# Lecture Notes on Recursion

15-814: Types and Programming Languages
Frank Pfenning

Lecture 3
Tuesday, September 8, 2020

## 1 Introduction

In this lecture we first complete our development of recursion: from iteration through primitive recursion to full recursion. Then we will introduce *simple types* to sort out our data representations.

## 2 General Recursion

Recall the schemas of iteration and primitive recursion:

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

We have already seen how functions defined by iteration and primitive recursion can be represented in the $\lambda$-calculus. We can also see that functions defined in this manner are terminating as long as $c$ and $g$ 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 $a, b > 0$.

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

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 (see Exercise L1.4).

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 $\lambda$-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 $\lambda$-calculus, so there is reason to hope there are explicit representations for such $f$. The answer is given by the so-called $Y$ combinator.[1] 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 $\beta$-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 $\Omega$) has the remarkable property that it only reduces to itself! It therefore does not have a normal form. In

[1] For our purposes, a *combinator* is simply a $\lambda$-expression without any free variables.

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: $f = \lambda n.\, \mathit{succ}\ n$. 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 $\beta$-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 L2.3) 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 $\lambda$-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).

## 3 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 $\lambda$-calculus we first define a zero test *if0* satisfying

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

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 $\lambda$-calculus since the right-hand side mentions fact which we are just trying to define. The function $h_{\mathsf{fact}}$ 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 nat.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 label

## 4 Introduction to Types

We have experienced the expressive power of the $\lambda$-calculus in multiple ways. We followed the slogan of *data as functions* and represented types such as Booleans and natural numbers. On the natural numbers, we were able to express the same set of partial functions as with Turing machines, which gave rise to the Church-Turing thesis that these are all the effectively computable functions.

On the other hand, Church's original purpose of the pure calculus of functions was a new foundations of mathematics distinct from set theory [Chu32, Chu33]. Unfortunately, this foundation suffered from similar paradoxes as early attempts at set theory and was shown to be *inconsistent*, that is, every proposition has a proof. Church's reaction was to return to the ideas by Russell and Whitehead [WR13] and introduce *types*. The resulting calculus, called *Church's Simple Theory of Types* [Chu40] is much simpler than

Russell and Whitehead's *Ramified Theory of Types* and, indeed, serves well as a foundation for (classical) mathematics.

We will follow Church and introduce *simple types* as a means to classify $\lambda$-expressions. An important consequence is that we can recognize the representation of Booleans, natural numbers, and other data types and distinguish them from other forms of $\lambda$-expressions. We also explore how typing interacts with computation.

## 5 Simple Types, Intuitively

Since our language of expression consists only of $\lambda$-abstraction to form functions, juxtaposition to apply functions, and variables, we would expect our language of types $\tau$ to just contain $\tau ::= \tau_1 \to \tau_2$. This type might be considered "empty" since there is no base case, so we add type variables $\alpha$, $\beta$, $\gamma$, etc.

$$\begin{array}{lll} \text{Type variables} & \alpha & \\ \text{Types} & \tau \quad ::= \quad \tau_1 \to \tau_2 \mid \alpha \end{array}$$

We follow the convention that the function type constructor "$\to$" is *right-associative*, that is, $\tau_1 \to \tau_2 \to \tau_3 = \tau_1 \to (\tau_2 \to \tau_3)$.

We write $e : \tau$ if expression $e$ has type $\tau$. For example, the identity function takes an argument of arbitrary type $\alpha$ and returns a result of the same type $\alpha$. But the type is not unique. For example, the following two hold:

$$\begin{array}{lcl} \lambda x.\, x & : & \alpha \to \alpha \\ \lambda x.\, x & : & (\alpha \to \beta) \to (\alpha \to \beta) \end{array}$$

What about the Booleans? $\mathit{true} = \lambda x.\, \lambda y.\, x$ is a function that takes an argument of some arbitrary type $\alpha$, a second argument $y$ of a potentially different type $\beta$ and returns a result of type $\alpha$. We can similarly analyze *false*:

$$\begin{array}{lcll} \mathit{true} & = & \lambda x.\, \lambda y.\, x & : \quad \alpha \to (\beta \to \alpha) \\ \mathit{false} & = & \lambda x.\, \lambda y.\, y & : \quad \alpha \to (\beta \to \beta) \end{array}$$

This looks like bad news: how can we capture the Booleans by their type if *true* and *false* have a different type? We have to realize that types are not unique and we can indeed find a type that is shared by *true* and *false*:

$$\begin{array}{lcll} \mathit{true} & = & \lambda x.\, \lambda y.\, x & : \quad \alpha \to (\alpha \to \alpha) \\ \mathit{false} & = & \lambda x.\, \lambda y.\, y & : \quad \alpha \to (\alpha \to \alpha) \end{array}$$

The type $\alpha \to (\alpha \to \alpha)$ then becomes our candidate as a type of Booleans in the $\lambda$-calculus. Before we get there, we formalize the type system so we can rigorously prove the right properties.

## 6 The Typing Judgment

We like to formalize various judgments about expressions and types in the form of inference rules. For example, we might say

$$\frac{e_1 : \tau_2 \to \tau_1 \quad e_2 : \tau_2}{e_1\ e_2 : \tau_1}$$

We usually read such rules from the conclusion to the premises, pronouncing the horizontal line as "*if*":

> *The application $e_1\ e_2$ has type $\tau_1$ if $e_1$ maps arguments of type $\tau_2$ to results of type $\tau_1$ and $e_2$ has type $\tau_2$.*

When we arrive at functions, we might attempt

$$\frac{x_1 : \tau_1 \quad e_2 : \tau_2}{\lambda x_1.\, e_2 : \tau_1 \to \tau_2}\ ?$$

This is (more or less) Church's approach. It requires that each variable $x$ intrinsically has a type that we can check, so probably we should write $x^\tau$. In modern programming languages this can be bit awkward because we might substitute for type variables or apply other operations on types, so instead we record the types of variable in a *typing context*.

$$\text{Typing context} \quad \Gamma \quad ::= \quad x_1 : \tau_1, \ldots, x_n : \tau_n$$

Critically, we always assume:

> *All variables declared in a context are distinct.*

This avoids any ambiguity when we try to determine the type of a variable. The typing judgment now becomes

$$\Gamma \vdash e : \tau$$

where the context $\Gamma$ contains declarations for the free variables in $e$. It is defined by the following three rules

$$\frac{\Gamma, x_1 : \tau_1 \vdash e_2 : \tau_2}{\Gamma \vdash \lambda x_1.\, e_2 : \tau_1 \to \tau_2}\ \mathsf{lam} \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \mathsf{var}$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1\, e_2 : \tau_1}\ \mathsf{app}$$

As a simple example, let's type-check *true*. Note that we always construct such derivations bottom-up, starting with the final conclusion, deciding on rules, writing premises, and continuing.

$$\dfrac{\dfrac{\dfrac{}{x : \alpha, y : \alpha \vdash x : \alpha}\ \mathsf{var}}{x : \alpha \vdash \lambda y.\, x : \alpha \to \alpha}\ \mathsf{lam}}{\cdot \vdash \lambda x.\, \lambda y.\, x : \alpha \to (\alpha \to \alpha)}\ \mathsf{lam}$$

In this construction we exploit that the rules for typing are *syntax-directed*: for every form of expression there is exactly one rule we can use to infer its type.

How about the expression $\lambda x.\, \lambda x.\, x$? This is $\alpha$-equivalent to $\lambda x.\, \lambda y.\, y$ and therefore should check (among other types) as having type $\alpha \to (\beta \to \beta)$. It appears we get stuck:

$$\dfrac{\dfrac{\dfrac{??}{x : \alpha \vdash \lambda x.\, x : \beta \to \beta}\ \mathsf{lam}??}{\cdot \vdash \lambda x.\, \lambda x.\, x : \alpha \to (\beta \to \beta)}\ \mathsf{lam}}$$

The worry is that applying the rule lam would violate our presupposition that no variable is declared more than once and $x : \alpha, x : \beta \vdash x : \beta$ would be ambiguous. But we said we can "*silently*" apply $\alpha$-conversion, so we do it here, renaming $x$ to $x'$. We can then apply the rule:

$$\dfrac{\dfrac{\dfrac{}{x : \alpha, x' : \beta \vdash x' : \beta}\ \mathsf{var}}{x : \alpha \vdash \lambda x.\, x : \beta \to \beta}\ \mathsf{lam}}{\cdot \vdash \lambda x.\, \lambda x.\, x : \alpha \to (\beta \to \beta)}\ \mathsf{lam}$$

A final observation here about type variables: if $\cdot \vdash e : \alpha \to (\beta \to \beta)$ then also $\cdot \vdash e : \tau_1 \to (\tau_2 \to \tau_2)$ for any types $\tau_1$ and $\tau_2$. In other words, we can *substitute* arbitrary types for type variables in a typing judgment $\Gamma \vdash e : \tau$ and still get a valid judgment. In particular, the expressions *true* and *false* have *infinitely many types*.

## 7 Type Inference

An important property of the typing rules we have so far is that they are *syntax-directed*, that is, for every form of expression there is exactly one

typing rule that can be applied. We then perform *type inference* by constructing the skeleton of the typing derivation, filling it with *unknown types*, and reading off a set of equations that have be satisfied between the unknowns. Fortunately, these equations are relatively straightforward to solve with an algorithm called *unification*. This is the core of what is used in the implementation of modern functional languages such as Standard ML, OCaml, or Haskell.

We sketch how this process work, but only for a specific example; we might return to the general algorithm form in a future lecture. Consider the representation of 2:

$$\lambda s.\, \lambda z.\, s\,(s\,z)$$

We know it must have type $?\tau_1 \to (?\tau_2 \to ?\tau_3)$ for some unknown types $?\tau_1$, $?\tau_2$, and $?\tau_3$ where

$$s : ?\tau_1, z : ?\tau_2 \vdash s\,(s\,z) : ?\tau_3$$

Now, $s\,z$ applies $s$ to $z$, so $?\tau_1 = ?\tau_2 \to ?\tau_4$ for some new $?\tau_4$. Next, the $s$ is applied to the result of $s\,z$, so $?\tau_4 = ?\tau_2$. Also, the right-hand side is the same as the result type of $s$, so $?\tau_3 = ?\tau_4 = ?\tau_2$. Substituting everything out, we obtain

$$s : ?\tau_2 \to ?\tau_2, z : ?\tau_2 \vdash s\,(s\,z) : ?\tau_2$$

It is straightforward to write down the typing derivation for this judgment. Also, because we did not need to commit to what $?\tau_2$ actually is, we obtain

$$\lambda s.\, \lambda z.\, s\,(s\,z) : (\tau_2 \to \tau_2) \to (\tau_2 \to \tau_2) \qquad \text{for any type } \tau_2$$

We can express this by using a type variable instead, writing

$$\lambda s.\, \lambda z.\, s\,(s\,z) : (\alpha \to \alpha) \to (\alpha \to \alpha) \qquad \text{for any type } \alpha$$

because if the the type of an expression contains type variables we can alway substitute arbitrary types for them and still obtain a valid type.

We find that

$$\vdash \overline{n} : (\alpha \to \alpha) \to (\alpha \to \alpha)$$

even though some of the representations (such as $\overline{0} = \textit{zero}$) also have other types. So our current hypothesis is that this type is a good candidate as a characterization of Church numerals, just as $\alpha \to (\alpha \to \alpha)$ is a characterization of the Booleans.

## Exercises

**Exercise 1** 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 $\beta$-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 $\lambda$-expression generated by $f^0\, c = c$ and $f^{k+1}\, c = f\,(f^k\, c)$ where $f$ and $c$ are $\lambda$-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 2** Give an implementation of the factorial function in the $\lambda$-calculus as it arises from the schema of primitive recursion. How many $\beta$-reduction steps are required for factorial of 0, 1, 2, 3, 4, 5 in each of the two implementations?

**Exercise 3** 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 $\lambda$-calculus (using the LAMBDA implementation). You may use the functions in (see file nat.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 $\beta$-reductions)?

**Exercise 4** 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 $\lambda$-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

[Chu32] A. Church. A set of postulates for the foundation of logic I. *Annals of Mathematics*, 33:346–366, 1932.

[Chu33] A. Church. A set of postulates for the foundation of logic II. *Annals of Mathematics*, 34:839–864, 1933.

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[WR13] Alfred North Whitehead and Bertrand Russell. *Principia Mathematica*. Cambridge University Press, 1910–13. 3 volumes.