8 *Typed Arithmetic Expressions*

In Chapter 3, we used a simple language of boolean and arithmetic expressions to introduce basic tools for the precise description of syntax and evaluation. We now return to this simple language and augment it with static types. Again, the type system itself is nearly trivial, but it provides a setting in which to introduce concepts that will recur throughout the book.

8.1 Types

Recall the syntax for arithmetic expressions:

`t` ::=		*terms:*
	`true`	*constant true*
	`false`	*constant false*
	`if t then t else t`	*conditional*
	`0`	*constant zero*
	`succ t`	*successor*
	`pred t`	*predecessor*
	`iszero t`	*zero test*

We saw in Chapter 3 that evaluating a term can either result in a value...

`v` ::=		*values:*
	`true`	*true value*
	`false`	*false value*
	`nv`	*numeric value*
`nv` ::=		*numeric values:*
	`0`	*zero value*
	`succ nv`	*successor value*

The system studied in this chapter is the typed calculus of booleans and numbers (Figure 8-2). The corresponding OCaml implementation is `tyarith`.

or else get *stuck* at some stage, by reaching a term like `pred false`, for which no evaluation rule applies.

Stuck terms correspond to meaningless or erroneous programs. We would therefore like to be able to tell, without actually evaluating a term, that its evaluation will definitely *not* get stuck. To do this, we need to be able to distinguish between terms whose result will be a numeric value (since these are the only ones that should appear as arguments to `pred`, `succ`, and `iszero`) and terms whose result will be a boolean (since only these should appear as the guard of a conditional). We introduce two *types*, `Nat` and `Bool`, for classifying terms in this way. The metavariables `S`, `T`, `U`, etc. will be used throughout the book to range over types.

Saying that "a term `t` has type `T`" (or "`t` belongs to `T`," or "`t` is an element of `T`") means that `t` "obviously" evaluates to a value of the appropriate form—where by "obviously" we mean that we can see this *statically,* without doing any evaluation of `t`. For example, the term `if true then false else true` has type `Bool`, while `pred (succ (pred (succ 0)))` has type `Nat`. However, our analysis of the types of terms will be *conservative,* making use only of static information. This means that we will not be able to conclude that terms like `if (iszero 0) then 0 else false` or even `if true then 0 else false` have any type at all, even though their evaluation does not, in fact, get stuck.

8.2 The Typing Relation

The typing relation for arithmetic expressions, written[1] "`t : T`", is defined by a set of inference rules assigning types to terms, summarized in Figures 8-1 and 8-2. As in Chapter 3, we give the rules for booleans and those for numbers in two different figures, since later on we will sometimes want to refer to them separately.

The rules T-True and T-False in Figure 8-1 assign the type `Bool` to the boolean constants `true` and `false`. Rule T-If assigns a type to a conditional expression based on the types of its subexpressions: the guard t_1 must evaluate to a boolean, while t_2 and t_3 must both evaluate to values of the *same* type. The two uses of the single metavariable `T` express the constraint that the result of the `if` is the type of the `then`- and `else`- branches, and that this may be any type (either `Nat` or `Bool` or, when we get to calculi with more interesting sets of types, any other type).

The rules for numbers in Figure 8-2 have a similar form. T-Zero gives the type `Nat` to the constant `0`. T-Succ gives a term of the form `succ` t_1 the type `Nat`, as long as t_1 has type `Nat`. Likewise, T-Pred and T-IsZero say that `pred`

1. The symbol $\in$ is often used instead of `:`.

$\mathbb{B}$ *(typed)* *Extends* **B** *(3-1)*

New syntactic forms

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

New typing rules $\boxed{\mathtt{t : T}}$

$$\mathtt{true : Bool} \qquad \text{(T-True)}$$

$$\mathtt{false : Bool} \qquad \text{(T-False)}$$

$$\frac{\mathtt{t_1 : Bool} \quad \mathtt{t_2 : T} \quad \mathtt{t_3 : T}}{\mathtt{if\ t_1\ then\ t_2\ else\ t_3 : T}} \qquad \text{(T-If)}$$

Figure 8-1: Typing rules for booleans (B)

$\mathbb{B}$ $\mathbb{N}$ *(typed)* *Extends* **NB** *(3-2) and 8-1*

New syntactic forms

T	::=	...	*types:*
		Nat	*type of natural numbers*

New typing rules $\boxed{\mathtt{t : T}}$

$$\mathtt{0 : Nat} \qquad \text{(T-Zero)}$$

$$\frac{\mathtt{t_1 : Nat}}{\mathtt{succ\ t_1 : Nat}} \qquad \text{(T-Succ)}$$

$$\frac{\mathtt{t_1 : Nat}}{\mathtt{pred\ t_1 : Nat}} \qquad \text{(T-Pred)}$$

$$\frac{\mathtt{t_1 : Nat}}{\mathtt{iszero\ t_1 : Bool}} \qquad \text{(T-IsZero)}$$

Figure 8-2: Typing rules for numbers (NB)

yields a `Nat` when its argument has type `Nat` and `iszero` yields a `Bool` when its argument has type `Nat`.

8.2.1 Definition: Formally, the *typing relation* for arithmetic expressions is the smallest binary relation between terms and types satisfying all instances of the rules in Figures 8-1 and 8-2. A term `t` is *typable* (or *well typed*) if there is some `T` such that `t : T`. □

When reasoning about the typing relation, we will often make statements like "If a term of the form `succ` $\mathtt{t_1}$ has any type at all, then it has type `Nat`." The following lemma gives us a compendium of basic statements of this form, each following immediately from the shape of the corresponding typing rule.

8.2.2 LEMMA [INVERSION OF THE TYPING RELATION]:

1. If $\mathtt{true} : \mathsf{R}$, then $\mathsf{R} = \mathtt{Bool}$.
2. If $\mathtt{false} : \mathsf{R}$, then $\mathsf{R} = \mathtt{Bool}$.
3. If $\mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 : \mathsf{R}$, then $\mathtt{t}_1 : \mathtt{Bool}$, $\mathtt{t}_2 : \mathsf{R}$, and $\mathtt{t}_3 : \mathsf{R}$.
4. If $\mathtt{0} : \mathsf{R}$, then $\mathsf{R} = \mathtt{Nat}$.
5. If $\mathtt{succ}\ \mathtt{t}_1 : \mathsf{R}$, then $\mathsf{R} = \mathtt{Nat}$ and $\mathtt{t}_1 : \mathtt{Nat}$.
6. If $\mathtt{pred}\ \mathtt{t}_1 : \mathsf{R}$, then $\mathsf{R} = \mathtt{Nat}$ and $\mathtt{t}_1 : \mathtt{Nat}$.
7. If $\mathtt{iszero}\ \mathtt{t}_1 : \mathsf{R}$, then $\mathsf{R} = \mathtt{Bool}$ and $\mathtt{t}_1 : \mathtt{Nat}$. □

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

The inversion lemma is sometimes called the *generation lemma* for the typing relation, since, given a valid typing statement, it shows how a proof of this statement could have been generated. The inversion lemma leads directly to a recursive algorithm for calculating the types of terms, since it tells us, for a term of each syntactic form, how to calculate its type (if it has one) from the types of its subterms. We will return to this point in detail in Chapter 9.

8.2.3 EXERCISE [★ ↛]: Prove that every subterm of a well-typed term is well typed. □

In §3.5 we introduced the concept of evaluation derivations. Similarly, a *typing derivation* is a tree of instances of the typing rules. Each pair $(\mathtt{t},\mathsf{T})$ in the typing relation is justified by a typing derivation with conclusion $\mathtt{t} : \mathsf{T}$. For example, here is the derivation tree for the typing statement "$\mathtt{if}\ \mathtt{iszero}\ \mathtt{0}\ \mathtt{then}\ \mathtt{0}\ \mathtt{else}\ \mathtt{pred}\ \mathtt{0} : \mathtt{Nat}$":

$$\dfrac{\dfrac{\dfrac{}{\mathtt{0} : \mathtt{Nat}}\ \textsc{T-Zero}}{\mathtt{iszero}\ \mathtt{0} : \mathtt{Bool}}\ \textsc{T-IsZero} \qquad \dfrac{}{\mathtt{0} : \mathtt{Nat}}\ \textsc{T-Zero} \qquad \dfrac{\dfrac{}{\mathtt{0} : \mathtt{Nat}}\ \textsc{T-Zero}}{\mathtt{pred}\ \mathtt{0} : \mathtt{Nat}}\ \textsc{T-Pred}}{\mathtt{if}\ \mathtt{iszero}\ \mathtt{0}\ \mathtt{then}\ \mathtt{0}\ \mathtt{else}\ \mathtt{pred}\ \mathtt{0} : \mathtt{Nat}}\ \textsc{T-If}$$

In other words, *statements* are formal assertions about the typing of programs, *typing rules* are implications between statements, and *derivations* are deductions based on typing rules.

8.2.4 THEOREM [UNIQUENESS OF TYPES]: Each term $\mathtt{t}$ has at most one type. That is, if $\mathtt{t}$ is typable, then its type is unique. Moreover, there is just one derivation of this typing built from the inference rules in Figures 8-1 and 8-2. □

Proof: Straightforward structural induction on `t`, using the appropriate clause of the inversion lemma (plus the induction hypothesis) for each case. □

In the simple type system we are dealing with in this chapter, every term has a single type (if it has any type at all), and there is always just one derivation tree witnessing this fact. Later on—e.g., when we get to type systems with subtyping in Chapter 15—both of these properties will be relaxed: a single term may have many types, and there may in general be many ways of deriving the statement that a given term has a given type.

Properties of the typing relation will often be proved by induction on derivation trees, just as properties of the evaluation relation are typically proved by induction on evaluation derivations. We will see many examples of induction on typing derivations, beginning in the next section.

8.3 Safety = Progress + Preservation

The most basic property of this type system or any other is *safety* (also called *soundness*): well-typed terms do not "go wrong." We have already chosen how to formalize what it means for a term to go wrong: it means reaching a "stuck state" (Definition 3.5.15) that is not designated as a final value but where the evaluation rules do not tell us what to do next. What we want to know, then, is that well-typed terms do not get stuck. We show this in two steps, commonly known as the *progress* and *preservation* theorems.[2]

> *Progress:* A well-typed term is not stuck (either it is a value or it can take a step according to the evaluation rules).
>
> *Preservation:* If a well-typed term takes a step of evaluation, then the resulting term is also well typed.[3]

These properties together tell us that a well-typed term can never reach a stuck state during evaluation.

For the proof of the progress theorem, it is convenient to record a couple of facts about the possible shapes of the *canonical forms* of types `Bool` and `Nat` (i.e., the well-typed values of these types).

2. The slogan "safety is progress plus preservation" (using a canonical forms lemma) was articulated by Harper; a variant was proposed by Wright and Felleisen (1994).

3. In most of the type systems we will consider, evaluation preserves not only well-typedness but the exact types of terms. In some systems, however, types can change during evaluation. For example, in systems with subtyping (Chapter 15), types can become smaller (more informative) during evaluation.

8.3.1 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 `Nat`, then `v` is a numeric value according to the grammar in Figure 3-2. □

Proof: For part (1), according to the grammar in Figures 3-1 and 3-2, values in this language can have four forms: `true`, `false`, `0`, and `succ nv`, where `nv` is a numeric value. The first two cases give us the desired result immediately. The last two cannot occur, since we assumed that `v` has type `Bool` and cases 4 and 5 of the inversion lemma tell us that `0` and `succ nv` can have only type `Nat`, not `Bool`. Part (2) is similar. □

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

Proof: By induction on a derivation of `t : T`. The T-TRUE, T-FALSE, and T-ZERO cases are immediate, since `t` in these cases is a value. For the other cases, we argue as follows.

Case T-IF: $t = \text{if } t_1 \text{ then } t_2 \text{ else } t_3$

$t_1 : \text{Bool} \quad t_2 : T \quad t_3 : T$

By the induction hypothesis, either t_1 is a value or else there is some t_1' such that $t_1 \longrightarrow t_1'$. If t_1 is a value, then the canonical forms lemma (8.3.1) assures us that it must be either `true` or `false`, in which case either E-IFTRUE or E-IFFALSE applies to `t`. On the other hand, if $t_1 \longrightarrow t_1'$, then, by T-IF, $t \longrightarrow \text{if } t_1' \text{ then } t_2 \text{ else } t_3$.

Case T-SUCC: $t = \text{succ } t_1 \qquad t_1 : \text{Nat}$

By the induction hypothesis, either t_1 is a value or else there is some t_1' such that $t_1 \longrightarrow t_1'$. If t_1 is a value, then, by the canonical forms lemma, it must be a numeric value, in which case so is `t`. On the other hand, if $t_1 \longrightarrow t_1'$, then, by E-SUCC, $\text{succ } t_1 \longrightarrow \text{succ } t_1'$.

Case T-PRED: $t = \text{pred } t_1 \qquad t_1 : \text{Nat}$

By the induction hypothesis, either t_1 is a value or else there is some t_1' such that $t_1 \longrightarrow t_1'$. If t_1 is a value, then, by the canonical forms lemma, it must be a numeric value, i.e., either `0` or $\text{succ } nv_1$ for some nv_1, and one of the rules E-PREDZERO or E-PREDSUCC applies to `t`. On the other hand, if $t_1 \longrightarrow t_1'$, then, by E-PRED, $\text{pred } t_1 \longrightarrow \text{pred } t_1'$.

Case T-ISZERO: $t = \text{iszero } t_1 \qquad t_1 : \text{Nat}$

Similar. □

The proof that types are preserved by evaluation is also quite straightforward for this system.

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

Proof: By induction on a derivation of $\mathtt{t} : \mathtt{T}$. At each step of the induction, we assume that the desired property holds for all subderivations (i.e., that if $\mathtt{s} : \mathtt{S}$ and $\mathtt{s} \longrightarrow \mathtt{s}'$, then $\mathtt{s}' : \mathtt{S}$, whenever $\mathtt{s} : \mathtt{S}$ is proved by a subderivation of the present one) and proceed by case analysis on the final rule in the derivation. (We show only a subset of the cases; the others are similar.)

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

If the last rule in the derivation is T-TRUE, then we know from the form of this rule that $\mathtt{t}$ must be the constant $\mathtt{true}$ and $\mathtt{T}$ must be $\mathtt{Bool}$. But then $\mathtt{t}$ is a value, so it cannot be the case that $\mathtt{t} \longrightarrow \mathtt{t}'$ for any $\mathtt{t}'$, and the requirements of the theorem are vacuously satisfied.

Case T-IF*:* $\mathtt{t} = \mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 \quad \mathtt{t}_1 : \mathtt{Bool} \quad \mathtt{t}_2 : \mathtt{T} \quad \mathtt{t}_3 : \mathtt{T}$

If the last rule in the derivation is T-IF, then we know from the form of this rule that $\mathtt{t}$ must have the form $\mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3$, for some $\mathtt{t}_1$, $\mathtt{t}_2$, and $\mathtt{t}_3$. We must also have subderivations with conclusions $\mathtt{t}_1 : \mathtt{Bool}$, $\mathtt{t}_2 : \mathtt{T}$, and $\mathtt{t}_3 : \mathtt{T}$. Now, looking at the evaluation rules with $\mathtt{if}$ on the left-hand side (Figure 3-1), we find that there are three rules by which $\mathtt{t} \longrightarrow \mathtt{t}'$ can be derived: E-IFTRUE, E-IFFALSE, and E-IF. We consider each case separately (omitting the E-FALSE case, which is similar to E-TRUE).

> *Subcase* E-IFTRUE*:* $\mathtt{t}_1 = \mathtt{true} \qquad \mathtt{t}' = \mathtt{t}_2$
>
> If $\mathtt{t} \longrightarrow \mathtt{t}'$ is derived using E-IFTRUE, then from the form of this rule we see that $\mathtt{t}_1$ must be $\mathtt{true}$ and the resulting term $\mathtt{t}'$ is the second subexpression $\mathtt{t}_2$. This means we are finished, since we know (by the assumptions of the T-IF case) that $\mathtt{t}_2 : \mathtt{T}$, which is what we need.
>
> *Subcase* E-IF*:* $\mathtt{t}_1 \longrightarrow \mathtt{t}_1' \qquad \mathtt{t}' = \mathtt{if}\ \mathtt{t}_1'\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3$
>
> From the assumptions of the T-IF case, we have a subderivation of the original typing derivation whose conclusion is $\mathtt{t}_1 : \mathtt{Bool}$. We can apply the induction hypothesis to this subderivation, obtaining $\mathtt{t}_1' : \mathtt{Bool}$. Combining this with the facts (from the assumptions of the T-IF case) that $\mathtt{t}_2 : \mathtt{T}$ and $\mathtt{t}_3 : \mathtt{T}$, we can apply rule T-IF to conclude that $\mathtt{if}\ \mathtt{t}_1'\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 : \mathtt{T}$, that is $\mathtt{t}' : \mathtt{T}$.

Case T-ZERO*:* $\mathtt{t} = \mathtt{0} \qquad \mathtt{T} = \mathtt{Nat}$

Can't happen (for the same reasons as T-TRUE above).

Case T-SUCC*:* $\mathtt{t} = \mathtt{succ}\ \mathtt{t}_1 \qquad \mathtt{T} = \mathtt{Nat} \qquad \mathtt{t}_1 : \mathtt{Nat}$

By inspecting the evaluation rules in Figure 3-2, we see that there is just one rule, E-SUCC, that can be used to derive $\mathtt{t} \longrightarrow \mathtt{t}'$. The form of this rule tells us that $t_1 \longrightarrow t_1'$. Since we also know t_1 : Nat, we can apply the induction hypothesis to obtain t_1' : Nat, from which we obtain succ t_1' : Nat, i.e., 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.[4]

8.3.5 Exercise [★]: The evaluation rule E-PredZero (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 $t \longrightarrow t'$ 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 (Flatt, Krishnamurthi, and Felleisen, 1998a; Igarashi, Pierce, and Wadler, 1999), type preservation in the form we have given it here fails (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: `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

$$\lambda\mathtt{x.t} : \to$$

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 $\mathtt{T}_1 \to \mathtt{T}_2$, each classifying functions that expect arguments of type $\mathtt{T}_1$ and return results of type $\mathtt{T}_2$.

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 $\mathtt{T}_1 \to \mathtt{T}_2 \to \mathtt{T}_3$ stands for $\mathtt{T}_1 \to (\mathtt{T}_2 \to \mathtt{T}_3)$. □

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 $\lambda\mathtt{x}.\mathtt{t}$, we will write $\lambda\mathtt{x}{:}\mathsf{T}_1.\mathtt{t}_2$, 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 $\mathtt{t}_2$, where occurrences of $\mathtt{x}$ in $\mathtt{t}_2$ are assumed to denote terms of type T_1. This intuition is captured by the following typing rule:

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

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, $\mathtt{t} : \mathsf{T}$, to a three-place relation, $\Gamma \vdash \mathtt{t} : \mathsf{T}$, where Γ is a set of assumptions about the types of the free variables in $\mathtt{t}$.

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 $\vdash \mathtt{t} : \mathsf{T}$ for "The closed term $\mathtt{t}$ 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 $\mathtt{x}$ 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 $dom(\Gamma)$ for the set of variables bound by Γ.

The rule for typing abstractions has the general form

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

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 $\mathtt{x}$ in Γ is $\mathtt{T}$."

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 $\mathtt{T}$ 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 $(\lambda\mathtt{x}{:}\mathtt{Bool}.\mathtt{x})\ \mathtt{true}$ has type $\mathtt{Bool}$ 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).

→ *(typed)* *Based on* λ *(5-3)*

Syntax

t	::=		*terms:*
		x	*variable*
		λx:T.t	*abstraction*
		t t	*application*
v	::=		*values:*
		λx:T.t	*abstraction value*
T	::=		*types:*
		T→T	*type of functions*
Γ	::=		*contexts:*
		∅	*empty context*
		Γ, x:T	*term variable binding*

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

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

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

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

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

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

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

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

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

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

9.2.2 Exercise [★ ↛]: Show (by drawing derivation trees) that the following terms have the indicated types:

1. `f:Bool→Bool` ⊢ `f (if false then true else false)` : `Bool`

2. `f:Bool→Bool` ⊢ `λx:Bool. f (if x then false else x)` : `Bool→Bool` □

9.2.3 Exercise [★]: Find a context Γ under which the term `f x y` has type `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 $\Gamma \vdash \mathtt{x} : \mathtt{R}$, then $\mathtt{x}:\mathtt{R} \in \Gamma$.
2. If $\Gamma \vdash \lambda\mathtt{x}:\mathtt{T}_1.\ \mathtt{t}_2 : \mathtt{R}$, then $\mathtt{R} = \mathtt{T}_1 \to \mathtt{R}_2$ for some $\mathtt{R}_2$ with $\Gamma, \mathtt{x}:\mathtt{T}_1 \vdash \mathtt{t}_2 : \mathtt{R}_2$.
3. If $\Gamma \vdash \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{R}$, then there is some type $\mathtt{T}_{11}$ such that $\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_{11} \to \mathtt{R}$ and $\Gamma \vdash \mathtt{t}_2 : \mathtt{T}_{11}$.
4. If $\Gamma \vdash \mathtt{true} : \mathtt{R}$, then $\mathtt{R} = \mathtt{Bool}$.
5. If $\Gamma \vdash \mathtt{false} : \mathtt{R}$, then $\mathtt{R} = \mathtt{Bool}$.
6. If $\Gamma \vdash \mathtt{if}\ \mathtt{t}_1\ \mathtt{then}\ \mathtt{t}_2\ \mathtt{else}\ \mathtt{t}_3 : \mathtt{R}$, then $\Gamma \vdash \mathtt{t}_1 : \mathtt{Bool}$ and $\Gamma \vdash \mathtt{t}_2, \mathtt{t}_3 : \mathtt{R}$. □

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

9.3.2 Exercise [Recommended, ★★★]: Is there any context Γ and type $\mathtt{T}$ such that $\Gamma \vdash \mathtt{x}\ \mathtt{x} : \mathtt{T}$? If so, give Γ and $\mathtt{T}$ and show a typing derivation for $\Gamma \vdash \mathtt{x}\ \mathtt{x} : \mathtt{T}$; 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 $\mathtt{T}_1 \to \mathtt{T}_2$, then $\mathtt{v} = \lambda\mathtt{x}{:}\mathtt{T}_1\mathtt{.t}_2$. □

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, $\vdash \mathtt{t} : \mathtt{T}$ for some `T`). Then either `t` is a value or else there is some $\mathtt{t}'$ with $\mathtt{t} \longrightarrow \mathtt{t}'$. □

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 $\mathtt{t} = \mathtt{t}_1\ \mathtt{t}_2$ with $\vdash \mathtt{t}_1 : \mathtt{T}_{11} \to \mathtt{T}_{12}$ and $\vdash \mathtt{t}_2 : \mathtt{T}_{11}$. By the induction hypothesis, either $\mathtt{t}_1$ is a value or else it can make a step of evaluation, and likewise $\mathtt{t}_2$. If $\mathtt{t}_1$ can take a step, then rule E-App1 applies to `t`. If $\mathtt{t}_1$ is a value and $\mathtt{t}_2$ can take a step, then rule E-App2 applies. Finally, if both $\mathtt{t}_1$ and $\mathtt{t}_2$ are values, then the canonical forms lemma tells us that $\mathtt{t}_1$ has the form $\lambda\mathtt{x}{:}\mathtt{T}_{11}\mathtt{.t}_{12}$, 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 $\Gamma \vdash \mathtt{t} : \mathtt{T}$ and Δ is a permutation of Γ, then $\Delta \vdash \mathtt{t} : \mathtt{T}$. Moreover, the latter derivation has the same depth as the former. □

Proof: Straightforward induction on typing derivations. □

9.3.7 Lemma [Weakening]: If $\Gamma \vdash \mathtt{t} : \mathtt{T}$ and $\mathtt{x} \notin dom(\Gamma)$, then $\Gamma, \mathtt{x}{:}\mathtt{S} \vdash \mathtt{t} : \mathtt{T}$. 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 $\Gamma, \mathtt{x}{:}\mathtt{S} \vdash \mathtt{t} : \mathtt{T}$ and $\Gamma \vdash \mathtt{s} : \mathtt{S}$, then $\Gamma \vdash [\mathtt{x} \mapsto \mathtt{s}]\mathtt{t} : \mathtt{T}$. □

Proof: By induction on a derivation of the statement $\Gamma, \mathtt{x}{:}\mathtt{S} \vdash \mathtt{t} : \mathtt{T}$. 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*:* $\mathtt{t} = \mathtt{z}$
with $\mathtt{z}{:}\mathtt{T} \in (\Gamma, \mathtt{x}{:}\mathtt{S})$

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

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

Case T-ABS*:* $t = \lambda y{:}T_2.t_1$
$T = T_2 \to T_1$
$\Gamma, x{:}S, y{:}T_2 \vdash t_1 : T_1$

By convention 5.3.4, we may assume $x \neq y$ and $y \notin FV(s)$. Using permutation on the given subderivation, we obtain $\Gamma, y{:}T_2, x{:}S \vdash t_1 : T_1$. Using weakening on the other given derivation ($\Gamma \vdash s : S$), we obtain $\Gamma, y{:}T_2 \vdash s : S$. Now, by the induction hypothesis, $\Gamma, y{:}T_2 \vdash [x \mapsto s]t_1 : T_1$. By T-ABS, $\Gamma \vdash \lambda y{:}T_2.\ [x \mapsto s]t_1 : T_2 \to T_1$. But this is precisely the needed result, since, by the definition of substitution, $[x \mapsto s]t = \lambda y{:}T_1.\ [x \mapsto s]t_1$.

Case T-APP*:* $t = t_1\ t_2$
$\Gamma, x{:}S \vdash t_1 : T_2 \to T_1$
$\Gamma, x{:}S \vdash t_2 : T_2$
$T = T_1$

By the induction hypothesis, $\Gamma \vdash [x \mapsto s]t_1 : T_2 \to T_1$ and $\Gamma \vdash [x \mapsto s]t_2 : T_2$. By T-APP, $\Gamma \vdash [x \mapsto s]t_1\ [x \mapsto s]t_2 : T$, i.e., $\Gamma \vdash [x \mapsto s](t_1\ t_2) : T$.

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

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

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

Similar.

Case T-IF*:* $t = \mathtt{if}\ t_1\ \mathtt{then}\ t_2\ \mathtt{else}\ t_3$
$\Gamma, x{:}S \vdash t_1 : \mathtt{Bool}$
$\Gamma, x{:}S \vdash t_2 : T$
$\Gamma, x{:}S \vdash t_3 : T$

Three uses of the induction hypothesis yield

$$\begin{array}{l} \Gamma \vdash [x \mapsto s]t_1 : \mathtt{Bool} \\ \Gamma \vdash [x \mapsto s]t_2 : T \\ \Gamma \vdash [x \mapsto s]t_3 : T, \end{array}$$

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 $\Gamma \vdash t : T$ and $t \longrightarrow t'$, 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. □

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 t is defined as follows:

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

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

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 m, can we find a simply typed term t that erases to m?

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