Lecture 4: Type Checking

So far, we've been working with languages that don't have a static type system. Therefore, our programs may have lurking type errors that won't be caught until they are run. A type checker is a decision procedure that proves the absence of type errors. In particular, a type checker can prove the absence of type errors even if the program itself would run forever.

1 Typing Rules

Figure 5.1 presents a typed version of the language we examined in the last lecture. We've elided the semantics for the language, since it is essentially the same. Instead, the figure presents a (slightly modified) syntax and the type system for the language, which we present using the same inference rule notation that we used earlier.

We use the metavariable T to represent types. The types include Int, Bool, and $T_1 \rightarrow T_2$, which is the type of a function that takes arguments of type T_1 and produces results of type T_2. To type-check variables, we need a *type environment* (Γ) which is a partial function from variables to types. We $\Gamma \vdash e : T$ to mean "in the type environment Γ, the expression e has type T", or simply "Γ proves that e has type T".

To type-check constants (T-Const), we assume the existence of a total function ty that maps constants to their types. We type-check identifiers (T-Id) by looking them up in the environment. To type-check binary operations e_1 $\mathsf{e_2}$, we calculate the types of e_1 and e_2 and use the auxiliary function $Delta$, which returns the type of value that op_2 would produce given e_1 and e_2. Note that Δ is a partial function. E.g., if the operation is arithmetic and the expressions have type Bool, then Δ is not defined.

To type-check if-expressions (T-If), we require the conditional to have type Bool and constraint the both branches to have the same type T. This means that expressions such as these are untypable,

$$\mathsf{if\ true\ then}\ 10\ \mathsf{else\ true}$$

even though it is obvious that it always produces a value of type Int.

To type-check functions (T-Fun), we need to change their syntax. The argument of a function is now annotated with its type: $\lambda x : T_1.e$. Given this annotation, we type-check the function body e in a type environment that is augmented with $x : T_1$. If we show that the body has type T_2 in an environment where x has type T_1, then the function has type $T_1 \rightarrow T_2$. To type-check function applications (T-App), we calculate the type of the function and the actual argument, and ensure that the actual argument type is the same as the type of argument that the function expects. If so, the type of an application is the type of the function body.

$\boxed{e}$

Binary Operators

$$
\begin{array}{lcl}
op_2 & ::= & + \mid > \mid \cdots \\
ty & \in & c \to T \\
\Delta & \in & op_2 \times T \times T \rightharpoonup T
\end{array}
$$

Constants

c	$::=$	true	True
	$\mid$	false	False
	$\mid$	n	Integers

Types

T	$::=$	Int	Integer type
	$\mid$	Bool	Boolean type
	$\mid$	$T_1 \to T_2$	Function types

Expressions

e	$::=$	c	Constants
	$\mid$	x	Identifiers
	$\mid$	$op_2(e_1, e_2)$	Bin. Ops.
	$\mid$	$\mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3$	Conditionals
	$\mid$	$e_1\ e_2$	Applications
	$\mid$	$\lambda x : T.e$	Functions

Type Environments

$\Gamma \in x \rightharpoonup T$

(a) Syntax.

$\boxed{\Gamma \vdash e : T}$

$$\textsc{T-Const}\ \Gamma \vdash c : ty(c) \qquad \textsc{T-Id}\ \Gamma \vdash x : \Gamma(x)$$

$$\textsc{T-BinOp}\ \frac{\Gamma \vdash e_1 : T_1 \qquad \Gamma \vdash e_2 : T_2}{\Gamma \vdash op_2(e_1, e_2) : \Delta(op_2, T_1, T_2)}$$

$$\textsc{T-If}\ \frac{\Gamma \vdash e_1 : \mathsf{Bool} \qquad \Gamma \vdash e_2 : T \qquad \Gamma \vdash e_3 : T}{\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : T}$$

$$\textsc{T-Fun}\ \frac{\Gamma[x \mapsto T_1] \vdash e : T_2}{\Gamma \vdash \lambda x : T_1.e : T_1 \to T_2}$$

$$\textsc{T-App}\ \frac{\Gamma \vdash e_1 : T_1 \to T_2 \qquad \Gamma \vdash e_2 : T_1}{\Gamma \vdash e_1\ e_2 : T_2}$$

(b) Type Checking.

Figure 5.1: Simple Types.

Correctness We've seen how to formally define a type system, but we haven't yet discussed what it means for a type system to be correct. Intuitively, a type system is making a prediction about the kind of value that an expression will produce, and a type system is correct if its predictions are always accurate. In the next few lectures, we'll be more precise about this notion of correctness and prove that this type system is correct.