us that \( t_1 : t \). Since we also know \( t_1 : \text{Nat} \), we can apply the induction hypothesis to obtain \( t'_1 : \text{Nat} \), from which we obtain \( \text{succ} \ t'_1 : \text{Nat} \). Note that \( t' : T \), by applying rule T-Succ.

8.3.4 Exercise [*** →†]: Restructure this proof so that it goes by induction on evaluation derivations rather than typing derivations.

The preservation theorem is often called subject reduction (or subject evaluation)—the intuition being that a typing statement \( t : T \) can be thought of as a sentence, "\( t \) has type \( T \)." The term \( t \) is the subject of this sentence, and the subject reduction property then says that the truth of the sentence is preserved under reduction of the subject.

Unlike uniqueness of types, which holds in some type systems and not in others, progress and preservation will be basic requirements for all of the type systems that we consider.

8.3.5 Exercise [†]: The evaluation rule E-PIZERO (Figure 3-2) is a bit counterintuitive: we might feel that it makes more sense for the predecessor of zero to be undefined, rather than being defined to be zero. Can we achieve this simply by removing the rule from the definition of single-step evaluation?

8.3.6 Exercise [***, Recommended]: Having seen the subject reduction property, it is reasonable to wonder whether the opposite property—subject expansion—also holds. Is it always the case that, if \( t \rightarrow t' \) and \( t' : T \), then \( t : T \)? If so, prove it. If not, give a counterexample.

8.3.7 Exercise [Recommended, ***]: Suppose our evaluation relation is defined in the big-step style, as in Exercise 3.5.17. How should the intuitive property of type safety be formalized?

8.3.8 Exercise [Recommended, **†]: Suppose our evaluation relation is augmented with rules for reducing nonsensical terms to an explicit wrong state, as in Exercise 3.5.16. Now how should type safety be formalized?

---

The road from untyped to typed universes has been followed many times, in many different fields, and largely for the same reasons.

—Luca Cardelli and Peter Wegner (1985)

4. There are languages where these properties do not hold, but which can nevertheless be considered to be type-safe. For example, if we formalize the operational semantics of Java in a small-step style (Elart, Kahan, and Mitton, 1993; Leavens, 1996); in Java, type safety is maintained by the compiler (see Chapter 19 for details). However, this should be considered an artifact of the formalization, rather than a defect in the language itself, since it disappears, for example, in a big-step presentation of the semantics.

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: \( \text{Bool} \), classifying terms whose evaluation yields a boolean, and \( \text{Nat} \), 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., \( \text{if } 0 \text{ then } 1 \text{ else } 2 \)) as well as some terms that actually behave fine during evaluation, but for which our static classification is too conservative (like \( \text{if } \text{true then } 0 \text{ else false} \)).

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

\[
\text{if } \text{long and tricky computation} \text{ then true else } (\lambda x.x)
\]

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 (0, 1). The associated OCaml implementation is \textit{FaLaSimple}.
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 \( \lambda \)-abstraction the type \( \rightarrow \), we can classify both simple terms like
\( \lambda x. x \) and compound terms like \( \text{if} \ \text{true} \ \text{then} \ (\lambda x. \text{true}) \ \text{else} \ (\lambda x. \text{true}) \) as yielding functions.
But this rough analysis is clearly too conservative: functions like \( \lambda x. \text{true} \) and \( \lambda x. \lambda y. y \) are lumped together in the same type \( \rightarrow \), ignoring the fact that
applying the first to \text{true} yields a boolean, while applying the second to \text{true}
yields another function. In general, in order to give a useful type to the re-
sult 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 \( \rightarrow \) by an infinite family of
types of the form \( T_1 \to T_2 \), 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 \( \text{Bool} \) is generated by the following grammar:

\[
T ::= \text{Bool} \quad \text{type of booleans} \\
T \to T \quad \text{type of functions}
\]

The type constructor \( \to \) is right-associative—that is, the expression \( T_1 \to T_2 \to T_3 \) stands for \( (T_1 \to T_2) \to T_3 \).

For example \( \text{Bool} \to \text{Bool} \) is the type of functions mapping boolean argu-
ments to boolean results. \( (\text{Bool} \to \text{Bool}) \to (\text{Bool} \to \text{Bool}) \) or, equivalently,
\( (\text{Bool} \to \text{Bool}) \to \text{Bool} \) 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 \( \lambda x. t \), 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 ex-
pect? There are two possible responses: either we can simply annotate the

\[ \lambda \text{abstraction with the intended type of its arguments, or else we can ana-
lyze 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 al-
ternative. Instead of just \( \lambda x. t \), we will write \( \lambda x : T_1. t \) 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 \( \lambda \)-calculus literature, the term type-assignment systems is also
used.) Most of this book will concentrate on explicitly typed languages; im-
plex 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 \( t \), where
occurrences of \( x \) in \( t \) are renamed to denote terms of type \( T_1 \). This intuition is captured by the following typing rule:

\[
\frac{\Gamma, x : T_1 \vdash t : T_2}{\Gamma \vdash \lambda x : T_1. t : T_1 \to T_2} \quad \text{(T-Abs)}
\]

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

Formally, a typing context (also called a \textit{contextive}) \( \Gamma \) is a sequence
of variables and their types, and the “comma” operator extends \( \Gamma \) by adding
a new binding on the right. The empty context is sometimes written \( \varnothing \), 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 \( \Gamma \), we require that the name \( x \) be chosen so that it is distinct
from the variables bound by \( \Gamma \). Since our convention is that variables bound
by \( \lambda \)-abstractions may be renamed whenever convenient, this condition can
always be satisfied by renaming the bound variable if necessary. \( \Gamma \) can thus
be thought of as a finite function from variables to their types. Following this
intuition, we write \( \text{dom}(\Gamma) \) for the set of variables bound by \( \Gamma \).

The rule for typing abstractions has the general form

\[
\frac{\Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1. t_2 : T_1 \to T_2} \quad \text{(T-Abs)}
\]

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.
9 Simply Typed Lambda-Calculus

The premise \( x : T \in \Gamma \) is read "The type assumed for \( x \) in \( \Gamma \) is \( T \)."

Finally, we need a typing rule for applications.

\[
\begin{align*}
\Gamma &\vdash t_1 : T_{11} \rightarrow T_{12} \\
\Gamma &\vdash t_2 : T_{11} \\
\Gamma &\vdash t_3 : T_{12} \\
\Gamma &\vdash t_1 t_2 : T_{12} \\
\end{align*}
\] (T-App)

If \( t_1 \) evaluates to a function mapping arguments in \( T_{11} \) to results in \( T_{12} \) (under the assumption that the values represented by its free variables have the types assumed for them in \( \Gamma \)), and if \( t_2 \) evaluates to a result in \( T_{11} \), then the result of applying \( t_2 \) to \( t_3 \) will be a value of type \( T_{12} \).

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

\[
\Gamma \vdash t_1 : \text{Bool} \\
\Gamma \vdash t_2 : T \\
\Gamma \vdash t_3 : T \\
\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T
\] (T-If)

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

\[
\text{if true then } (\lambda x : \text{Bool}. \; x) \text{ else } (\lambda x : \text{Bool}. \; \text{not } x) ;
\]

\[(\lambda x : \text{Bool}. \; x) : \text{Bool} \rightarrow \text{Bool}\]

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 purely 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 \( \Gamma \) to every typing statement in that figure, of course).

We often use the symbol \( \lambda \_ \) 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 purely 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 \( \lambda \_ \) can be combined into derivation trees, just as we did for typed arithmetic expressions. For example, here is a derivation demonstrating that the term \( (\lambda x : \text{Bool}. x) \text{ true } \) has type \( \text{Bool} \) in the empty context.

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

9.2.2 Exercise**: Show (by drawing derivation trees) that the following terms have the indicated types:

1. \( f : \text{Bool} \rightarrow \text{Bool} \) \( (\text{if false then false else false}) : \text{Bool} \)

2. \( f : \text{Bool} \rightarrow \text{Bool} \) \( \lambda x : \text{Bool}. f (\text{if } x \text{ then false else } x) : \text{Bool} \rightarrow \text{Bool} \)

9.2.3 Exercise**: Find a context \( \Gamma \) under which the term \( x y \) has type \( \text{Bool} \). 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 Γ ⊢ x : R, then x : R ∈ Γ.
2. If Γ ⊢ λx:T₁₁ . t₂ : R, then R = T₁₁ → R₂ for some R₂ with Γ, x : T₁₁ ⊢ t₂ : R₂.
3. If Γ ⊢ t₁ ; t₂ : R, then there is some type T₁₁ such that Γ ⊢ t₁ : T₁₁ → R and Γ ⊢ t₂ : T₁₁.
4. If Γ ⊢ true : R, then R = Bool.
5. If Γ ⊢ false : R, then R = Bool.
6. If Γ ⊢ if t₁ then t₂ else t₃ : R, then Γ ⊢ t₁ : Bool and Γ ⊢ t₂, t₃ : 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 Γ ⊢ x : ? If so, give Γ and T and show a typing derivation for Γ ⊢ x : 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 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 T₁ → T₂, then v = λx:T₁ . t₂.

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, ⊢ t : T for some T). Then either t is a value or else there is some t' with t → 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 = t₁ t₂ with ⊢ t₁ : T₁₁ → T₁₂ and ⊢ t₂ : T₁₁. By the induction hypothesis, either t₁ is a value or else it can make a step of evaluation, and likewise t₂. If t₁ can take a step, then rule E-APP₁ applies to t. If t₁ is a value and t₂ can take a step, then rule E-APP₂ applies. Finally, if both t₁ and t₂ are values, then the canonical forms lemma tells us that t₁ has the form λx:T₁₁ . t₁₂ and so rule E-APP₃ applies to t.
9 Simply Typed Lambda-Calculus

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 \( \Gamma \vdash t : T \) and \( \Delta \) is a permutation of \( \Gamma \), then \( \Delta \vdash t : T \). Moreover, the latter derivation has the same depth as the former.

**Proof:** Straightforward induction on typing derivations.

9.3.7 **LEMMA [WEAKENING]:** If \( \Gamma \vdash t : T \) and \( x \notin \text{dom}(\Gamma) \), then \( \Gamma, x : S \vdash t : T \). 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 they is often called just "the substitution lemma."

9.3.8 **LEMMA [PRESERVATION OF TYPES UNDER SUBSTITUTION]:** If \( \Gamma, x : S \vdash t : T \) and \( \Gamma \vdash s : S \), then \( \Gamma \vdash [x \leftarrow s]t : T \).

**Proof:** By induction on a derivation of the statement \( \Gamma, x : S \vdash t : T \). For a given derivation, we proceed by cases on the final typing rule used in the proof.\(^3\) The most interesting cases are the ones for variables and abstractions.

- **Case T-VAR:** \( t = z \) with \( z : T \in (\Gamma, x : S) \)

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

3. Or, equivalently, by cases on the possible shapes of \( t \); since for each syntactic constructor there is exactly one typing rule.

9.3.9 **THEOREM [PRESERVATION]:** If \( \Gamma \vdash t : T \) and \( t \to t' \), then \( \Gamma \vdash t' : T \).

**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.