CSE301(Functional Programming)

Lecture 3: Lambda calculus and typing

Noam Zeilberger

version: September 22, 2025

The **lambda calculus** is a formal system that was originally introduced in the late 1920s by the American logician Alonzo Church, whose motivation was to develop a foundation for mathematics more natural than Russell and Whitehead's *Principia Mathematica*. It has since found application in diverse areas, including automated theorem proving, programming language design and implementation, category theory, linguistics, and combinatorics. From the perspective of functional programming, lambda calculus can be seen as the archetypal and minimalistic example of a functional programming language, but also serves simply as a lucid notation for defining functions passed as arguments to other functions.

One important historical remark is that Church's original calculus turned out to be inconsistent as a mathematical foundation (i.e., you could use it to prove anything), which was discovered by his PhD students Stephen Kleene and Barkley Rosser.[1] Church's solution was to split the original lambda calculus in two separate systems: an *untyped* calculus that could express any functional computation but that had no pretenses to logic, and a *typed* calculus which was considerably weaker computationally yet could safely encode logical reasoning. Both ended up being extremely influential, and while the tension between the demands of computational power versus foolproof reasoning is fundamental, modern descendants of lambda calculus resolve this tension through increasingly expressive type systems, which include a large degree of **type inference**. Learning the fundamentals of simple typing is an important step to understanding such systems.

The following presentation borrows heavily from Peter Selinger's excellent lecture notes (2013). Other treatments of lambda calculus include Barendregt's classic monograph (1984) and his more recent textbook with Dekkers and Statman (2010), as well as Girard's book (1989) which is more logic-focused.

1 Lambda notation and variable binding

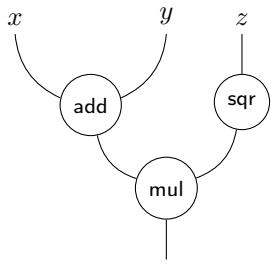
Before introducing lambda notation, we can motivate it with a brief recall of ordinary algebraic notation, for example as in the arithmetic expression

$$(x + y) \times z^2$$

[1] Cardone and Hindley's article (2006) was already mentioned in Lecture 0, but let me repeat that recommendation if you are interested in a deeper account of the early development of lambda calculus and its historical context.

in three variables x, y, z. An advantage of this notation for arithmetic expressions is that it avoids explicitly staging intermediate computations. We write « $(x + y) \times z^2$ » rather than « Let $w = x + y$, then let $u = z^2$, then let $v = w \times u$ ».

Diagrammatically, the algebraic expression may also be represented as a tree:



Observe that we labelled the three "input" edges x, y, z as well as the three operator nodes, but left the two inner edges and the root edge unlabelled. Of course we could label these edges if we wanted to, but the precise choice of labels is arbitrary as long as we pick labels distinct from the labels of the input edges. The only thing that really matters is the shape of the tree!

Similarly, rather than writing

$$\text{« Let } f \text{ be the function } x \mapsto x^2. \text{ Then consider } e = f(5)... \text{ »},$$

in the lambda calculus we can just write

$$(\lambda x.x^2)(5) \tag{1}$$

The variable x is said to be **free** in the subexpression x^2, and **bound** in $\lambda x.x^2$. Of course the variable name x is arbitrary. We could write

$$(\lambda y.y^2)(5) \tag{2}$$

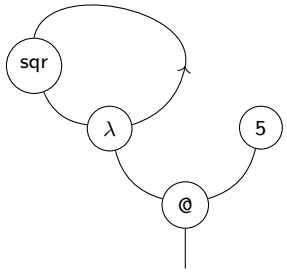
instead, which would be entirely equivalent—in the jargon of lambda calculus originally introduced by Church, (1) and (2) are said to be *α-equivalent*.

In Haskell, lambda notation is introduced using the backward slash symbol. Thus $\lambda x.x^2$ is notated $\backslash x \to x \hat{}\ 2$. This is equivalent to `fn x => x ** 2` in OCaml, or `lambda x:x ** 2` in Python.

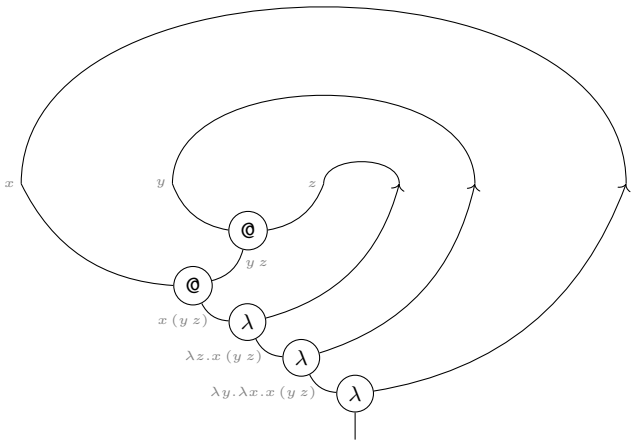
By convention, the scope of a lambda extends as far to the right as possible unless explicitly delimited by parentheses, and iterated lambdas may be shortened into a string of variables. Thus $\lambda xy.y\,x$ is equivalent to $\lambda x.(\lambda y.(y\,x))$, but distinct from $\lambda x.(\lambda y.y)\,x$. (In Haskell, iterated lambdas can be similarly coalesced: $\backslash x \to \backslash y \to x + y$ shortens to $\backslash x\ y \to x + y$, for example.)

Diagrammatically, lambda expressions may be represented as trees with "back pointers". For example, the expression (1) above can be represented by

the following graph:



Here the bound variable x in the subexpression $\lambda x.x^2$ is represented by the back pointer from the λ-node to the input of the sqr-node. This lambda expression is then fed as an input on the function side of an application node (@), which in this example takes a constant expression on its argument side. The α-equivalent expression (2) can, of course, be represented by the same graph. Here is another example graph, of the pure lambda expression $\lambda x.\lambda y.\lambda z.x\,(y\,z)$, where for clarity we have labelled all the inner edges to indicate the corresponding subexpressions:



As the diagrammatic representation makes clear,[2] what is important is the *linking relationship* between occurrences of variables and their binding lambdas.

Lambda notation is particularly useful when constructing arguments to feed to higher-order functions. Such higher-order functions are used implicitly in many different areas of science and mathematics, often with ad hoc conventions that could be clarified using lambda notation. For example, ordinary differentiation of real-valued functions may be considered as a higher-order function:

$$D = \lambda f.\lambda x.\lim_{h\to 0}\frac{f(x+h)-f(x)}{h}$$

Rather than writing, say, $\frac{d}{dx}x^2 = 2x$, we can write $D(\lambda x.x^2) = \lambda x.2x$, with the latter notation having the advantage that it makes clear that the variable x is bound in both the input function and the output function.

[2]George Kaye's lamviz (`https://www.georgejkaye.com/lamviz/visualiser/`) is a handy tool for visualizing such lambda graphs.

2 β-reduction and substitution

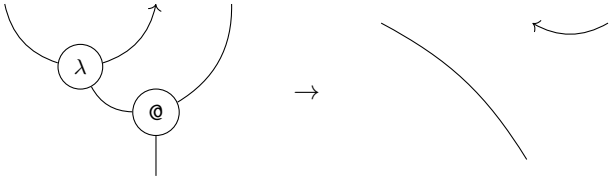
A felicitous property of lambda notation is that it tells you exactly how to *apply* the function you are defining: to apply a lambda-defined function $\lambda x.e$ to some argument d, just substitute d for x in e. Formally, this is called the rule of **β-reduction**:

$$(\lambda x.e)\, d \to e[d/x]$$

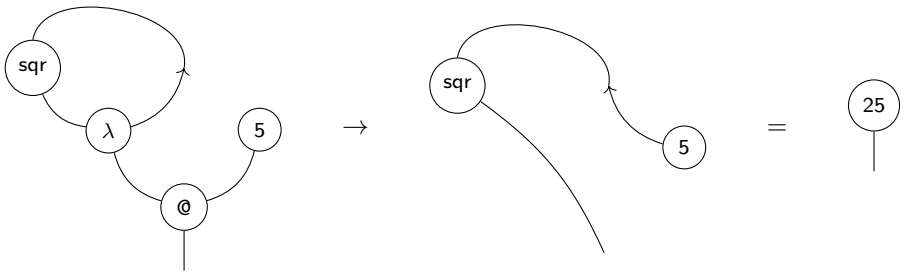
On the left hand side we have the application of a lambda expression to an argument. On the right hand side the notation "$[d/x]$" stands for the action of *substituting* the expression d for all occurrences of the variable x. For example, applying one beta reduction, we have

$$(\lambda x.x^2)\, 5 \to 5^2 = 25.$$

Graphically, the rule of beta reduction may be depicted as a kind of "unzipping" operation that rewires the graph to link the argument of an application directly to the occurrences of the variable in a function definition:



Or as illustrated on the above example:



The intuition should be clear, although making precise what this means in the general case is a little bit tricky. The usual explanation is in terms of so-called "capture-avoiding substitution", the formal definition of which we recall below.

3 Untyped lambda calculus

As a programming language, the untyped lambda calculus (also called *pure lambda calculus*) is extremely simple: it has lambda notation for defining functions, function application, and nothing else! Although making this description precise requires adding a few details, the surprising complexity that emerges from the relatively simple formal definition of the lambda calculus is a beautiful thing to encounter for the first time.

3.1 Syntax and capture-avoiding substitution

Formally, untyped lambda expressions can be defined inductively like so:

(variables) $x, y, z, \ldots$ are lambda expressions;

(application) if e and e' are lambda expressions then $e\,e'$ is a lambda expression;

(abstraction) if e is a lambda expression and x is a variable then $\lambda x.e$ is a lambda expression.

Or in more concise BNF-style notation:

$$e ::= x \mid e\ e' \mid \lambda x.e$$

Note that in a lambda abstraction $\lambda x.e$, the variable x is allowed to occur any number of times in e – or even zero times, in which case the abstraction may be thought of as defining a constant function that ignores its argument. We always consider lambda expressions modulo alpha equivalence.

As remarked earlier, to define beta reduction of lambda expressions completely precisely we need to define what we mean by substitution. Formally, $e[d/x]$ can be defined by induction on e:

$$y[d/x] = \begin{cases} d & \text{if } y = x \\ y & \text{if } y \neq x \end{cases}$$

$$(e_1\ e_2)[d/x] = (e_1[d/x])\ (e_2[d/x])$$

$$(\lambda x.e')[d/x] = \lambda x.e'$$

$$(\lambda y.e')[d/x] = \lambda y.(e'[d/x]) \qquad \text{if } x \neq y \text{ and } y \notin \text{free}(d)$$

$$(\lambda y.e')[d/x] = \lambda z.(e'\{z/y\}[d/x]) \qquad \text{if } x \neq y, y \in \text{free}(d) \text{ and } z \text{ fresh}$$

From a glance at the definition one can see that substituting into a lambda abstraction is a bit subtle, dividing into three subcases. You do not have to memorize the formal definition of substitution, but let us unpack these subcases to better understand the reasons for the complexity:

1. $e = \lambda x.e'$ is a lambda abstraction binding the substituted-for variable x: in this case, the substituted-for variable is "shadowed" by the abstraction, and we return the abstraction unchanged. An example of this case is

   $$(x\ (\lambda x.\lambda y.x))[d/x] = d\ (\lambda x.\lambda y.x)$$

   (Notice that $(x\ (\lambda x.\lambda y.x))[d/x] = d\ (\lambda x.\lambda y.d)$ would be wrong!)

2. $e = \lambda y.e'$ is a lambda abstraction binding a distinct variable y that is not in the free variables of d: then we can safely substitute into the body of the abstraction and abstract the result. An example of this case is

   $$(\lambda y.x\ y)[(\lambda a.a)/x] = \lambda y.(\lambda a.a)\ y$$

3. $e = \lambda y.e'$ is a lambda abstraction binding a distinct variable y that is free in d: here is where we have to avoid "capturing" the free variable of d, which we achieve by alpha-converting the lambda abstraction to use

a fresh variable z (the notation $e'\{z/y\}$ means "swap z for y in e'"). An example of this case is

$$(\lambda y.x\ y)[(\lambda a.y\ a)/x] = \lambda z.(\lambda a.y\ a)\ z$$

(Notice that $(\lambda y.x\ y)[(\lambda a.y\ a)/x] = \lambda y.(\lambda a.y\ a)\ y$ would be wrong!)

3.2 Beta normalization

Now that we have a precise definition of capture-avoiding substitution, we can write the rule of beta reduction

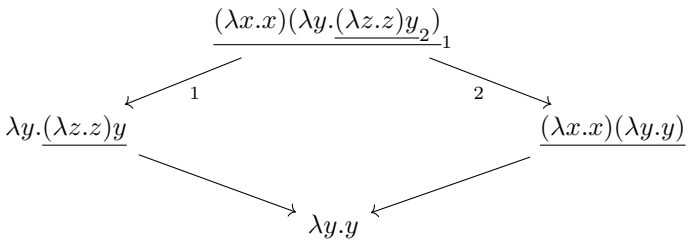
$$(\lambda x.e)\, d \to e[d/x]$$

without ambiguity. This is the *only* rule of computation in pure lambda calculus, and we are allowed to apply it anywhere inside an expression, that is, to any subexpression. A subexpression of the form $(\lambda x.e)\, d$ is called a β-*redex* and an expression with no β-redices is said to be β-*normal*. A priori, it may seem like repeatedly choosing different subexpressions to reduce could lead to different final results, but the Church-Rosser theorem – also called the *confluence* property – guarantees that this is not the case. In the statement of the theorem below, we write $\to^*$ for the reflexive-transitive closure of the β-reduction relation – that is, $e \to^* e'$ if e can be transformed into e' by applying any number of β-reduction steps (including the case of zero steps, with $e = e'$).

Theorem (Church and Rosser, 1936)**.** Let e, e_1, e_2 be lambda expressions such that $e \to^* e_1$ and $e \to^* e_2$. Then there exists a lambda expression d such that $e_1 \to^* d$ and $e_2 \to^* d$.

For a proof of the Church-Rosser theorem, see Selinger's notes.

As an illustration of the confluence property in action, consider the two different ways of reducing the expression $(\lambda x.x)(\lambda y.(\lambda z.z)y)$ (we have underlined the beta redices):



Both reduction paths end up at the same normal form. A corollary of the Church-Rosser theorem is that every lambda expression has *at most one* normal form (i.e., if e reduces to both $e \to^* d_1$ and $e \to^* d_2$ with both d_1 and d_2 β-normal, then $d_1 = d_2$). However, not every lambda expression has a normal form. The simplest example is the expression $\omega = (\lambda x.x\, x)(\lambda x.x\, x)$, which reduces to itself.

In general, it cannot be determined in advance whether a lambda expression has a normal form – in fact, this was the first example of an undecidable problem that appeared in print! It appeared in a 1936 paper by Church, which narrowly pre-dated Turing's famous paper on the Halting Problem.

Exercise 3.1. Reduce the expression $(\lambda x.\lambda y.x\, y\, y)\,(\lambda a.\lambda b.b)\,(\lambda z.z)$ to β-normal form, indicating every β-reduction step.

3.3 Representing data types

Pure lambda calculus has first-class functions, but nothing else. No booleans, no integers, no strings. Just functions. At first, it is not clear how any programming of interest could be accomplished in such an impoverished language, but we will see that it can be.

A key point is that athough pure lambda calculus does not have any built-in types nor any explicit mechanism (like Haskell's **data** keyword) for introducing new types, data types such as booleans and natural numbers may be naturally *represented* by lambda expressions in a way that allows most operations that one cares about to be performed easily. The idea again goes back to Church and is now referred to as *Church encoding*. For example, the booleans can be represented by the following two lambda expressions:

$$\mathit{True} \stackrel{\text{def}}{=} \lambda xy.x \quad \mathit{False} \stackrel{\text{def}}{=} \lambda xy.y$$

Intuitively, we encode a boolean as a function that selects between two options. This representation makes it easy to define conditional expressions in terms of function application:

$$\textbf{if } b \textbf{ then } e \textbf{ else } e' \stackrel{\text{def}}{=} b\ e\ e'$$

We can also easily define boolean negation, conjunction, disjunction:

$$\mathit{not} \stackrel{\text{def}}{=} \lambda bxy.b\ y\ x \quad \mathit{and} \stackrel{\text{def}}{=} \lambda bc.b\ c\ \mathit{False} \quad \mathit{or} \stackrel{\text{def}}{=} \lambda bc.b\ \mathit{True}\ c$$

Exercise 3.2. By performing β-reductions, verify that the definitions of the boolean operators satisfy the expected truth tables:

$$\begin{aligned} \mathit{not}\ \mathit{False} &\to^* \mathit{True} & \mathit{not}\ \mathit{True} &\to^* \mathit{False} \\ \mathit{and}\ \mathit{False}\ c &\to^* \mathit{False} & \mathit{and}\ \mathit{True}\ c &\to^* c \\ \mathit{or}\ \mathit{False}\ c &\to^* c & \mathit{or}\ \mathit{True}\ c &\to^* \mathit{True} \end{aligned}$$

Natural numbers can be represented as follows (for clarity, we write $\ulcorner n \urcorner$ to distinguish a number n from its encoding as a lambda expression):

$$\ulcorner 0 \urcorner \stackrel{\text{def}}{=} \lambda f.\lambda x.x \quad \ulcorner 1 \urcorner \stackrel{\text{def}}{=} \lambda f.\lambda x.f(x) \quad \ulcorner 2 \urcorner \stackrel{\text{def}}{=} \lambda f.\lambda x.f(f(x)) \quad \ldots$$

The idea is that the natural number n is represented by the higher-order function sending any function f to its n-fold composition $f^{(n)}$. Under this encoding, we can easily write the successor function, addition, and multiplication as lambda expressions:

$$\begin{aligned} \mathit{succ} &\stackrel{\text{def}}{=} \lambda n.\lambda f.\lambda x.f(n\ f\ x) \\ \mathit{add} &\stackrel{\text{def}}{=} \lambda mnfx.m\ f\ (n\ f\ x) \\ \mathit{mult} &\stackrel{\text{def}}{=} \lambda mnfx.m\ (n\ f)\ x \end{aligned}$$

Exercise 3.3. Verify that $\mathit{mult}\ \ulcorner 2 \urcorner\ \ulcorner 3 \urcorner \to^* \ulcorner 6 \urcorner$.

Exercise 3.4. Define a lambda expression *exp* such that for all $m, n \in \mathbb{N}$, $exp\ \ulcorner m \urcorner\ \ulcorner n \urcorner \to^* \ulcorner m^n \urcorner$.

Exercise 3.5. Define a lambda expression *isZero* such that $isZero\ \ulcorner n \urcorner \to^* True$ if $n = 0$ and $isZero\ \ulcorner n \urcorner \to^* False$ if $n > 0$. Similarly define a lambda expression *isEven* such that $isEven\ \ulcorner n \urcorner \to^* True$ if $n \equiv 0 \pmod 2$ and otherwise $isEven\ \ulcorner n \urcorner \to^* False$.

Other data types such as lists and pairs admit similar encodings:

$$[a_1, \ldots, a_n] \stackrel{\text{def}}{=} \lambda f.\lambda x.f\ a_1\ (f\ a_2\ (\ldots (f\ a_n\ x))) \qquad (u, v) \stackrel{\text{def}}{=} \lambda f.f\ u\ v$$

and so on and so forth. It is worth trying some of the exercises above and below to get a sense of the power of such encodings. Still, it is probably not immediately obvious to you that it is really possible to write *all* functions that one could ever want to write using just pure lambda expressions. Indeed, in the early days of lambda calculus, after Alonzo Church had already introduced his encoding of natural numbers, it took a while to confirm that it was even possible to write the predecessor function! (A trick for this was figured out by Church's student Stephen Kleene, reportedly while at the dentist: see §4.1.)

Exercise 3.6. Define a lambda expression *swap* which exchanges the components of a pair in the sense that $swap\ (u, v) \to^* (v, u)$ for all u, v, using the representation of pairs defined above.

Exercise 3.7. Define a lambda expression *append* such that for any pair of lists $[a_1, \ldots, a_n]$ and $[b_1, \ldots, b_m]$ represented as above,

$$append\ [a_1, \ldots, a_n]\ [b_1, \ldots, b_m] \to^* [a_1, \ldots, a_n, b_1, \ldots, b_m].$$

Hint: First recall how we expressed the concatenation operation ($+\!\!+$) using *foldr* in Lecture 2.

Exercise 3.8 (hard!)**.** Define a lambda expression *pred* such that $pred\ \ulcorner 0 \urcorner \to^* \ulcorner 0 \urcorner$ and $pred\ \ulcorner n + 1 \urcorner \to^* \ulcorner n \urcorner$ for all $n \in \mathbb{N}$. *Hint:* You do *not* need to use a fixed point operator in the sense discussed below.

3.4 Fixed point operators

As a functional programming language, something that may seem conspicuously absent from pure lambda calculus is the lack of recursive definition. What if we want to write a function using recursion, such as say the standard recursive definition of the factorial function? The key to representing such recursive definitions in pure lambda calculus is to use what is called a *fixed point operator*. An example of a fixed point operator is

$$\Theta \stackrel{\text{def}}{=} (\lambda xy.y\ (x\ x\ y))\ (\lambda xy.y\ (x\ x\ y))$$

The blur of symbols may look intimidating, but the key property of Θ, which is easy to verify by calculation, is that

$$\Theta\ e \to^* e\ (\Theta\ e)$$

for any lambda expression e. If we read the arrow as an equality sign, Θ may be interpreted as producing a "fixed point" $\Theta\, e$ for any expression e seen as a function. (Recall that a fixed point of a function f is an element x such that $f(x) = x$.) The above fixed point operator was actually found by Alan Turing, who used it to prove the equivalence between the expressive power of untyped lambda calculus and his Turing machines for representing functions on natural numbers.[3]

Fixed point operators allow us to easily translate recursive definitions to pure lambda expressions. For example, the factorial function can be written like so:

$$\mathit{fact} \stackrel{\text{def}}{=} \Theta(\lambda f n.\textbf{if } \mathit{isZero}\ n\ \textbf{then}\ 1\ \textbf{else}\ \mathit{mult}\ n\ (f\ (\mathit{pred}\ n)))$$

They also of course make it even easier to write non-terminating programs. For example, the non-normalizing expression $\omega = (\lambda x.x\,x)(\lambda x.x\,x)$ can be recovered as $\Theta\,(\lambda x.x) \twoheadrightarrow^* \omega$.

4 Some asides

4.1 On defining predecessor

From Henk Barendregt's essay, "Gems of Corrado Böhm":

> At first neither Church nor his students could find a way to lambda define the predecessor function. At the dentist's office Kleene did see how to simulate recursion by iteration and could in that way construct a lambda term defining the predecessor function, [Cro75]. (I believe Kleene told me it was under the influence of laughing gas, N2O, used as anesthetic.) When Church saw that result he stated "Then all intuitively computable functions must be lambda definable." That was the first formulation of Church's thesis and the functional model of computation was born.

4.2 On Alonzo Church as a teacher

From *Indiscrete Thoughts* by Gian-Carlo Rota:

> He spoke slowly in complete paragraphs which seemed to have been read out of a book, evenly and slowly enunciated, as by a talking machine. When interrupted, he would pause for an uncomfortably long period to recover the thread of the argument. He never made casual remarks: they did not belong in the baggage of formal logic. For example, he would not say: "It is raining." Such a statement, taken in isolation, makes no sense. (Whether it is actually raining or not does not matter; what matters is consistency.) He would say instead: "I must postpone my departure for Nassau Street, inasmuch as it is raining, a fact which I can verify by looking out the window." (These were not his exact words.) [...]

[3] Another famous fixed point operator is $\mathsf{Y} = \lambda f.(\lambda x.f\,(x\,x))\,(\lambda x.f\,(x\,x))$, due to Curry.

> It may be asked why anyone would bother to sit in a lecture which was the literal repetition of an available text. Such a question would betray an oversimplified view of what goes on in a classroom. What one really learns in class is what one does not know at the time one is learning. The person lecturing to us was logic incarnate. His pauses, hesitations, emphases, his betrayals of emotion (however rare), and sundry other nonverbal phenomena taught us a lot more logic than any written text could. We learned to think in unison with him as he spoke, as if following the demonstration of a calisthenics instructor. Church's course permanently improved the rigor of our reasoning.

5 Simple typing

As mentioned, Church's original motivation for introducing lambda calculus came from logic, but untyped lambda calculus turned out to be too powerful to serve as a consistent foundation for mathematical reasoning. Nevertheless, typed versions of lambda calculus are used in most modern proof assistants, and also serve as the theoretical foundation for the type systems of languages like Haskell and OCaml.

5.1 Typing rules

The basic mechanism of simple typing is easy to describe. A simple type A is either an *atomic type* $A = \iota, o, \ldots$ (here ι, iota, and o, omicron, are just arbitrary Greek letters) or a *function type* $A = B \to C$ where B, C are simple types. In Church's original 1940 formulation of the simply typed lambda calculus, every variable was annotated with a type, and expressions were built up in a way that ensures that every well-typed expression has a unique type, using the following rules (compare with the rules for building up untyped terms in §3.1):

(variables) $x^A, y^B, z^C, \ldots$ are expressions of types $A, B, C, \ldots$;

(application) if e is an expression of type $A \to B$ and e' is an expression of type A then $e\,e'$ is an expression of type B;

(abstraction) if e is an expression of type B and x^A is a variable then $\lambda x^A.e$ is an expression of type $A \to B$.

For example, both

$$\lambda x^\iota.\lambda y^\iota.x^\iota \quad \text{and} \quad \lambda x^\iota.\lambda y^\iota.y^\iota$$

are expressions of type $\iota \to \iota \to \iota$, and

$$\lambda f^{\iota\to\iota\to o}.\lambda x^\iota.\lambda y^\iota.f^{\iota\to\iota\to o}\,y^\iota\,x^\iota$$

is an expression of type $(\iota \to \iota \to o) \to (\iota \to \iota \to o)$.

More modern formulations of typing do not annotate variables but rather consider expressions within a *typing context* for their free variables. Given an expression e with free variables $\mathrm{free}(e) = x_1, \ldots, x_n$, and a typing context $\Gamma = x_1 : A_1, \ldots, x_n : A_n$, the judgment

$$\Gamma \vdash e : B$$

expresses that e has type B assuming $x_1, \ldots, x_n$ have the types asserted in Γ. The typing judgment is defined inductively through typing rules:

$$\frac{x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\Gamma \vdash e : A \to B \quad \Gamma \vdash e' : A}{\Gamma \vdash e\, e' : B} \qquad \frac{\Gamma, x : A \vdash e : B}{\Gamma \vdash \lambda x.e : A \to B}$$

Here is an example typing derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash f : \iota \to \iota \to o} \quad \dfrac{}{\Gamma \vdash y : \iota}}{\Gamma \vdash f\, y : \iota \to o} \quad \dfrac{}{\Gamma \vdash x : \iota} \quad (\text{where } \Gamma = f : \iota \to \iota \to o, x : \iota, y : \iota)}{f : \iota \to \iota \to o, x : \iota, y : \iota \vdash f\, y\, x : o}}{f : \iota \to \iota \to o, x : \iota \vdash \lambda y.f\, y\, x : \iota \to o}}{f : \iota \to \iota \to o \vdash \lambda x.\lambda y.f\, y\, x : \iota \to \iota \to o}}{\vdash \lambda f.\lambda x.\lambda y.f\, y\, x : (\iota \to \iota \to o) \to (\iota \to \iota \to o)}$$

Notice that in order to derive a typing judgment about a closed lambda expression $(\lambda f.\lambda x.\lambda y.f\, y\, x)$ we need to derive judgments about subexpressions with free variables.

5.2 Strong normalization and type preservation

An expression e is said to be *weakly normalizing* if it has some finite reduction sequence $e \to e_1 \to \ldots \to e_n$ to a normal form e_n. It is said to be *strongly normalizing* if there are no infinite reduction sequences starting at e. As we saw above, $(\lambda x.x)(\lambda y.(\lambda z.z)y)$ is an example of a strongly normalizing expression, whereas ω is an example of an expression that is not even weakly normalizing. An example of an expression that is weakly normalizing but not strongly normalizing is $(\lambda x.\lambda y.y)\,\omega$, since it reduces to $\lambda y.y$ in one step if we apply the outermost β-redex, but also reduces to itself if we decide to reduce the subexpression ω. An important property of the simply-typed lambda calculus is that *every* well-typed expression is strongly normalizing.

Theorem. Every simply-typed lambda expression is strongly normalizing.

A corollary is that non-normalizing expressions such as ω (and fixed point operators such as Θ) cannot be typed! For a proof of the strong normalization theorem, see Chapter 6 of Girard (1989).

Another important property of typing is that it is preserved by reduction in the following sense:

(type preservation) If $\Gamma \vdash e : A$ and $e \to e'$ then $\Gamma \vdash e' : A$.

Combining type preservation, strong normalization, and the Church-Rosser theorem, we can conclude that every simply-typed expression reduces to a unique normal form of the same type. A corollary is that simply-typed lambda calculus is *logically consistent* as a deductive system, in the sense that there are types that cannot be derived by any closed expression – although to see this it is helpful to first refine the typing rules in a way that we now explain.

5.3 Bidirectional type checking

The typing rules of simply-typed lambda calculus are "declarative" meaning that to show that an expression is well-typed it suffices build a derivation tree, but the rules don't give you a direct indication of how to construct one. In practice, it is desirable to have an algorithm that determines whether or not an expression is well-typed in any given context. Haskell and OCaml use an algorithm known as Hindley-Milner type inference, discussed in §6, which also accounts for a limited form of polymorphism. Here we discuss another approach known as *bidirectional type checking* that is very easy to describe and implement and has been gaining increasing traction (Dunfield and Krishnaswami 2022).

To define a bidirectional type system for simply-typed lambda calculus, we begin by refining the typing judgment $\Gamma \vdash e : A$ into a pair of judgments:

- a *checking* judgment $\Gamma \vdash e \Leftarrow A$, which given a typing context Γ, a lambda expression e, and a type A, determines whether e has the type A in the typing context Γ;
- a *synthesis* judgment $\Gamma \vdash e \Rightarrow A$, which given a typing context Γ and a lambda expression e, computes the unique type A of e in the typing context Γ, if it exists (i.e., if e is typable in Γ).

Notice that in the checking judgment $\Gamma \vdash e \Leftarrow A$, all three of the components Γ, e, and A are inputs, whereas in the synthesis judgment $\Gamma \vdash e \Rightarrow A$, the components Γ and e are inputs and A is an output.

Next we refine the typing rules of simply-typed lambda calculus to formulate them as either checking rules or synthesis rules:

$$\frac{x : A \in \Gamma}{\Gamma \vdash x \Rightarrow A} \qquad \frac{\Gamma \vdash e \Rightarrow A \to B \quad \Gamma \vdash e' \Leftarrow A}{\Gamma \vdash e\ e' \Rightarrow B} \qquad \frac{\Gamma, x : A \vdash e \Leftarrow B}{\Gamma \vdash \lambda x.e \Leftarrow A \to B}$$

Here is an algorithmic reading of these rules:

- if the binding $x : A$ is in the context Γ, then the variable x synthesizes type A in Γ;
- if e synthesizes type $A \to B$ in context Γ, and e' checks against type A in the same context, then the application $e\ e'$ synthesizes type B in the same context;
- the abstraction $\lambda x.e$ checks against type $A \to B$ in context Γ if e checks against type B in the context extended with the binding $x : A$.

Finally, we also need to add rules for going back and forth between the two typing modes. Here there is a bit of asymmetry: it is always possible to reduce checking to synthesis, but the other way around requires a type annotation.

$$\frac{\Gamma \vdash e \Rightarrow A \quad A = B}{\Gamma \vdash e \Leftarrow B} \qquad \frac{\Gamma \vdash e \Leftarrow A}{\Gamma \vdash (e : A) \Rightarrow A}$$

Again we can read these rules algorithmically:

- to check e against a type B (in a given context), we can try to synthesize a type A for e (in the same context) and check that $A = B$;

- we can synthesis the type A for an annotated term $(e : A)$, provided that e checks against A.

The collection of these five rules can be easily implemented as a bidirectional type checking algorithm for simply-typed lambda calculus – you should try it! Bidirectional type checking is sound and complete in the following sense. If e is a lambda expression potentially containing type annotations, we write $|e|$ for the expression with annotations erased.

Theorem (Soundness)**.** If $\Gamma \vdash e \Leftarrow A$ or $\Gamma \vdash e \Rightarrow A$ then $\Gamma \vdash |e| : A$.

Theorem (Completeness)**.** If $\Gamma \vdash e : A$ then there exists some type-annotated e' such that $e = |e'|$ and $\Gamma \vdash e' \Leftarrow A$.

One elegant property of the typing rules is that type annotations are only ever required at β-redices. In particular, β-normal expressions can be type-checked algorithmically without requiring any annotations on their subexpressions. This lets us justify the claim made above, that simply-typed lambda calculus is logically consistent in the sense that not every type is inhabited.

Proposition. There is no closed lambda expression e such that $\vdash e : o$.

Proof. Suppose otherwise, by strong normalization and type preservation we know that e reduces to some β-normal expression $e \to^* d$ such that $\vdash d : o$, and by completeness of bidirectional typing we have that $\vdash d \Leftarrow o$. But by inspection of the typing rules this leads to a contradiction. Indeed, since o is an atomic type rather than a function type, the only rule we could potentially apply to derive $\vdash d : o$ is the switch to checking from synthesis,

$$\frac{\vdash d \Rightarrow o}{\vdash d \Leftarrow o}$$

but it is impossible to synthesize a type for an unannotated expression in an empty context. □

6 Type inference and polymorphism

See Chapters 8 and 9 of (Selinger 2013)! (More notes may be added here later.)

References

Barendregt, H. P. (1984). *The Lambda Calculus: Its Syntax and Semantics*. Vol. 103. Studies in Logic and the Foundations of Mathematics. Elsevier.

Barendregt, Henk, Wil Dekkers, and Richard Statman (2010). *Lambda Calculus With Types*. Perspectives in Logic. Cambridge University Press.

Cardone, Felice and J. R. Hindley (2006). "History of Lambda-Calculus and Combinatory Logic". In *Handbook of the History of Logic*. Ed. by D. M. Gabbay and J. Woods. Vol. 5. Elsevier. URL: `https://hope.simons-rock.edu/~pshields/cs/cmpt312/cardone-hindley.pdf`.

Dunfield, Jana and Neel Krishnaswami (2022). "Bidirectional Typing". *ACM Comput. Surv.* 54:5, 98:1–98:38. DOI: `10.1145/3450952`. URL: `https://doi.org/10.1145/3450952`.

Girard, Jean-Yves (1989). *Proofs and Types*. Translated and with appendices by Paul Taylor and Yves Lafont. Cambridge University Press.
Selinger, Peter (2013). "Lecture notes on the lambda calculus". *CoRR* abs/0804.3434: arXiv: 0804.3434. URL: http://arxiv.org/abs/0804.3434.