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 `:`.

𝔹 *(typed)* *Extends* **B** *(3-1)*

New syntactic forms

T ::= *types:*

Bool *type of booleans*

New typing rules t : T

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

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

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

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

𝔹 ℕ *(typed)* *Extends* **NB** *(3-2) and 8-1*

New syntactic forms

T ::= ... *types:*

Nat *type of natural numbers*

New typing rules t : T

0 : Nat (T-Zero)

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

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

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

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` 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 `true : R`, then `R = Bool`.
2. If `false : R`, then `R = Bool`.
3. If `if t1 then t2 else t3 : R`, then `t1 : Bool`, `t2 : R`, and `t3 : R`.
4. If `0 : R`, then `R = Nat`.
5. If `succ t1 : R`, then `R = Nat` and `t1 : Nat`.
6. If `pred t1 : R`, then `R = Nat` and `t1 : Nat`.
7. If `iszero t1 : R`, then `R = Bool` and `t1 : 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 (`t`, `T`) in the typing relation is justified by a typing derivation with conclusion `t : T`. For example, here is the derivation tree for the typing statement "`if iszero 0 then 0 else pred 0 : Nat`":

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

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 `t` has at most one type. That is, if `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: $t = \texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3$, $t_1 : \texttt{Bool}$, $t_2 : T$, $t_3 : T$

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, $t \longrightarrow \texttt{if } t_1' \texttt{ then } t_2 \texttt{ else } t_3$.

Case T-SUCC: $t = \texttt{succ } t_1$, $t_1 : \texttt{Nat}$

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, $\texttt{succ } t_1 \longrightarrow \texttt{succ } t_1'$.

Case T-PRED: $t = \texttt{pred } t_1$, $t_1 : \texttt{Nat}$

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 $\texttt{succ } nv_1$ 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, $\texttt{pred } t_1 \longrightarrow \texttt{pred } t_1'$.

Case T-ISZERO: $t = \texttt{iszero } t_1$, $t_1 : \texttt{Nat}$

Similar. □

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

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

Proof: By induction on a derivation of $t : T$. At each step of the induction, we assume that the desired property holds for all subderivations (i.e., that if $s : S$ and $s \longrightarrow s'$, then $s' : S$, whenever $s : S$ 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: $t = \texttt{true}$ $T = \texttt{Bool}$

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

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

If the last rule in the derivation is T-IF, then we know from the form of this rule that t must have the form $\texttt{if } t_1 \texttt{ then } t_2 \texttt{ else } t_3$, for some t_1, t_2, and t_3. We must also have subderivations with conclusions $t_1 : \texttt{Bool}$, $t_2 : T$, and $t_3 : T$. Now, looking at the evaluation rules with $\texttt{if}$ on the left-hand side (Figure 3-1), we find that there are three rules by which $t \longrightarrow t'$ 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: $t_1 = \texttt{true}$ $t' = t_2$
>
> If $t \longrightarrow t'$ is derived using E-IFTRUE, then from the form of this rule we see that t_1 must be $\texttt{true}$ and the resulting term t' is the second subexpression t_2. This means we are finished, since we know (by the assumptions of the T-IF case) that $t_2 : T$, which is what we need.
>
> *Subcase* E-IF: $t_1 \longrightarrow t_1'$ $t' = \texttt{if } t_1' \texttt{ then } t_2 \texttt{ else } t_3$
>
> From the assumptions of the T-IF case, we have a subderivation of the original typing derivation whose conclusion is $t_1 : \texttt{Bool}$. We can apply the induction hypothesis to this subderivation, obtaining $t_1' : \texttt{Bool}$. Combining this with the facts (from the assumptions of the T-IF case) that $t_2 : T$ and $t_3 : T$, we can apply rule T-IF to conclude that $\texttt{if } t_1' \texttt{ then } t_2 \texttt{ else } t_3 : T$, that is $t' : T$.

Case T-ZERO: $t = \texttt{0}$ $T = \texttt{Nat}$

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

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

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 $t \longrightarrow t'$. 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.