Type checking is a lightweight technique for proving simple properties of programs. Unlike theorem-proving techniques based on axiomatic semantics like separation logic, 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 code performance and even 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 λ-Calculus

We begin our exploration into types with the Simply Typed λ-calculus (STLC). STLC is very similar to the basic λ-calculus we saw earlier in this course, but we now assign types to λ-terms according to a set of typing rules. A λ-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 λ-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 λ-calculus we saw before. First, we now have a primitive unit value ().[1] Second, λ-abstraction explicitly specifies the type of its argument. That is, $\lambda x{:}\tau.\, e$ is a function that takes one input *of type* τ and evaluates e. Embedding the type of a function argument into the syntax in this way is known as *Church style*—named for Alonzo Church—or *intrinsic types*. There is also a *Curry style* syntax (or *extrinsic types*) without the explicit types, which leads to very similar results.

A *type* represents a collection of related values. Roughly speaking, it represents a set of values that can all be used in the same way within a program. For STLC, 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 τ_1 and produces output of type τ_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 τ_1 and τ_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 λ-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.

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

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

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

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

1.2 Typing System

A type system is a way to give types to terms in a language. We might want to say, for instance, that () : unit or $\lambda x{:}\mathsf{unit}.\, x$: unit $\to$ unit. One might hope that a type system will then be a binary relation on terms and types.

However, using a binary relation runs into a problem with function abstraction. If we have the term $\lambda x{:}\tau.\, e$, we would like to give it the type $\tau \to \tau'$ for some type τ'. To do so, we need to know that when we it an argument of type τ, it will produce something of type τ'. But how can we know that? To type-check a λ-abstraction, we need to verify that the body e will have type τ' *when given a value of type* τ *in place of* x. That is, we need to know what type x will have when checking e!

To do that, we include a *typing context* Γ : **Var** $\rightharpoonup$ **Type** that maps variables to types. A *typing judgment* is then a ternary relation $\Gamma \vdash e : \tau$ meaning that we can prove e has type τ in typing context Γ. We will write $\vdash e : \tau$ as short-hand for $\varnothing \vdash e : \tau$, meaning we can prove e has type τ in an empty context (when e is closed).

In STLC—as in most type systems—this relation is defined inductively. Here are the typing rules, using $\Gamma, x{:}\tau$ as an extension operator that means the same as $\Gamma[x \mapsto \tau]$.[2]

[UNIT] $$\frac{}{\Gamma \vdash () : \mathsf{unit}}$$

[VAR] $$\frac{\Gamma(x) = \tau}{\Gamma \vdash x : \tau}$$

[ABS] $$\frac{\Gamma, x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x{:}\tau_1.\, e : \tau_1 \to \tau_2}$$

[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 Γ 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 λ-abstractions. Since $\lambda x{:}\tau_1.\, e$ is supposed to be a function that takes an argument of type τ_1, the type of the input to the function should match the annotation τ_1. The type τ_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 Γ, but also to the freshly-bound x. As the input x is assumed to have type τ_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 τ_2, e_1 must have a function type $\tau_1 \to \tau_2$ for some input type τ_1. As e_2 is passed as the argument to e_1, the type of e_2 must match τ_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 $(\lambda x{:}\mathsf{unit}.\, \lambda y{:}(\tau \to \tau).\, x)$ () id_τ (where $\mathsf{id}_\tau = \lambda x{:}\tau.\, x$), which evaluates to ().

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

Since $\vdash () : \mathsf{unit}$, we would expect $\vdash (\lambda x{:}\mathsf{unit}.\, \lambda y{:}(\tau \to \tau).\, x)\ ()\ \mathsf{id}_\tau : \mathsf{unit}$ as well. Here is a proof.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{x{:}\mathsf{unit}, y{:}(\tau \to \tau) \vdash x : \mathsf{unit}}\,[\textsc{Var}]}{x{:}\mathsf{unit} \vdash \lambda y{:}(\tau \to \tau).\, x : (\tau \to \tau) \to \mathsf{unit}}\,[\textsc{Abs}]}{\vdash \lambda x{:}\mathsf{unit}.\, \lambda y{:}(\tau \to \tau).\, x : \mathsf{unit} \to (\tau \to \tau) \to \mathsf{unit}}\,[\textsc{Abs}] \qquad \dfrac{}{\vdash () : \mathsf{unit}}\,[\textsc{Unit}]}{\vdash (\lambda x{:}\mathsf{unit}.\, \lambda y{:}(\tau \to \tau).\, x)\ () : (\tau \to \tau) \to \mathsf{unit}}\,[\textsc{App}] \qquad \dfrac{\dfrac{}{x{:}\tau \vdash x : \tau}\,[\textsc{Var}]}{\vdash \lambda x{:}\tau.\, x : \tau \to \tau}\,[\textsc{Abs}]}{\vdash (\lambda x{:}\mathsf{unit}.\, \lambda y{:}(\tau \to \tau).\, x)\ ()\ \mathsf{id}_\tau : \mathsf{unit}}\,[\textsc{App}]$$

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 λ-term in a context Γ, then that type is unique. That is, if $\Gamma \vdash e : \tau$ and $\Gamma \vdash e : \tau'$, then $\tau = \tau'$. This property is not true for all type systems, and in fact we will see one in a later lecture where it is false.

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:

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

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 τ_2. The only way for this to work is if $\tau_1 = \tau_2$. We can therefore simplify to

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

Indeed, for any choice of τ, we can write our Church numerals to have type $(\tau \to \tau) \to \tau \to \tau$, which we will denote more succinctly as num_τ.

Interestingly, the types have added some restrictions here. While we can choose any τ and correctly write a Church numeral of type num_τ, we cannot mix and match. num_τ 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 $\mathsf{add}_\tau : \mathsf{num}_\tau \to \mathsf{num}_\tau \to \mathsf{num}_\tau$. In Lecture 9 we had two different encodings of add:

$$\mathsf{add}^A \triangleq \lambda m n.\, \lambda f x.\, m\ f\ (n\ f\ x)$$
$$\mathsf{add}^B \triangleq \lambda m n.\, m\ \mathsf{succ}\ n$$

With add^A, we can give both m and n type num_τ, then f needs type $\tau \to \tau$ and x gets type τ, and all of the types check out, exactly as we hoped. With add^B, however, despite defining (semantically) the same behavior on Church numerals, things do not work out as well. If we give m type num_τ, 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.

While add^A works and we could use it, we only had one definition for mult, and that runs into the same problem as add^B, as do some of our other encodings. 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 would 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 Ω, 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.