9 *Simply Typed Lambda-Calculus*

This chapter introduces the most elementary member of the family of typed languages that we shall be studying for the rest of the book: the simply typed lambda-calculus of Church (1940) and Curry (1958).

9.1 Function Types

In Chapter 8, we introduced a simple static type system for arithmetic expressions with two types: `Bool`, classifying terms whose evaluation yields a boolean, and `Nat`, classifying terms whose evaluation yields a number. The "ill-typed" terms not belonging to either of these types include all the terms that reach stuck states during evaluation (e.g., `if 0 then 1 else 2`) as well as some terms that actually behave fine during evaluation, but for which our static classification is too conservative (like `if true then 0 else false`).

Suppose we want to construct a similar type system for a language combining booleans (for the sake of brevity, we'll ignore numbers in this chapter) with the primitives of the pure lambda-calculus. That is, we want to introduce typing rules for variables, abstractions, and applications that (a) maintain type safety—i.e., satisfy the type preservation and progress theorems, 8.3.2 and 8.3.3—and (b) are not too conservative—i.e., they should assign types to most of the programs we actually care about writing.

Of course, since the pure lambda-calculus is Turing complete, there is no hope of giving an *exact* type analysis for these primitives. For example, there is no way of reliably determining whether a program like

```
if <long and tricky computation> then true else (λx.x)
```

yields a boolean or a function without actually running the long and tricky computation and seeing whether it yields `true` or `false`. But, in general, the

The system studied in this chapter is the simply typed lambda-calculus (Figure 9-1) with booleans (8-1). The associated OCaml implementation is `fullsimple`.

long and tricky computation might even diverge, and any typechecker that tries to predict its outcome precisely will then diverge as well.

To extend the type system for booleans to include functions, we clearly need to add a type classifying terms whose evaluation results in a function. As a first approximation, let's call this type →. If we add a typing rule

$$\lambda x.t : \rightarrow$$

giving every λ-abstraction the type →, we can classify both simple terms like `λx.x` and compound terms like `if true then (λx.true) else (λx.λy.y)` as yielding functions.

But this rough analysis is clearly too conservative: functions like `λx.true` and `λx.λy.y` are lumped together in the same type →, ignoring the fact that applying the first to `true` yields a boolean, while applying the second to `true` yields another function. In general, in order to give a useful type to the result of an application, we need to know more about the left-hand side than just that it is a function: we need to know what type the function returns. Moreover, in order to be sure that the function will behave correctly when it is called, we need to keep track of what type of arguments it expects. To keep track of this information, we replace the bare type → by an infinite family of types of the form $T_1 \rightarrow T_2$, each classifying functions that expect arguments of type T_1 and return results of type T_2.

9.1.1 Definition: The set of *simple types* over the type `Bool` is generated by the following grammar:

T ::=		*types:*
	Bool	*type of booleans*
	T→T	*type of functions*

The *type constructor* → is right-associative—that is, the expression $T_1 \rightarrow T_2 \rightarrow T_3$ stands for $T_1 \rightarrow (T_2 \rightarrow T_3)$. □

For example `Bool→Bool` is the type of functions mapping boolean arguments to boolean results. `(Bool→Bool)→(Bool→Bool)`—or, equivalently, `(Bool→Bool)→Bool→Bool`—is the type of functions that take boolean-to-boolean functions as arguments and return them as results.

9.2 The Typing Relation

In order to assign a type to an abstraction like `λx.t`, we need to calculate what will happen when the abstraction is applied to some argument. The next question that arises is: how do we know what type of arguments to expect? There are two possible responses: either we can simply annotate the λ-abstraction with the intended type of its arguments, or else we can analyze the body of the abstraction to see how the argument is used and try to deduce, from this, what type it should have. For now, we choose the first alternative. Instead of just $\lambda x.t$, we will write $\lambda x{:}T_1.t_2$, where the annotation on the bound variable tells us to assume that the argument will be of type T_1.

In general, languages in which type annotations in terms are used to help guide the typechecker are called *explicitly typed*. Languages in which we ask the typechecker to *infer* or *reconstruct* this information are called *implicitly typed*. (In the λ-calculus literature, the term *type-assignment systems* is also used.) Most of this book will concentrate on explicitly typed languages; implicit typing is explored in Chapter 22.

Once we know the type of the argument to the abstraction, it is clear that the type of the function's result will be just the type of the body t_2, where occurrences of x in t_2 are assumed to denote terms of type T_1. This intuition is captured by the following typing rule:

$$\frac{x{:}T_1 \vdash t_2 : T_2}{\vdash \lambda x{:}T_1.t_2 : T_1 \to T_2} \qquad \text{(T-ABS)}$$

Since terms may contain nested λ-abstractions, we will need, in general, to talk about several such assumptions. This changes the typing relation from a two-place relation, $t : T$, to a three-place relation, $\Gamma \vdash t : T$, where Γ is a set of assumptions about the types of the free variables in t.

Formally, a *typing context* (also called a *type environment*) Γ is a sequence of variables and their types, and the "comma" operator extends Γ by adding a new binding on the right. The empty context is sometimes written $\emptyset$, but usually we just omit it, writing $\vdash t : T$ for "The closed term t has type T under the empty set of assumptions."

To avoid confusion between the new binding and any bindings that may already appear in Γ, we require that the name x be chosen so that it is distinct from the variables bound by Γ. Since our convention is that variables bound by λ-abstractions may be renamed whenever convenient, this condition can always be satisfied by renaming the bound variable if necessary. Γ can thus be thought of as a finite function from variables to their types. Following this intuition, we write *dom*(Γ) for the set of variables bound by Γ.

The rule for typing abstractions has the general form

$$\frac{\Gamma, x{:}T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x{:}T_1.t_2 : T_1 \to T_2} \qquad \text{(T-ABS)}$$

where the premise adds one more assumption to those in the conclusion.

The typing rule for variables also follows immediately from this discussion: a variable has whatever type we are currently assuming it to have.

$$\frac{\mathtt{x}{:}\mathtt{T} \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathtt{T}} \qquad \text{(T-VAR)}$$

The premise $\mathtt{x}{:}\mathtt{T} \in \Gamma$ is read "The type assumed for x in Γ is T."

Finally, we need a typing rule for applications.

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{11}{\to}\mathtt{T}_{12} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_{11}}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}_{12}} \qquad \text{(T-APP)}$$

If $\mathtt{t}_1$ evaluates to a function mapping arguments in $\mathtt{T}_{11}$ to results in $\mathtt{T}_{12}$ (under the assumption that the values represented by its free variables have the types assumed for them in Γ), and if $\mathtt{t}_2$ evaluates to a result in $\mathtt{T}_{11}$, then the result of applying $\mathtt{t}_1$ to $\mathtt{t}_2$ will be a value of type $\mathtt{T}_{12}$.

The typing rules for the boolean constants and conditional expressions are the same as before (Figure 8-1). Note, though, that the metavariable T in the rule for conditionals

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{Bool} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T} \qquad \Gamma \vdash \mathtt{t}_3 : \mathtt{T}}{\Gamma \vdash \mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 : \mathtt{T}} \qquad \text{(T-IF)}$$

can now be instantiated to any function type, allowing us to type conditionals whose branches are functions:[1]

```
  if true then (λx:Bool. x) else (λx:Bool. not x);
▸ (λx:Bool. x) : Bool → Bool
```

These typing rules are summarized in Figure 9-1 (along with the syntax and evaluation rules, for the sake of completeness). The highlighted regions in the figure indicate material that is new with respect to the untyped lambda-calculus—both new rules and new bits added to old rules. As we did with booleans and numbers, we have split the definition of the full calculus into two pieces: the *pure* simply typed lambda-calculus with no base types at all, shown in this figure, and a separate set of rules for booleans, which we have already seen in Figure 8-1 (we must add a context Γ to every typing statement in that figure, of course).

We often use the symbol $\lambda_{\to}$ to refer to the simply typed lambda-calculus (we use the same symbol for systems with different sets of base types).

9.2.1 EXERCISE [★]: The pure simply typed lambda-calculus with no base types is actually *degenerate,* in the sense that it has no well-typed terms at all. Why? □

Instances of the typing rules for $\lambda_{\to}$ can be combined into *derivation trees,* just as we did for typed arithmetic expressions. For example, here is a derivation demonstrating that the term (λx:Bool.x) true has type Bool in the empty context.

1. Examples showing sample interactions with an implementation will display both results and their types from now on (when they are obvious, they will be sometimes be elided).

$\to$ *(typed)* *Based on λ (5-3)*

Syntax

$\mathtt{t}$	::=		*terms:*
		$\mathtt{x}$	*variable*
		$\lambda\mathtt{x}\mathtt{:T}.\mathtt{t}$	*abstraction*
		$\mathtt{t}\ \mathtt{t}$	*application*
$\mathtt{v}$	::=		*values:*
		$\lambda\mathtt{x}\mathtt{:T}.\mathtt{t}$	*abstraction value*
$\mathtt{T}$	::=		*types:*
		$\mathtt{T}\to\mathtt{T}$	*type of functions*
Γ	::=		*contexts:*
		$\varnothing$	*empty context*
		$\Gamma, \mathtt{x}\mathtt{:T}$	*term variable binding*

Evaluation $\boxed{\mathtt{t} \longrightarrow \mathtt{t}'}$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{t}_1\ \mathtt{t}_2 \longrightarrow \mathtt{t}_1'\ \mathtt{t}_2} \qquad \text{(E-App1)}$$

$$\frac{\mathtt{t}_2 \longrightarrow \mathtt{t}_2'}{\mathtt{v}_1\ \mathtt{t}_2 \longrightarrow \mathtt{v}_1\ \mathtt{t}_2'} \qquad \text{(E-App2)}$$

$$(\lambda\mathtt{x}\mathtt{:T}_{11}.\mathtt{t}_{12})\ \mathtt{v}_2 \longrightarrow [\mathtt{x} \mapsto \mathtt{v}_2]\mathtt{t}_{12} \qquad \text{(E-AppAbs)}$$

Typing $\boxed{\Gamma \vdash \mathtt{t} : \mathtt{T}}$

$$\frac{\mathtt{x}\mathtt{:T} \in \Gamma}{\Gamma \vdash \mathtt{x} : \mathtt{T}} \qquad \text{(T-Var)}$$

$$\frac{\Gamma, \mathtt{x}\mathtt{:T}_1 \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \lambda\mathtt{x}\mathtt{:T}_1.\mathtt{t}_2 : \mathtt{T}_1\to\mathtt{T}_2} \qquad \text{(T-Abs)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{11}\to\mathtt{T}_{12} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_{11}}{\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{T}_{12}} \qquad \text{(T-App)}$$

Figure 9-1: Pure simply typed lambda-calculus ($\lambda_\to$)

$$\dfrac{\dfrac{\dfrac{\mathtt{x}\mathtt{:Bool} \in \mathtt{x}\mathtt{:Bool}}{\mathtt{x}\mathtt{:Bool} \vdash \mathtt{x} : \mathtt{Bool}}\ \text{T-Var}}{\vdash \lambda\mathtt{x}\mathtt{:Bool}.\mathtt{x} : \mathtt{Bool}\to\mathtt{Bool}}\ \text{T-Abs} \qquad \dfrac{}{\vdash \mathtt{true} : \mathtt{Bool}}\ \text{T-True}}{\vdash (\lambda\mathtt{x}\mathtt{:Bool}.\mathtt{x})\ \mathtt{true} : \mathtt{Bool}}\ \text{T-App}$$

9.2.2 Exercise [$\star$ $\not\rightarrow$]: Show (by drawing derivation trees) that the following terms have the indicated types:

1. $\mathtt{f}\mathtt{:Bool}\to\mathtt{Bool} \vdash \mathtt{f}\ (\mathtt{if}\ \mathtt{false}\ \mathtt{then}\ \mathtt{true}\ \mathtt{else}\ \mathtt{false}) : \mathtt{Bool}$

2. $\mathtt{f}\mathtt{:Bool}\to\mathtt{Bool} \vdash \lambda\mathtt{x}\mathtt{:Bool}.\ \mathtt{f}\ (\mathtt{if}\ \mathtt{x}\ \mathtt{then}\ \mathtt{false}\ \mathtt{else}\ \mathtt{x}) : \mathtt{Bool}\to\mathtt{Bool}$ □

9.2.3 Exercise [$\star$]: Find a context Γ under which the term $\mathtt{f}\ \mathtt{x}\ \mathtt{y}$ has type $\mathtt{Bool}$. Can you give a simple description of the set of *all* such contexts? □

9.3 Properties of Typing

As in Chapter 8, we need to develop a few basic lemmas before we can prove type safety. Most of these are similar to what we saw before—we just need to add contexts to the typing relation and add clauses to each proof for λ-abstractions, applications, and variables. The only significant new requirement is a *substitution lemma* for the typing relation (Lemma 9.3.8).

First off, an *inversion lemma* records a collection of observations about how typing derivations are built: the clause for each syntactic form tells us "if a term of this form is well typed, then its subterms must have types of these forms. . ."

9.3.1 Lemma [Inversion of the typing relation]:

1. If $\Gamma \vdash \mathtt{x} : \mathsf{R}$, then $\mathtt{x}{:}\mathsf{R} \in \Gamma$.
2. If $\Gamma \vdash \lambda \mathtt{x}{:}\mathsf{T}_1.\ \mathtt{t}_2 : \mathsf{R}$, then $\mathsf{R} = \mathsf{T}_1 \to \mathsf{R}_2$ for some R_2 with $\Gamma, \mathtt{x}{:}\mathsf{T}_1 \vdash \mathtt{t}_2 : \mathsf{R}_2$.
3. If $\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathsf{R}$, then there is some type T_{11} such that $\Gamma \vdash \mathtt{t}_1 : \mathsf{T}_{11} \to \mathsf{R}$ and $\Gamma \vdash \mathtt{t}_2 : \mathsf{T}_{11}$.
4. If $\Gamma \vdash \mathtt{true} : \mathsf{R}$, then $\mathsf{R} = \mathsf{Bool}$.
5. If $\Gamma \vdash \mathtt{false} : \mathsf{R}$, then $\mathsf{R} = \mathsf{Bool}$.
6. If $\Gamma \vdash \mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 : \mathsf{R}$, then $\Gamma \vdash \mathtt{t}_1 : \mathsf{Bool}$ and $\Gamma \vdash \mathtt{t}_2, \mathtt{t}_3 : \mathsf{R}$. □

Proof: Immediate from the definition of the typing relation. □

9.3.2 Exercise [Recommended, ★★★]: Is there any context Γ and type T such that $\Gamma \vdash \mathtt{x}\ \mathtt{x} : \mathsf{T}$? If so, give Γ and T and show a typing derivation for $\Gamma \vdash \mathtt{x}\ \mathtt{x} : \mathsf{T}$; if not, prove it. □

In §9.2, we chose an explicitly typed presentation of the calculus to simplify the job of typechecking. This involved adding type annotations to bound variables in function abstractions, but nowhere else. In what sense is this "enough"? One answer is provided by the "uniqueness of types" theorem, which tells us that well-typed terms are in one-to-one correspondence with their typing derivations: the typing derivation can be recovered uniquely from the term (and, of course, vice versa). In fact, the correspondence is so straightforward that, in a sense, there is little difference between the term and the derivation.

9.3.3 Theorem [Uniqueness of Types]: In a given typing context Γ, a term $\mathtt{t}$ (with free variables all in the domain of Γ) has at most one type. That is, if a term is typable, then its type is unique. Moreover, there is just one derivation of this typing built from the inference rules that generate the typing relation. □

Proof: Exercise. The proof is actually so direct that there is almost nothing to say; but writing out some of the details is good practice in "setting up" proofs about the typing relation. □

For many of the type systems that we will see later in the book, this simple correspondence between terms and derivations will not hold: a single term will be assigned many types, and each of these will be justified by many typing derivations. In these systems, there will often be significant work involved in showing that typing derivations can be recovered effectively from terms.

Next, a canonical forms lemma tells us the possible shapes of values of various types.

9.3.4 Lemma [Canonical Forms]:

1. If `v` is a value of type `Bool`, then `v` is either `true` or `false`.
2. If `v` is a value of type $T_1 \to T_2$, then $v = \lambda x{:}T_1.t_2$. □

Proof: Straightforward. (Similar to the proof of the canonical forms lemma for arithmetic expressions, 8.3.1.) □

Using the canonical forms lemma, we can prove a progress theorem analogous to Theorem 8.3.2. The statement of the theorem needs one small change: we are interested only in *closed* terms, with no free variables. For open terms, the progress theorem actually fails: a term like `f true` is a normal form, but not a value. However, this failure does not represent a defect in the language, since complete programs—which are the terms we actually care about evaluating—are always closed.

9.3.5 Theorem [Progress]: Suppose `t` is a closed, well-typed term (that is, $\vdash t : T$ for some `T`). Then either `t` is a value or else there is some t' with $t \longrightarrow t'$. □

Proof: Straightforward induction on typing derivations. The cases for boolean constants and conditions are exactly the same as in the proof of progress for typed arithmetic expressions (8.3.2). The variable case cannot occur (because `t` is closed). The abstraction case is immediate, since abstractions are values.

The only interesting case is the one for application, where $t = t_1\ t_2$ with $\vdash t_1 : T_{11}\to T_{12}$ and $\vdash t_2 : T_{11}$. By the induction hypothesis, either t_1 is a value or else it can make a step of evaluation, and likewise t_2. If t_1 can take a step, then rule E-App1 applies to `t`. If t_1 is a value and t_2 can take a step, then rule E-App2 applies. Finally, if both t_1 and t_2 are values, then the canonical forms lemma tells us that t_1 has the form $\lambda x{:}T_{11}.t_{12}$, and so rule E-AppAbs applies to `t`. □

Our next job is to prove that evaluation preserves types. We begin by stating a couple of "structural lemmas" for the typing relation. These are not particularly interesting in themselves, but will permit us to perform some useful manipulations of typing derivations in later proofs.

The first structural lemma tells us that we may permute the elements of a context, as convenient, without changing the set of typing statements that can be derived under it. Recall (from page 101) that all the bindings in a context must have distinct names, and that, whenever we add a binding to a context, we tacitly assume that the bound name is different from all the names already bound (using Convention 5.3.4 to rename the new one if needed).

9.3.6 Lemma [Permutation]: If $\Gamma \vdash \texttt{t} : \texttt{T}$ and Δ is a permutation of Γ, then $\Delta \vdash \texttt{t} : \texttt{T}$. Moreover, the latter derivation has the same depth as the former. □

Proof: Straightforward induction on typing derivations. □

9.3.7 Lemma [Weakening]: If $\Gamma \vdash \texttt{t} : \texttt{T}$ and $\texttt{x} \notin \mathit{dom}(\Gamma)$, then $\Gamma, \texttt{x:S} \vdash \texttt{t} : \texttt{T}$. Moreover, the latter derivation has the same depth as the former. □

Proof: Straightforward induction on typing derivations. □

Using these technical lemmas, we can prove a crucial property of the typing relation: that well-typedness is preserved when variables are substituted with terms of appropriate types. Similar lemmas play such a ubiquitous role in the safety proofs of programming languages that it is often called just "the substitution lemma."

9.3.8 Lemma [Preservation of types under substitution]: If $\Gamma, \texttt{x:S} \vdash \texttt{t} : \texttt{T}$ and $\Gamma \vdash \texttt{s} : \texttt{S}$, then $\Gamma \vdash [\texttt{x} \mapsto \texttt{s}]\texttt{t} : \texttt{T}$. □

Proof: By induction on a derivation of the statement $\Gamma, \texttt{x:S} \vdash \texttt{t} : \texttt{T}$. For a given derivation, we proceed by cases on the final typing rule used in the proof.[2] The most interesting cases are the ones for variables and abstractions.

Case T-Var*:* $\texttt{t} = \texttt{z}$
with $\texttt{z:T} \in (\Gamma, \texttt{x:S})$

There are two sub-cases to consider, depending on whether $\texttt{z}$ is $\texttt{x}$ or another variable. If $\texttt{z} = \texttt{x}$, then $[\texttt{x} \mapsto \texttt{s}]\texttt{z} = \texttt{s}$. The required result is then $\Gamma \vdash \texttt{s} : \texttt{S}$, which is among the assumptions of the lemma. Otherwise, $[\texttt{x} \mapsto \texttt{s}]\texttt{z} = \texttt{z}$, and the desired result is immediate.

2. Or, equivalently, by cases on the possible shapes of t, since for each syntactic constructor there is exactly one typing rule.

Case T-ABS: $\mathtt{t} = \lambda\mathtt{y}{:}\mathtt{T}_2.\mathtt{t}_1$
$\mathtt{T} = \mathtt{T}_2 \to \mathtt{T}_1$
$\Gamma, \mathtt{x}{:}\mathtt{S}, \mathtt{y}{:}\mathtt{T}_2 \vdash \mathtt{t}_1 : \mathtt{T}_1$

By convention 5.3.4, we may assume $\mathtt{x} \neq \mathtt{y}$ and $\mathtt{y} \notin FV(\mathtt{s})$. Using permutation on the given subderivation, we obtain $\Gamma, \mathtt{y}{:}\mathtt{T}_2, \mathtt{x}{:}\mathtt{S} \vdash \mathtt{t}_1 : \mathtt{T}_1$. Using weakening on the other given derivation ($\Gamma \vdash \mathtt{s} : \mathtt{S}$), we obtain $\Gamma, \mathtt{y}{:}\mathtt{T}_2 \vdash \mathtt{s} : \mathtt{S}$. Now, by the induction hypothesis, $\Gamma, \mathtt{y}{:}\mathtt{T}_2 \vdash [\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_1 : \mathtt{T}_1$. By T-ABS, $\Gamma \vdash \lambda\mathtt{y}{:}\mathtt{T}_2.\ [\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_1 : \mathtt{T}_2 \to \mathtt{T}_1$. But this is precisely the needed result, since, by the definition of substitution, $[\mathtt{x} \mapsto \mathtt{s}]\mathtt{t} = \lambda\mathtt{y}{:}\mathtt{T}_1.\ [\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_1$.

Case T-APP: $\mathtt{t} = \mathtt{t}_1\ \mathtt{t}_2$
$\Gamma, \mathtt{x}{:}\mathtt{S} \vdash \mathtt{t}_1 : \mathtt{T}_2 \to \mathtt{T}_1$
$\Gamma, \mathtt{x}{:}\mathtt{S} \vdash \mathtt{t}_2 : \mathtt{T}_2$
$\mathtt{T} = \mathtt{T}_1$

By the induction hypothesis, $\Gamma \vdash [\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_1 : \mathtt{T}_2 \to \mathtt{T}_1$ and $\Gamma \vdash [\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_2 : \mathtt{T}_2$. By T-APP, $\Gamma \vdash [\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_1\ [\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_2 : \mathtt{T}$, i.e., $\Gamma \vdash [\mathtt{x} \mapsto \mathtt{s}](\mathtt{t}_1\ \mathtt{t}_2) : \mathtt{T}$.

Case T-TRUE: $\mathtt{t} = \mathtt{true}$
$\mathtt{T} = \mathtt{Bool}$

Then $[\mathtt{x} \mapsto \mathtt{s}]\mathtt{t} = \mathtt{true}$, and the desired result, $\Gamma \vdash [\mathtt{x} \mapsto \mathtt{s}]\mathtt{t} : \mathtt{T}$, is immediate.

Case T-FALSE: $\mathtt{t} = \mathtt{false}$
$\mathtt{T} = \mathtt{Bool}$

Similar.

Case T-IF: $\mathtt{t} = \mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3$
$\Gamma, \mathtt{x}{:}\mathtt{S} \vdash \mathtt{t}_1 : \mathtt{Bool}$
$\Gamma, \mathtt{x}{:}\mathtt{S} \vdash \mathtt{t}_2 : \mathtt{T}$
$\Gamma, \mathtt{x}{:}\mathtt{S} \vdash \mathtt{t}_3 : \mathtt{T}$

Three uses of the induction hypothesis yield

$$\begin{array}{l} \Gamma \vdash [\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_1 : \mathtt{Bool} \\ \Gamma \vdash [\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_2 : \mathtt{T} \\ \Gamma \vdash [\mathtt{x} \mapsto \mathtt{s}]\mathtt{t}_3 : \mathtt{T}, \end{array}$$

from which the result follows by T-IF. □

Using the substitution lemma, we can prove the other half of the type safety property—that evaluation preserves well-typedness.

9.3.9 THEOREM [PRESERVATION]: If $\Gamma \vdash \mathtt{t} : \mathtt{T}$ and $\mathtt{t} \longrightarrow \mathtt{t}'$, then $\Gamma \vdash \mathtt{t}' : \mathtt{T}$. □

Proof: EXERCISE [RECOMMENDED, ★★★]. The structure is very similar to the proof of the type preservation theorem for arithmetic expressions (8.3.3), except for the use of the substitution lemma. □

9.3.10 Exercise [Recommended, ★★]: In Exercise 8.3.6 we investigated the *subject expansion* property for our simple calculus of typed arithmetic expressions. Does it hold for the "functional part" of the simply typed lambda-calculus? That is, suppose `t` does not contain any conditional expressions. Do $\mathtt{t} \longrightarrow \mathtt{t}'$ and $\Gamma \vdash \mathtt{t}' : \mathtt{T}$ imply $\Gamma \vdash \mathtt{t} : \mathtt{T}$? □

9.4 The Curry-Howard Correspondence

The "→" type constructor comes with typing rules of two kinds:

1. an *introduction rule* (T-Abs) describing how elements of the type can be *created,* and
2. an *elimination rule* (T-App) describing how elements of the type can be *used.*

When an introduction form (λ) is an immediate subterm of an elimination form (application), the result is a redex—an opportunity for computation.

The terminology of introduction and elimination forms is frequently useful in discussing type systems. When we come to more complex systems later in the book, we'll see a similar pattern of linked introduction and elimination rules for each type constructor we consider.

9.4.1 Exercise [★]: Which of the rules for the type `Bool` in Figure 8-1 are introduction rules and which are elimination rules? What about the rules for `Nat` in Figure 8-2? □

The introduction/elimination terminology arises from a connection between type theory and logic known as the *Curry-Howard correspondence* or *Curry-Howard isomorphism* (Curry and Feys, 1958; Howard, 1980). Briefly, the idea is that, in constructive logics, a proof of a proposition P consists of concrete *evidence* for P.[3] What Curry and Howard noticed was that such evidence has a strongly computational feel. For example, a proof of a proposition $P \supset Q$ can be viewed as a mechanical procedure that, given a proof of P, constructs a proof of Q—or, if you like, a proof of Q *abstracted on* a proof of P. Similarly, a proof of $P \wedge Q$ consists of a proof of P together with a proof of Q. This observation gives rise to the following correspondence:

3. The characteristic difference between classical and constructive logics is the omission from the latter of proof rules like the law of the *excluded middle,* which says that, for every proposition Q, either Q holds or $\neg Q$ does. To prove $Q \vee \neg Q$ in a constructive logic, we must provide evidence either for Q or for $\neg Q$.

Logic	Programming languages
propositions	types
proposition $P \supset Q$	type `P→Q`
proposition $P \wedge Q$	type `P×Q` (see §11.6)
proof of proposition P	term `t` of type `P`
proposition P is provable	type `P` is inhabited (by some term)

On this view, a term of the simply typed lambda-calculus is a proof of a logical proposition corresponding to its type. Computation—reduction of lambda-terms—corresponds to the logical operation of proof simplification by *cut elimination*. The Curry-Howard correspondence is also called the *propositions as types* analogy. Thorough discussions of this correspondence can be found in many places, including Girard, Lafont, and Taylor (1989), Gallier (1993), Sørensen and Urzyczyn (1998), Pfenning (2001), Goubault-Larrecq and Mackie (1997), and Simmons (2000).

The beauty of the Curry-Howard correspondence is that it is not limited to a particular type system and one related logic—on the contrary, it can be extended to a huge variety of type systems and logics. For example, System F (Chapter 23), whose parametric polymorphism involves quantification over types, corresponds precisely to a second-order constructive logic, which permits quantification over propositions. System F_ω (Chapter 30) corresponds to a higher-order logic. Indeed, the correspondence has often been exploited to transfer new developments between the fields. Thus, Girard's *linear logic* (1987) gives rise to the idea of *linear type systems* (Wadler, 1990, Wadler, 1991, Turner, Wadler, and Mossin, 1995, Hodas, 1992, Mackie, 1994, Chirimar, Gunter, and Riecke, 1996, Kobayashi, Pierce, and Turner, 1996, and many others), while *modal logics* have been used to help design frameworks for *partial evaluation* and *run-time code generation* (see Davies and Pfenning, 1996, Wickline, Lee, Pfenning, and Davies, 1998, and other sources cited there).

9.5 Erasure and Typability

In Figure 9-1, we defined the evaluation relation directly on simply typed terms. Although type annotations play no role in evaluation—we don't do any sort of run-time checking to ensure that functions are applied to arguments of appropriate types—we do carry along these annotations inside of terms as we evaluate them.

Most compilers for full-scale programming languages actually avoid carrying annotations at run time: they are used during typechecking (and during code generation, in more sophisticated compilers), but do not appear in the compiled form of the program. In effect, programs are converted back to an untyped form before they are evaluated. This style of semantics can be formalized using an *erasure* function mapping simply typed terms into the corresponding untyped terms.

9.5.1 Definition: The *erasure* of a simply typed term $\mathtt{t}$ is defined as follows:

$$\begin{aligned} \mathit{erase}(\mathtt{x}) &= \mathtt{x} \\ \mathit{erase}(\lambda\mathtt{x}{:}\mathtt{T}_1.\,\mathtt{t}_2) &= \lambda\mathtt{x}.\ \mathit{erase}(\mathtt{t}_2) \\ \mathit{erase}(\mathtt{t}_1\ \mathtt{t}_2) &= \mathit{erase}(\mathtt{t}_1)\ \mathit{erase}(\mathtt{t}_2) \end{aligned}$$ □

Of course, we expect that the two ways of presenting the semantics of the simply typed calculus actually coincide: it doesn't really matter whether we evaluate a typed term directly, or whether we erase it and evaluate the underlying untyped term. This expectation is formalized by the following theorem, summarized by the slogan "evaluation commutes with erasure" in the sense that these operations can be performed in either order—we reach the same term by evaluating and then erasing as we do by erasing and then evaluating:

9.5.2 Theorem:

1. If $\mathtt{t} \longrightarrow \mathtt{t}'$ under the typed evaluation relation, then $\mathit{erase}(\mathtt{t}) \longrightarrow \mathit{erase}(\mathtt{t}')$.
2. If $\mathit{erase}(\mathtt{t}) \longrightarrow \mathtt{m}'$ under the typed evaluation relation, then there is a simply typed term $\mathtt{t}'$ such that $\mathtt{t} \longrightarrow \mathtt{t}'$ and $\mathit{erase}(\mathtt{t}') = \mathtt{m}'$. □

Proof: Straightforward induction on evaluation derivations. □

Since the "compilation" we are considering here is so straightforward, Theorem 9.5.2 is obvious to the point of triviality. For more interesting languages and more interesting compilers, however, it becomes a quite important property: it tells us that a "high-level" semantics, expressed directly in terms of the language that the programmer writes, coincides with an alternative, lower-level evaluation strategy actually used by an implementation of the language.

Another interesting question arising from the erasure function is: Given an untyped lambda-term $\mathtt{m}$, can we find a simply typed term $\mathtt{t}$ that erases to $\mathtt{m}$?

9.5.3 Definition: A term $\mathtt{m}$ in the untyped lambda-calculus is said to be *typable* in $\lambda_{\rightarrow}$ if there are some simply typed term $\mathtt{t}$, type $\mathtt{T}$, and context Γ such that $\mathit{erase}(\mathtt{t}) = \mathtt{m}$ and $\Gamma \vdash \mathtt{t} : \mathtt{T}$. □

We will return to this point in more detail in Chapter 22, when we consider the closely related topic of *type reconstruction* for $\lambda_{\rightarrow}$.

9.6 Curry-Style vs. Church-Style

We have seen two different styles in which the semantics of the simply typed lambda-calculus can be formulated: as an evaluation relation defined directly on the syntax of the simply typed calculus, or as a compilation to an untyped calculus plus an evaluation relation on untyped terms. An important commonality of the two styles is that, in both, it makes sense to talk about the behavior of a term `t`, whether or not `t` is actually well typed. This form of language definition is often called *Curry-style*. We first define the terms, then define a semantics showing how they behave, then give a type system that rejects some terms whose behaviors we don't like. Semantics is prior to typing.

A rather different way of organizing a language definition is to define terms, then identify the well-typed terms, then give semantics just to these. In these so-called *Church-style* systems, typing is prior to semantics: we never even ask the question "what is the behavior of an ill-typed term?" Indeed, strictly speaking, what we actually evaluate in Church-style systems is typing *derivations,* not terms. (See §15.6 for an example of this.)

Historically, implicitly typed presentations of lambda-calculi are often given in the Curry style, while Church-style presentations are common only for explicitly typed systems. This has led to some confusion of terminology: "Church-style" is sometimes used when describing an explicitly typed *syntax* and "Curry-style" for implicitly typed.

9.7 Notes

The simply typed lambda-calculus is studied in Hindley and Seldin (1986), and in even greater detail in Hindley's monograph (1997).

Well-typed programs cannot "go wrong." —*Robin Milner (1978)*