# Simple Types

CAS CS 320: Concepts of Programming Languages

October 29, 2026

Type theory is ultimately about restriction. We want to "draw a circle" around the well-behaved programs, and we want to do this *locally* and *syntactically*. The "local" part means we don't delineate well-behavedness by analyzing programs on the whole, but rather by restricting their constituent parts, e.g., when it's reasonable to apply a function, and what I should expect after I apply it. The "syntactic" part means we want the restriction mechanism to be part of the stuff that the programmer can read and manipulate.

**An aside on paradoxes.** Type theory, like many things in PL, has its origins in the foundations of mathematics, particularly set theory. In the late 19th/early 20th century, we were pretty cavalier about set construction. We had the wide-eyed notion that we could construct sets via ***unrestricted comprehension***: given any property, we could collect together the things satisfying that property. For example, I could construct the set of red things, or in set-builder notation:

$$\mathsf{Red} = \{X : X \text{ is red}\}$$

where Red includes some apples, the BU logo, and a certain big dog.

This works up to a point, but the cracks start to show once we start thinking about properties that can refer to *the set we're trying to construct.* There's no issue with Red since Red itself is not red[1] (i.e., $\mathsf{Red} \notin \mathsf{Red}$) but the philosopher-mathematician Bertrand Russell at the turn of the 20th century considered the following set:

$$R = \{X : X \notin X\}$$

That is, $R$ is the set of sets *that don't contain themselves.* It may be that $\mathsf{Red} \notin \mathsf{Red}$ but is it the case that $R \in R$? If $R \in R$, then by the properties of $R$ it must be that $R$ it doesn't contain itself, so $R \notin R$. But if $R \notin R$, i.e., it doesn't contain itself, then it should appear in $R$, so $R \in R$. (Whoops.) This is what's now known as ***Russell's paradox*** and it was incredibly disruptive in the development of modern mathematics (paradoxes are nasty business foundationally speaking).

*What does this have to do with type theory?* Russell developed a doctrine of types in order to resolve the paradox. As a massive oversimplification, we can think of a set as a *function* of type `'a -> bool`, which "picks out" the

[1] Or is it?

elements that are included in the set. This is essentially what's called *restricted* set comprehension, i.e., a set is defined by a property that picks out which elements *of a given universe* are included. We can then think of the membership predicate "$x \in S$" as the function `fun x -> fun s -> s x` of type `'a -> ('a -> bool) -> bool` (i.e., given an element `x` and a set `s`, ask `s` if it contains `x`). The problem with the Russell set is that it uses the membership predicate on two things "of the same complexity." When we scaffold our sets with types in this way, we can ensure that we only ask whether $s \in S$ when it's possible for $s$ to be in $S$ in the first place. In particular, the defining property of the Russell set isn't well-typed! Here's an OCaml interaction which demonstrates this:

```
# type 'a set = 'a -> bool;;
type 'a set = 'a -> bool
# let mem (x : 'a) (s : 'a set) : bool = s x;;
val mem : 'a -> 'a set -> bool = <fun>
# let russell = fun x -> not (mem x x);;
                                       ^
Error: This expression has type 'a but an expression was
expected of type
         'a set = 'a -> bool
       The type variable 'a occurs inside 'a set
```

Types restrict us from even being able to state the Russell property, let alone construct the Russell set.

**Type systems.** Types are syntactic objects that *describe* (and sometimes annotate) programs. They're usually described by a BNF grammar, alongside (and sometimes intermingled with) the abstract syntax of the PL in which they're used. A ***typing judgment*** is a judgment which expresses that a program has a particular type, relative to requisite ambient information. The most common shape of a typing judgment is

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

where $e$ is an expression (often called the **subject**), $\tau$ is a type (often called the **predicate**), and $\Gamma$ is a ***context***, which describes the types of possibly free variables in $e$. A ***type system*** is an inference system over typing judgments. We say that an expression $e$ is ***well-typed*** in $\Gamma$ according to a type system if $\Gamma \vdash e : \tau$ is derivable for some type $\tau$.

In reality typing judgments take a whole bunch of forms—later in the course we'll extend judgments to include *constraint* information for type inference. It would be difficult (and not terribly enlightening) to give a characterization of typing judgments in general. A well-specified PL should formally define its type system and its underlying typing judgments.

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau}\ \textsc{var} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}\ \textsc{fun}$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 e_2 : \tau}\ \textsc{app}$$

Figure 1: Type system for the simply-typed $\lambda$-calculus

**Simply-typed $\lambda$-calculus.** The abstract syntax of STLC expressions is the same as that of the untyped $\lambda$-calculus:

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

But in addition to defining the syntax of expressions, we also have to give syntax of types:

$$\tau \ ::= \ \bot \mid \tau \to \tau$$

The first alternative in this grammar is a ***base type*** and is necessary for the grammar to be nontrivial. The second alternative describes ***function types***, also called ***arrow types.*** These will ultimately be used to describe (you guessed it) functions, where the type $A \to B$ is given to a function with input type $A$ and output type $B$. We take arrows to be right associative, just as we do in OCaml. For example, $(\bot \to \bot) \to \bot \to \bot \to \bot$ is shorthand for $(\bot \to \bot) \to (\bot \to (\bot \to \bot))$. STLC typing judgments are of the form

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

where $e$ (the subject) is an expression, $\tau$ (the predicate) is a type, and $\Gamma$ is a ***context***. Contexts are sets of ***variable declarations***, and variable declarations are of the form "$x : \tau$" where $x$ is a variable and $\tau$ is a type. So, for example,

$$\{x : \bot \to \bot, y : \bot\} \vdash \lambda z.z(xy) : (\bot \to \bot) \to \bot$$

is an STLC typing judgment. We read the judgment "$\Gamma \vdash e : \tau$" as "given that every declaration $(x : \tau) \in \Gamma$ holds, it follows that $e$ has type $\tau$."

The type system for STLC is given in Figure 1. The VAR rule just expresses that variables can be given whatever type the context says they are. The FUN rule says that a function has the type $\tau_1 \to \tau_2$ if its body has type $\tau_2$ *relative to* the assumption that the parameter has type $\tau_1$. The APP rule is the most important; it says that *a function can only be applied to an expression whose type is the same as its input type. Furthermore, the type of the result is the output type of the function.* This rule is what gives us control over how functions get used.

**Exercise 1.** *Give a typing derivation for each of the following typing judgments.*

$$\frac{(x:\tau) \in \Gamma}{\Gamma \vdash x : \tau}\ \text{VAR} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x^{\tau_1}.e : \tau_1 \to \tau_2}\ \text{FUN}$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 e_2 : \tau}\ \text{APP}$$

Figure 2: Type system for the simply-typed $\lambda$-calculus with type annotations

1. $\{x : \bot \to \bot, y : \bot\} \vdash \lambda z.z(xy) : (\bot \to \bot) \to \bot$
2. $\varnothing \vdash \lambda f.\lambda x.f(fx) : (\bot \to \bot) \to \bot \to \bot$
3. $\{x : \bot \to \bot \to \bot\} \vdash \lambda y.\lambda z.xz(yz) : (\bot \to \bot) \to \bot \to \bot$

**Exercise 2.** *For each of the following expressions $e$, determine a minimal context $\Gamma$ and a type $\tau$ such that $\Gamma \vdash e : \tau$. Also provide a derivation.*

1. $\lambda x.yx$
2. $\lambda x.\lambda y.x$
3. $(\lambda x.x)(\lambda y.y)z$

**Exercise 3.** *Attempt to give a typing derivation for $\lambda x.xx$. Describe why there is no type $\tau$ such that $\varnothing \vdash \lambda x.xx : \tau$.*

**Type annotations.** There's a somewhat unsatisfying (though not entirely undesirable) feature of STLC as presented: it doesn't satisfy ***type uniqueness.*** We say that a type system satisfies type uniqueness if $\Gamma \vdash e : \tau_1$ and $\Gamma \vdash e : \tau_2$ implies $\tau_1 = \tau_2$.

**Exercise 4.** *Demonstrate that the STLC according to the rules in Figure 1 doesn't satisfy type uniqueness.*

The issue is that bound variables aren't "pinned down," so occasionally we have a degree of freedom in the choice of typing. One fix is to annotate binder variables with types:

$$\begin{aligned} e &::= \lambda x^{\tau}.e \mid ee \mid x \\ \tau &::= \bot \mid \tau \to \tau \end{aligned}$$

What's interesting here is that types now play a role in the syntax of the PL itself. The typing rules change slightly (Figure 2) to ensure that the input type of a function matches its type annotation. The rules in Figure 1 are for the **domain-free** version of STLC. Moving forward, STLC refers to the system described in Figure 2, with type annotations.

**Exercise 5.** *For each of the following expressions $e$, determine a minimal context $\Gamma$ and a type $\tau$ such that $\Gamma \vdash e : \tau$. Also provide a derivation.*

1. $\lambda x^{\bot}.\lambda y^{\bot \to \bot}.yx$
2. $\lambda x^{\bot \to \bot}.x(yx)$
3. $(\lambda x^{(\bot \to \bot) \to \bot}.x(\lambda z^{\bot}.x(wz)))y$

**Exercise 6.** ***(Challenge)*** *Prove that STLC according to the rules in Figure 2 satisfies type uniqueness. Hint: Use induction over the structure of the subject.*

**Meta-theory.** Type uniqueness is an example of a *meta-theoretic* property. The "meta" part here means it's not something derivable *in* STLC, but provable *about* STLC. Much of type theory is the study of meta-theoretic properties of type systems; in a sense this is the point: we delineate classes of programs via types, and then prove that those classes of programs are "nice." We'll deal with a number of meta-theoretic properties, especially as we move towards the notion of type safety, but just to give another example, we can look at ***well-scopedness***. In rough terms, an expression is well-scoped if all of its free variables are declared. We can state this formally as follows.

**Theorem 1.** *If $\Gamma \vdash e : \tau$ and $x \in \mathsf{FV}(e)$, then there is a type $\tau'$ such that $(x : \tau') \in \Gamma$.*

In other words, if $e$ is well-typed, then all of its free variables are declared in a context in which $e$ is well-typed. We prove this by *induction over derivations.*

*Proof.* Suppose that $\Gamma \vdash e : \tau$ is derivable. We consider all the possible cases of the last rule in the given derivation.

*Case 1.* Suppose the last rule applied is the VAR rule. Then the derivation looks like:

$$\frac{}{\Gamma \vdash x : \tau}\ \textsc{var}$$

In particular, the whole derivation is a single axiom, and the side condition of the VAR rule ensures that $(x : \tau) \in \Gamma$.

*Case 2.* Suppose the last rule applied is the FUN rule. Then the derivation looks like:

$$\frac{\begin{array}{c}\vdots \\ \Gamma, x : \tau_1 \vdash e' : \tau_2\end{array}}{\Gamma \vdash \lambda x^{\tau_1}.e' : \tau_1 \to \tau_2}\ \textsc{fun}$$

By definition, $\mathsf{FV}(\lambda x^{\tau_1}.e') = \mathsf{FV}(e') \setminus \{x\}$. In other words, $y \in \mathsf{FV}(\lambda x^{\tau_1}.e')$ implies $y \in \mathsf{FV}(e')$ and $y \neq x$. By the inductive hypothesis, we know that every free variable of $e'$ is declared in $\Gamma, x : \tau_1$. Since $y \neq x$, it must be that $y$ is declared in $\Gamma$.

*Case 3.* Suppose the last rule applied is the APP rule. Then the derivation looks like:

$$\frac{\begin{array}{cc}\vdots & \vdots \\ \Gamma \vdash e_1 : \tau_2 \to \tau & \Gamma \vdash e_2 : \tau_2\end{array}}{\Gamma \vdash e_1 e_2 : \tau}\ \text{APP}$$

By definition, $\mathsf{FV}(e_1 e_2) = \mathsf{FV}(e_1) \cup \mathsf{FV}(e_2)$. In other words, if $y$ is free in $e_1 e_2$, then it's free in $e_1$ or $e_2$. If $y$ is free in $e_1$, then by the induction hypothesis applied to the left premise, $y$ is declared in $\Gamma$. If $y$ is free in $e_2$, then by the induction hypothesis applied to the right premise, $y$ is declared in $\Gamma$. In either case, $y$ is declared in $\Gamma$. □

To some degree, this shouldn't be surprising. The context exists to keep track of the free variables that appear in a given expression. But this isn't something that we get "for free"; we have to prove it as we did above to appeal to this fact.

**Exercise 7.** *Demonstrate that the converse does not hold, i.e., it's not the case that if* $\Gamma \vdash e : \tau$ *and* $x$ *is declared in* $\Gamma$ *then* $x \in \mathsf{FV}(e)$.

**Why types?** The major benefit of working in STLC over the untyped $\lambda$-calculus is that the well-typed expressions are semantically well-behaved. Our next topic is *type safety*, at which point we'll state exactly what "well-behaved" means. That said, in a sense STLC may be *too* well-behaved: STLC is ***strongly normalizing***, i.e., every well-typed expression has a normal form *and any evaluation strategy will evaluate to it.* This implies there's no way to define general recursion in STLC, so it's not Turing complete.