\,0\rangle$$

Second

$$\begin{array}{lcll} f'\,(n+1) & = & \mathit{letpair}\,(f'\,n)\,(\lambda x.\,\lambda r.\,\langle x+1, h\,x\,r\rangle) & \\ & =_\beta & \mathit{letpair}\,\langle n, f\,n\rangle\,(\lambda x.\,\lambda r.\,\langle x+1, h\,x\,r\rangle) & \text{by induction hypothesis} \\ & =_\beta & \langle n+1, h\,n\,(f\,n)\rangle & \text{by reduction for } \mathit{letpair} \\ & = & \langle n+1, f\,(n+1)\rangle & \text{by definition of } f\,(n+1) \end{array}$$

It remains to give the definitions of *pair* (implementing $\langle -, -\rangle$) and *letpair*. Actually, we will do a little more, also providing explicit projections onto the first and second components of a pair. But first, we form a pair by abstracting over a function g applied to both components.

$$\mathit{pair} = \lambda x.\,\lambda y.\,\lambda g.\,g\,x\,y$$

which means that $\mathit{pair}\,e_1\,e_2 =_\beta \lambda g.\,g\,e_1\,e_2$. To extract the first component of the pair, we simply apply it to the first projection function! And for the second component we project onto the second argument.

$$\begin{array}{lcl} \mathit{fst} & = & \lambda p.\,p\,(\lambda x.\,\lambda y.\,x) = \lambda p.\,p\;\mathit{true} \\ \mathit{snd} & = & \lambda p.\,p\,(\lambda x.\,\lambda y.\,y) = \lambda p.\,p\;\mathit{false} \end{array}$$

[1] We called with *case* in lecture, but in hindsight that seems like a poor choice.

The *letpair* function is interesting. Recall that we want

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

We define

$$\mathit{letpair} \quad = \quad \lambda p.\, \lambda k.\, p\ k$$

which works because

$$\begin{array}{rcl} \mathit{letpair}\ (\mathit{pair}\ e_1\ e_2)\ k & =_\beta & \mathit{letpair}\ (\lambda g.\, g\ e_1\ e_2)\ k \\ & =_\beta & (\lambda k.\, (\lambda g.\, g\ e_1\ e_2)\ k)\ k \\ & =_\beta & k\ e_1\ e_2 \end{array}$$

One further remark here: the right-hand side in the definition of *letpair* can be simplified further using η-conversion.

$$\mathit{letpair} = \lambda p.\, \lambda k.\, p\ k =_\eta \lambda p.\, p$$

so it is the identity function! Intuitively, a pair is represented by its own destructor function, so this destructor (here *letpair*) is just the identity. Similarly, if we wanted to define an iterator function

$$\mathit{iter}\ \overline{n}\ f\ c = \underbrace{f\ (f \ldots (f}_{n \text{ times}}\ c))$$

then $\mathit{iter} = \lambda m.\, m$ will satisfy this equation for Church numerals.

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:

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

Eliminating $\mathit{letpair} = \lambda p.\, p$ we obtain the slightly shorter version

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

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

$$\begin{array}{lcl} \mathit{pred}' & = & \lambda n.\, n\ (\lambda r.\, r\ (\lambda x.\, \lambda y.\, \mathit{pair}\ (\mathit{succ}\ x)\ x))\ (\mathit{pair}\ \mathit{zero}\ \mathit{zero}) \\ \mathit{pred} & = & \lambda n.\, \mathit{pred}'\ n\ (\lambda x.\, \lambda y.\, y) \end{array}$$

4 General Recursion

Schematic function definitions (even at the generality of primitive recursion) can be restrictive. Let's consider the subtraction-based specification of a *gcd* function for the greatest common divisor of strictly positive natural numbers $a, b > 0$.

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

Why is this correct? First, the result of $\mathit{gcd}\ a\ b$ 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:

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

Here, we assume functions for testing $x = y$ and $x > y$ on natural numbers, for subtraction $x - y$ (assuming $x > y$) and for conditionals *if* $b\ e_1\ e_2$ where *if true* $e_1\ e_2 =_\beta e_1$ and *if false* $e_1\ e_2 =_\beta e_2$ (see Exercise 2).

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. 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:

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

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

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

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

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 recursive at all: $f = \lambda n.\, \mathit{succ}\ n$. Then $h_0 = \lambda g.\, \lambda n.\, \mathit{succ}\ n$. And we calculate further

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

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 1 this means that at any point during such an infinite reduction sequence we could still also reduce to $\lambda n.\, \mathit{succ}\ n$. 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).

5 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.

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

$\lambda x.\, e$ *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

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

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.

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

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 1 (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 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 2 We know we can represent all functions on Booleans returning Booleans once we have exclusive or. But we can also represent the more general conditional *if* with the requirements

$$\begin{array}{lcl} \mathit{if}\ \mathit{true}\ e_1\ e_2 & = & e_1 \\ \mathit{if}\ \mathit{false}\ e_1\ e_2 & = & e_2 \end{array}$$

Give a definition of *if* in the λ-calculus and verify (showing each step) that the equations above are satisfied using β-conversion.

Exercise 3 Recall the specification of the greatest common divisor (*gcd*) from this lecture for natural numbers $a, b > 0$:

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

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*, etc. from this lecture and *if* from Exercise 2, but you have to define other functions you may need such as subtraction or arithmetic comparisons.

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

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.