*Type checking* is a lightweight technique for proving simple properties of programs. Unlike theorem-proving techniques based on axiomatic semantics, type checking usually cannot determine if a program will produce the correct output. Instead, it is a way to test whether a program is *well-formed*, with the idea that a well-formed program satisfies certain desirable properties. The traditional definition of *type soundness* is that well-formed programs cannot get stuck—they either go into an infinite loop or terminate with a value. This requirement corresponds to the absence of a wide variety of run-time errors in real languages. This is a weak notion of program correctness, but nevertheless very useful in practice for catching bugs.

Type systems, however, are powerful and extend beyond this basic notion of correctness. In the past few decades, researchers have figured out how type systems can verify properties ranging from safe concurrency in languages like Rust, to checking maintenance of certain program state, to statically analysis of code performance and some security conditions. We do not have time to cover most of these applications, but we will cover the foundations of type systems starting with a very simple typed language and building from there.

# 1 Simply Typed $\lambda$-Calculus

We begin our exploration into types with the Simply Typed $\lambda$-calculus (STLC). STLC is very similar to the basic $\lambda$-calculus we saw earlier in this course, but we now assign types to $\lambda$-terms according to a set of typing rules. A $\lambda$-term is considered to be well-formed if we can derive a type for it using the typing rules.

## 1.1 Syntax and Semantics

The syntax of STLC is similar to that of untyped $\lambda$-calculus, with a few notable additions. The biggest change is that we now have two inductively defined expressions: *terms* and *types*.

$$\begin{array}{lrcl} \text{Types} & \tau & ::= & \mathsf{unit} \mid \tau_1 \to \tau_2 \\ \text{Terms} & e & ::= & () \mid x \mid e_1\ e_2 \mid \lambda x{:}\tau.\, e \end{array}$$

The definition of terms differs in two ways from the pure $\lambda$-calculus we saw before. First, we now have a primitive unit value ().[1] Second, $\lambda$-abstraction explicitly specifies the type of its argument. That is, $\lambda x{:}\tau.\, e$ is a function that takes one input *of type* $\tau$ and evaluates $e$.

A *type* represents a collection of related values. The definition of types includes two cases: unit, the type of (), and $\tau_1 \to \tau_2$, the type of a function that takes input of type $\tau_1$ and produces output of type $\tau_2$. By convention, the function arrow is right-associative, so $\tau_1 \to \tau_2 \to \tau_3$ is the same as $\tau_1 \to (\tau_2 \to \tau_3)$. This convention matches left-associative function application. If $f$ has type $\tau_1 \to (\tau_2 \to \tau_3)$ and $v_1$ and $v_2$ have types $\tau_1$ and $\tau_2$, respectively, then $f\ v_1 : \tau_2 \to \tau_3$, so $f\ v_1\ v_2 = (f\ v_1)\ v_2 : \tau_3$, as we would hope.

The operational semantics of this language is unchanged from CBV $\lambda$-calculus, counting () as a value which, accordingly, has no reduction rule. That is, we have the following definitions of values, evaluation contexts, and reduction rules.

$$\begin{array}{rcl} v & ::= & () \mid \lambda x{:}\tau.\, e \\ E & ::= & [\cdot] \mid E\ e \mid v\ E \end{array}$$

$$\frac{e \longrightarrow e'}{E[e] \longrightarrow E[e']} \qquad \frac{}{(\lambda x{:}\tau.\, e)\ v \longrightarrow e[x \mapsto v]}$$

[1] Technically this is unnecessary for the language, but it makes explanations simpler and more intuitive. Without it we need an uninhabited base type $A$, which is a bit bizarre to work with.

One other small change is that the presence of () means there are now closed $\lambda$-terms can get stuck. For example, () ($\lambda x : \mathsf{unit}.\, x$) is a perfectly good closed $\lambda$-term that is also stuck. We would like to eliminate these, and the type system will allow us to do so.

## 1.2 Typing Rules

The typing rules for STLC are an inductive relation on three inputs: a typing context $\Gamma$ : **Var** $\rightharpoonup$ **Type** that maps variables to types, an expression $e$, and a type $\tau$. These *typing judgments* are written $\Gamma \vdash e : \tau$ and mean that we can prove that expression $e$ has type $\tau$ in typing context $\Gamma$ using the typing rules. We also write $\vdash e : \tau$ as short-hand for $\varnothing \vdash e : \tau$, meaning we can prove $e$ has type $\tau$ in an empty context.

Here are the typing rules, using $\Gamma, x : \tau$ as an extension operator that means the same as $\Gamma[x \mapsto \tau]$.[2]

$$\text{[UNIT]}\ \frac{}{\Gamma \vdash () : \mathsf{unit}} \qquad \text{[VAR]}\ \frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau} \qquad \text{[ABS]}\ \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x : \tau_1.\, e : \tau_1 \to \tau_2} \qquad \text{[APP]}\ \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \qquad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

Let us examine these rules more closely.

- UNIT says that () has type unit in any environment.
- VAR says that a variable $x$ has whatever type the environment $\Gamma$ maps $x$ to. If $x \notin \mathrm{dom}(\Gamma)$, then this rule cannot apply as $\Gamma(x)$ is undefined, and $x$ does not have a type.
- The function abstraction rule ABS gives types for $\lambda$-abstractions. Since $\lambda x : \tau_1.\, e$ is supposed to be a function that takes an argument of type $\tau_1$, the type of the input to the function should match the annotation $\tau_1$. The type $\tau_2$ the function outputs is the type of whatever the function body $e$ evaluates to.

  However, $e$ has access not only to every variable already bound in $\Gamma$, but also to the freshly-bound $x$. As the input $x$ is assumed to have type $\tau_1$, $e$ has access to the extended environment $\Gamma, x : \tau_1$.
- APP defines the typing rule for function application. The expression $e_1\ e_2$ applies the function represented by $e_1$ to the argument represented by $e_2$. For this to have type $\tau_2$, $e_1$ must have a function type $\tau_1 \to \tau_2$ for some input type $\tau_1$. As $e_2$ is passed as the argument to $e_1$, the type of $e_2$ must match $\tau_1$, the argument type expected by $e_1$.

Every well-typed term in STLC has a proof tree consisting of applications of the typing rules to derive the term. For instance, consider ($\lambda x : \mathsf{unit}.\, \lambda y : (\tau \to \tau).\, x$) () $\mathsf{id}_\tau$ (where $\mathsf{id}_\tau = \lambda x : \tau.\, x$), which evaluates to (). Since $\vdash$ () : unit, we would expect $\vdash$ ($\lambda x : \mathsf{unit}.\, \lambda y : (\tau \to \tau).\, x$) () $\mathsf{id}_\tau$ : unit as well. Here is a proof.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x : \mathsf{unit} \vdash, y : (\tau \to \tau) \vdash x : \mathsf{unit}}\,\text{[VAR]}}{x : \mathsf{unit} \vdash \lambda y : (\tau \to \tau).\, x : (\tau \to \tau) \to \mathsf{unit}}\,\text{[ABS]}}{\vdash \lambda x : \mathsf{unit}.\, \lambda y : (\tau \to \tau).\, x : \mathsf{unit} \to (\tau \to \tau) \to \mathsf{unit}}\,\text{[ABS]} \qquad \dfrac{}{\vdash () : \mathsf{unit}}\,\text{[UNIT]}}{\vdash (\lambda x : \mathsf{unit}.\, \lambda y : (\tau \to \tau).\, x)\ () : (\tau \to \tau) \to \mathsf{unit}}\,\text{[APP]} \qquad \dfrac{\dfrac{}{x : \tau \vdash x : \tau}\,\text{[VAR]}}{\vdash \lambda x : \tau.\, x : \tau \to \tau}\,\text{[ABS]}}{\vdash (\lambda x : \mathsf{unit}.\, \lambda y : (\tau \to \tau).\, x)\ ()\ \mathsf{id}_\tau : \mathsf{unit}}\,\text{[APP]}$$

An automated type checker can effectively construct proof trees like this to test if a program is type-correct.

Note that, in this type system, if a type exists for some $\lambda$-term in a context $\Gamma$, then that type is unique. That is, if $\Gamma \vdash e : \tau$ and $\Gamma \vdash e : \tau'$, then $\tau = \tau'$. One can easily prove this fact by structural induction on $e$ using the fact that at most one typing rule can apply for each syntactic form in the BNF grammar. Note that this uniqueness property is not true for all type systems.

[2] This different syntax is more standard in the literature for two reasons. One, it mirrors the colon-based syntax for "$x$ has type $\tau$." Two, it suggests that $\Gamma$ is a list, which is a common way to implement the contexts in simple interpreters.

### 1.3 An Example

To see how this might work, we can write out the types for some of the encodings we saw previously. For instance, what is the type of the Church numeral for 1? Recall that $\overline{1} \triangleq \lambda f x.\ f\ x$. To give this a type, we need to give $f$ and $x$ types such that $f\ x$ is a well-typed operation. Looking at the App rule, this means $f : \tau_1 \to \tau_2$ and $x : \tau_1$. We therefore could say:

$$\overline{1} \triangleq \lambda f{:}\tau_1 \to \tau_2.\, \lambda x{:}\tau_1.\, f\ x \ \ :\ \ (\tau_1 \to \tau_2) \to \tau_1 \to \tau_2.$$

What about the Church numeral $\overline{2} \triangleq \lambda f x.\ f\ (f\ x)$? In this case, we have that $f : \tau_1 \to \tau_2$ *and that* $f\ x : \tau_1$! However, given the type of $f$, the App rule requires that $f\ x$ has type $\tau_2$. The only way for this to work is if $\tau_1 = \tau_2$. We can therefore simplify to

$$\overline{2} \triangleq \lambda f{:}\tau \to \tau.\, \lambda x{:}\tau.\, f\ x \ \ :\ \ (\tau \to \tau) \to \tau \to \tau.$$

Indeed, for any choice of $\tau$, we can write our Church numerals to have type $(\tau \to \tau) \to \tau \to \tau$, which we will denote more succinctly as $\mathsf{num}_\tau$.

Interestingly, the types have added some restrictions here. While we can choose any $\tau$ and correctly write a Church numeral of type $\mathsf{num}_\tau$, we cannot mix and match. $\mathsf{num}_\tau$ does not necessarily play nicely with $\mathsf{num}_{\tau'}$. Unfortunately, some of our encodings also stop working well. Take add, for example. We would hope the type would be $\mathsf{add}_\tau : \mathsf{num}_\tau \to \mathsf{num}_\tau \to \mathsf{num}_\tau$. However, we defined $\mathsf{add} \triangleq \lambda nm.\, n\ \mathsf{succ}\ m$. If we give $m$ type $\mathsf{num}_\tau$, then succ must have type $\mathsf{num}_\tau \to \mathsf{num}_\tau$, which means $n$ must have type

$$n : (\mathsf{num}_\tau \to \mathsf{num}_\tau) \to (\mathsf{num}_\tau \to \mathsf{num}_\tau) = \mathsf{num}_{\mathsf{num}_\tau}.$$

This is not the type we were hoping for.

It is for this reason that most typed language include things like numbers and booleans directly as language primitives. We will see how to add those and some other programming constructs next time.

## 2 Expressive Power

By this point you might be wondering, was this encoding difficulty with Church numerals merely annoying and we could have worked around it, or have we fundamentally lost some expressive power in the language by introducing types? The answer is, resoundingly, *we have lost expressive power*. The fact that we can no longer apply functions with mismatched types is a big deal.

More importantly, we have actually lost the ability to write loops! Recall the simple infinite loop

$$\Omega \triangleq (\lambda x.\, x\ x)\ (\lambda x.\, x\ x).$$

We can show that there is no way to give this a type by showing that we cannot give a type to $\lambda x{:}\tau.\, x\ x$. If we could, the typing derivation we have to look something like this:

$$\dfrac{\dfrac{\dfrac{\Gamma, x{:}\tau \vdash x : \tau \to \tau' \qquad \dfrac{}{\Gamma, x{:}\tau \vdash x : \tau}\,[\textsc{Var}]}{\Gamma, x{:}\tau \vdash x\ x : \tau'}\,[\textsc{App}]}{\Gamma \vdash \lambda x{:}\tau.\, x\ x : \tau \to \tau'}\,[\textsc{Abs}]}$$

However, types in Stlc are unique in a given context! That means that if $\Gamma, x{:}\tau \vdash x : \tau \to \tau'$ and $\Gamma, x{:}\tau \vdash x : \tau$, then it must be the case that $\tau = \tau \to \tau'$. But our types are defined inductively, which prohibits a type from being a subexpression of itself. There is thus no type with the property that $\tau = \tau \to \tau'$, meaning $\lambda x{:}\tau.\, x\ x$, and consequently also $\Omega$, cannot have a type.

In fact, we will later see that we cannot write down *any* nonterminating program in Stlc. In later lectures we will show how to extend the type system to allow for loops and nontermination.