11 *Simple Extensions*

The simply typed lambda-calculus has enough structure to make its theoretical properties interesting, but it is not yet much of a programming language. In this chapter, we begin to close the gap with more familiar languages by introducing a number of familiar features that have straightforward treatments at the level of typing. An important theme throughout the chapter is the concept of *derived forms*.

11.1 Base Types

Every programming language provides a variety of *base types*—sets of simple, unstructured values such as numbers, booleans, or characters—plus appropriate primitive operations for manipulating these values. We have already examined natural numbers and booleans in detail; as many other base types as the language designer wants can be added in exactly the same way.

Besides `Bool` and `Nat`, we will occasionally use the base types `String` (with elements like `"hello"`) and `Float` (with elements like `3.14159`) to spice up the examples in the rest of the book.

For theoretical purposes, it is often useful to abstract away from the details of particular base types and their operations, and instead simply suppose that our language comes equipped with some set $\mathcal{A}$ of *uninterpreted* or *unknown* base types, with no primitive operations on them at all. This is accomplished simply by including the elements of $\mathcal{A}$ (ranged over by the metavariable A) in the set of types, as shown in Figure 11-1. We use the letter $\mathcal{A}$ for base types, rather than $\mathcal{B}$, to avoid confusion with the symbol $\mathbb{B}$, which we have used to indicate the presence of booleans in a given system. $\mathcal{A}$ can be thought of as standing for *atomic types*—another name that is often used for base types, because they have no internal structure as far as the type system

---

The systems studied in this chapter are various extensions of the pure typed lambda-calculus (Figure 9-1). The associated OCaml implementation, `fullsimple`, includes all the extensions.

→ A *Extends λ→ (9-1)*

New syntactic forms

T	::=	...	*types:*
		A	*base type*

Figure 11-1: Uninterpreted base types

is concerned. We will use A, B, C, etc. as the names of base types. Note that, as we did before with variables and type variables, we are using A both as a base type and as a metavariable ranging over base types, relying on context to tell us which is intended in a particular instance.

Is an uninterpreted type useless? Not at all. Although we have no way of naming its elements directly, we can still bind variables that range over the elements of a base type. For example, the function[1]

```
λx:A. x;
▸ <fun> : A → A
```

is the identity function on the elements of A, whatever these may be. Likewise,

```
λx:B. x;
▸ <fun> : B → B
```

is the identity function on B, while

```
λf:A→A. λx:A. f(f(x));
▸ <fun> : (A→A) → A → A
```

is a function that repeats two times the behavior of some given function f on an argument x.

11.2 The Unit Type

Another useful base type, found especially in languages in the ML family, is the singleton type Unit described in Figure 11-2. In contrast to the uninterpreted base types of the previous section, this type is interpreted in the

1. From now on, we will save space by eliding the bodies of λ-abstractions—writing them as just <fun>—when we display the results of evaluation.

$\to$ `Unit` *Extends* $\lambda_\to$ *(9-1)*

New syntactic forms			
t	::=	...	*terms:*
		`unit`	*constant* `unit`
v	::=	...	*values:*
		`unit`	*constant* `unit`
T	::=	...	*types:*
		`Unit`	*unit type*

New typing rules $\boxed{\Gamma \vdash t : T}$

$$\Gamma \vdash \texttt{unit} : \texttt{Unit} \qquad \text{(T-UNIT)}$$

New derived forms

$$t_1;t_2 \stackrel{\text{def}}{=} (\lambda x{:}\texttt{Unit}.t_2)\ t_1 \quad \text{where } x \notin FV(t_2)$$

Figure 11-2: Unit type

simplest possible way: we explicitly introduce a single element—the term constant `unit` (written with a small `u`)—and a typing rule making `unit` an element of `Unit`. We also add `unit` to the set of possible result values of computations—indeed, `unit` is the *only* possible result of evaluating an expression of type `Unit`.

Even in a purely functional language, the type `Unit` is not completely without interest,[2] but its main application is in languages with side effects, such as assignments to reference cells—a topic we will return to in Chapter 13. In such languages, it is often the side effect, not the result, of an expression that we care about; `Unit` is an appropriate result type for such expressions.

This use of `Unit` is similar to the role of the `void` type in languages like `C` and `Java`. The name `void` suggests a connection with the empty type `Bot` (cf. §15.4), but the usage of `void` is actually closer to our `Unit`.

11.3 Derived Forms: Sequencing and Wildcards

In languages with side effects, it is often useful to evaluate two or more expressions in sequence. The *sequencing notation* $t_1;t_2$ has the effect of evaluating t_1, throwing away its trivial result, and going on to evaluate t_2.

---

2. The reader may enjoy the following little puzzle:

11.2.1 EXERCISE [★★★]: Is there a way of constructing a sequence of terms t_1, t_2, ..., in the simply typed lambda-calculus with *only* the base type `Unit`, such that, for each n, the term t_n has size at most $O(n)$ but requires at least $O(2^n)$ steps of evaluation to reach a normal form? □

There are actually two different ways to formalize sequencing. One is to follow the same pattern we have used for other syntactic forms: add $\mathtt{t}_1\mathtt{;t}_2$ as a new alternative in the syntax of terms, and then add two evaluation rules

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

$$\mathtt{unit;t}_2 \longrightarrow \mathtt{t}_2 \qquad \text{(E-SEQNEXT)}$$

and a typing rule

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{Unit} \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{T}_2}{\Gamma \vdash \mathtt{t}_1\mathtt{;t}_2 : \mathtt{T}_2} \qquad \text{(T-SEQ)}$$

capturing the intended behavior of `;`.

An alternative way of formalizing sequencing is simply to regard $\mathtt{t}_1\mathtt{;t}_2$ as an *abbreviation* for the term $(\lambda\mathtt{x:Unit.t}_2)\ \mathtt{t}_1$, where the variable `x` is chosen *fresh*—i.e., different from all the free variables of $\mathtt{t}_2$.

It is intuitively fairly clear that these two presentations of sequencing add up to the same thing as far as the programmer is concerned: the high-level typing and evaluation rules for sequencing can be *derived* from the abbreviation of $\mathtt{t}_1\mathtt{;t}_2$ as $(\lambda\mathtt{x:Unit.t}_2)\ \mathtt{t}_1$. This intuitive correspondence is captured more formally by arguing that typing and evaluation both "commute" with the expansion of the abbreviation.

11.3.1 THEOREM [SEQUENCING IS A DERIVED FORM]: Write λ^E ("*E*" for *external language*) for the simply typed lambda-calculus with the `Unit` type, the sequencing construct, and the rules E-SEQ, E-SEQNEXT, and T-SEQ, and λ^I ("*I*" for *internal language*) for the simply typed lambda-calculus with `Unit` only. Let $e \in \lambda^E \to \lambda^I$ be the *elaboration function* that translates from the external to the internal language by replacing every occurrence of $\mathtt{t}_1\mathtt{;t}_2$ with $(\lambda\mathtt{x:Unit.t}_2)\ \mathtt{t}_1$, where `x` is chosen fresh in each case. Now, for each term `t` of λ^E, we have

- $\mathtt{t} \longrightarrow_E \mathtt{t}'$ iff $e(\mathtt{t}) \longrightarrow_I e(\mathtt{t}')$
- $\Gamma \vdash^E \mathtt{t} : \mathtt{T}$ iff $\Gamma \vdash^I e(\mathtt{t}) : \mathtt{T}$

where the evaluation and typing relations of λ^E and λ^I are annotated with E and I, respectively, to show which is which. □

Proof: Each direction of each "iff" proceeds by straightforward induction on the structure of `t`. □

Theorem 11.3.1 justifies our use of the term *derived form,* since it shows that the typing and evaluation behavior of the sequencing construct can be derived from those of the more fundamental operations of abstraction and application. The advantage of introducing features like sequencing as derived forms rather than as full-fledged language constructs is that we can extend the surface syntax (i.e., the language that the programmer actually uses to write programs) without adding any complexity to the internal language about which theorems such as type safety must be proved. This method of factoring the descriptions of language features can already be found in the Algol 60 report (Naur et al., 1963), and it is heavily used in many more recent language definitions, notably the Definition of Standard ML (Milner, Tofte, and Harper, 1990; Milner, Tofte, Harper, and MacQueen, 1997).

Derived forms are often called *syntactic sugar,* following Landin. Replacing a derived form with its lower-level definition is called *desugaring.*

Another derived form that will be useful in examples later on is the "wildcard" convention for variable binders. It often happens (for example, in terms created by desugaring sequencing) that we want to write a "dummy" lambda-abstraction in which the parameter variable is not actually used in the body of the abstraction. In such cases, it is annoying to have to explicitly choose a name for the bound variable; instead, we would like to replace it by a *wildcard binder,* written `_`. That is, we will write `λ_:S.t` to abbreviate `λx:S.t`, where `x` is some variable not occurring in `t`.

11.3.2 EXERCISE [★]: Give typing and evaluation rules for wildcard abstractions, and prove that they can be derived from the abbreviation stated above. □

11.4 Ascription

Another simple feature that will frequently come in handy later is the ability to explicitly *ascribe* a particular type to a given term (i.e., to record in the text of the program an assertion that this term has this type). We write "`t as T`" for "the term `t`, to which we ascribe the type `T`." The typing rule T-ASCRIBE for this construct (cf. Figure 11-3) simply verifies that the ascribed type `T` is, indeed, the type of `t`. The evaluation rule E-ASCRIBE is equally straightforward: it just throws away the ascription, leaving `t` free to evaluate as usual.

There are a number of situations where ascription can be useful in programming. One common one is *documentation.* It can sometimes become difficult for a reader to keep track of the types of the subexpressions of a large compound expression. Judicious use of ascription can make such programs much easier to follow. Similarly, in a particularly complex expression, it may not even be clear to the *writer* what the types of all the subexpressions are. Sprinkling in a few ascriptions is a good way of clarifying the programmer's

→ as *Extends* $\lambda_{\rightarrow}$ *(9-1)*

New syntactic forms

$\texttt{t} ::= \ldots$ *terms:*

$\quad \texttt{t as T}$ *ascription*

New evaluation rules $\boxed{\texttt{t} \longrightarrow \texttt{t}'}$

$$\texttt{v}_1 \texttt{ as T} \longrightarrow \texttt{v}_1 \quad \text{(E-Ascribe)}$$

$$\frac{\texttt{t}_1 \longrightarrow \texttt{t}_1'}{\texttt{t}_1 \texttt{ as T} \longrightarrow \texttt{t}_1' \texttt{ as T}} \quad \text{(E-Ascribe1)}$$

New typing rules $\boxed{\Gamma \vdash \texttt{t} : \texttt{T}}$

$$\frac{\Gamma \vdash \texttt{t}_1 : \texttt{T}}{\Gamma \vdash \texttt{t}_1 \texttt{ as T} : \texttt{T}} \quad \text{(T-Ascribe)}$$

Figure 11-3: Ascription

thinking. Indeed, ascription is sometimes a valuable aid in pinpointing the source of puzzling type errors.

Another use of ascription is for controlling the *printing* of complex types. The typecheckers used to check the examples shown in this book—and the accompanying OCaml implementations whose names begin with the prefix `full`—provide a simple mechanism for introducing abbreviations for long or complex type expressions. (The abbreviation mechanism is omitted from the other implementations to make them easier to read and modify.) For example, the declaration

```
UU = Unit→Unit;
```

makes `UU` an abbreviation for `Unit→Unit` in what follows. Wherever `UU` is seen, `Unit→Unit` is understood. We can write, for example:

```
(λf:UU. f unit) (λx:Unit. x);
```

During type-checking, these abbreviations are expanded automatically as necessary. Conversely, the typecheckers attempt to collapse abbreviations whenever possible. (Specifically, each time they calculate the type of a subterm, they check whether this type exactly matches any of the currently defined abbreviations, and if so replace the type by the abbreviation.) This normally gives reasonable results, but occasionally we may want a type to print differently, either because the simple matching strategy causes the typechecker to miss an opportunity to collapse an abbreviation (for example, in systems where the fields of record types can be permuted, it will not recognize that `{a:Bool,b:Nat}` is interchangeable with `{b:Nat,a:Bool}`), or because we want the type to print differently for some other reason. For example, in

```
λf:Unit→Unit. f;
▸ <fun> : (Unit→Unit) → UU
```

the abbreviation UU is collapsed in the result of the function, but not in its argument. If we want the type to print as UU→UU, we can either change the type annotation on the abstraction

```
λf:UU. f;
▸ <fun> : UU → UU
```

or else add an ascription to the whole abstraction:

```
(λf:Unit→Unit. f) as UU→UU;
▸ <fun> : UU → UU
```

When the typechecker processes an ascription `t as T`, it expands any abbreviations in T while checking that t has type T, but then yields T itself, exactly as written, as the type of the ascription. This use of ascription to control the printing of types is somewhat particular to the way the implementations in this book have been engineered. In a full-blown programming language, mechanisms for abbreviation and type printing will either be unnecessary (as in Java, for example, where by construction all types are represented by short names—cf. Chapter 19) or else much more tightly integrated into the language (as in OCaml—cf. Rémy and Vouillon, 1998; Vouillon, 2000).

A final use of ascription that will be discussed in more detail in §15.5 is as a mechanism for *abstraction*. In systems where a given term t may have many different types (for example, systems with subtyping), ascription can be used to "hide" some of these types by telling the typechecker to treat t as if it had only a smaller set of types. The relation between ascription and *casting* is also discussed in §15.5.

11.4.1 Exercise [Recommended, ★★]: (1) Show how to formulate ascription as a derived form. Prove that the "official" typing and evaluation rules given here correspond to your definition in a suitable sense. (2) Suppose that, instead of the pair of evaluation rules E-Ascribe and E-Ascribe1, we had given an "eager" rule

$$\mathtt{t}_1 \ \mathtt{as}\ \mathsf{T} \longrightarrow \mathtt{t}_1 \qquad \text{(E-AscribeEager)}$$

that throws away an ascription as soon as it is reached. Can ascription still be considered as a derived form? □

$\rightarrow$ let *Extends $\lambda_\rightarrow$ (9-1)*

New syntactic forms

$$\texttt{t} ::= \ldots \qquad \textit{terms:}$$
$$\texttt{let x=t in t} \qquad \textit{let binding}$$

New evaluation rules $\boxed{\texttt{t} \longrightarrow \texttt{t}'}$

$$\texttt{let x=v}_1\texttt{ in t}_2 \longrightarrow [\texttt{x} \mapsto \texttt{v}_1]\texttt{t}_2 \qquad \text{(E-LetV)}$$

$$\frac{\texttt{t}_1 \longrightarrow \texttt{t}_1'}{\texttt{let x=t}_1\texttt{ in t}_2 \longrightarrow \texttt{let x=t}_1'\texttt{ in t}_2} \qquad \text{(E-Let)}$$

New typing rules $\boxed{\Gamma \vdash \texttt{t} : \texttt{T}}$

$$\frac{\Gamma \vdash \texttt{t}_1 : \texttt{T}_1 \qquad \Gamma, \texttt{x:T}_1 \vdash \texttt{t}_2 : \texttt{T}_2}{\Gamma \vdash \texttt{let x=t}_1\texttt{ in t}_2 : \texttt{T}_2} \qquad \text{(T-Let)}$$

Figure 11-4: Let binding

11.5 Let Bindings

When writing a complex expression, it is often useful—both for avoiding repetition and for increasing readability—to give names to some of its subexpressions. Most languages provide one or more ways of doing this. In ML, for example, we write $\texttt{let x=t}_1\texttt{ in t}_2$ to mean "evaluate the expression $\texttt{t}_1$ and bind the name `x` to the resulting value while evaluating $\texttt{t}_2$."

Our `let`-binder (summarized in Figure 11-4) follows ML's in choosing a call-by-value evaluation order, where the `let`-bound term must be fully evaluated before evaluation of the `let`-body can begin. The typing rule T-Let tells us that the type of a `let` can be calculated by calculating the type of the `let`-bound term, extending the context with a binding with this type, and in this enriched context calculating the type of the body, which is then the type of the whole `let` expression.

11.5.1 Exercise [Recommended, ★★★]: The `letexercise` typechecker (available at the book's web site) is an incomplete implementation of `let` expressions: basic parsing and printing functions are provided, but the clauses for `TmLet` are missing from the `eval1` and `typeof` functions (in their place, you'll find dummy clauses that match everything and crash the program with an assertion failure). Finish it. □

Can `let` also be defined as a derived form? Yes, as Landin showed; but the details are slightly more subtle than what we did for sequencing and ascription. Naively, it is clear that we can use a combination of abstraction and application to achieve the effect of a `let`-binding:

$$\texttt{let x=t}_1\texttt{ in t}_2 \quad \stackrel{\text{def}}{=} \quad (\lambda\texttt{x:T}_1\texttt{.t}_2)\texttt{ t}_1$$

But notice that the right-hand side of this abbreviation includes the type annotation T_1, which does not appear on the left-hand side. That is, if we imagine derived forms as being desugared during the parsing phase of some compiler, then we need to ask how the parser is supposed to know that it should generate T_1 as the type annotation on the λ in the desugared internal-language term.

The answer, of course, is that this information comes from the typechecker! We discover the needed type annotation simply by calculating the type of t_1. More formally, what this tells us is that the `let` constructor is a slightly different sort of derived form than the ones we have seen up till now: we should regard it not as a desugaring transformation on terms, but as a transformation on *typing derivations* (or, if you prefer, on terms decorated by the typechecker with the results of its analysis) that maps a derivation involving `let`

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash \mathsf{t}_1 : \mathsf{T}_1} \qquad \dfrac{\vdots}{\Gamma, \mathsf{x}{:}\mathsf{T}_1 \vdash \mathsf{t}_2 : \mathsf{T}_2}}{\Gamma \vdash \mathsf{let}\ \mathsf{x}{=}\mathsf{t}_1\ \mathsf{in}\ \mathsf{t}_2 : \mathsf{T}_2} \text{ (T-Let)}$$

to one using abstraction and application:

$$\dfrac{\dfrac{\dfrac{\vdots}{\Gamma, \mathsf{x}{:}\mathsf{T}_1 \vdash \mathsf{t}_2 : \mathsf{T}_2}}{\Gamma \vdash \lambda \mathsf{x}{:}\mathsf{T}_1.\mathsf{t}_2 : \mathsf{T}_1 \to \mathsf{T}_2} \text{ (T-Abs)} \qquad \dfrac{\vdots}{\Gamma \vdash \mathsf{t}_1 : \mathsf{T}_1}}{\Gamma \vdash (\lambda \mathsf{x}{:}\mathsf{T}_1.\mathsf{t}_2)\ \mathsf{t}_1 : \mathsf{T}_2} \text{ (T-App)}$$

Thus, `let` is "a little less derived" than the other derived forms we have seen: we can derive its evaluation behavior by desugaring it, but its typing behavior must be built into the internal language.

In Chapter 22 we will see another reason not to treat `let` as a derived form: in languages with Hindley-Milner (i.e., unification-based) polymorphism, the `let` construct is treated specially by the typechecker, which uses it for *generalizing* polymorphic definitions to obtain typings that cannot be emulated using ordinary λ-abstraction and application.

11.5.2 Exercise [★★]: Another way of defining `let` as a derived form might be to desugar it by "executing" it immediately—i.e., to regard $\mathsf{let}\ \mathsf{x}{=}\mathsf{t}_1\ \mathsf{in}\ \mathsf{t}_2$ as an abbreviation for the substituted body $[\mathsf{x} \mapsto \mathsf{t}_1]\mathsf{t}_2$. Is this a good idea? □

$\rightarrow$ $\times$ *Extends $\lambda_{\rightarrow}$ (9-1)*

New syntactic forms

$\mathtt{t}$	::=	...	*terms:*
		$\{\mathtt{t},\mathtt{t}\}$	*pair*
		$\mathtt{t.1}$	*first projection*
		$\mathtt{t.2}$	*second projection*
$\mathtt{v}$	::=	...	*values:*
		$\{\mathtt{v},\mathtt{v}\}$	*pair value*
T	::=	...	*types:*
		$\mathsf{T}_1 \times \mathsf{T}_2$	*product type*

New evaluation rules $\boxed{\mathtt{t} \longrightarrow \mathtt{t}'}$

$$\{\mathtt{v}_1,\mathtt{v}_2\}.1 \longrightarrow \mathtt{v}_1 \qquad \text{(E-PairBeta1)}$$

$$\{\mathtt{v}_1,\mathtt{v}_2\}.2 \longrightarrow \mathtt{v}_2 \qquad \text{(E-PairBeta2)}$$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{t}_1\mathtt{.1} \longrightarrow \mathtt{t}_1'\mathtt{.1}} \qquad \text{(E-Proj1)}$$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{t}_1\mathtt{.2} \longrightarrow \mathtt{t}_1'\mathtt{.2}} \qquad \text{(E-Proj2)}$$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\{\mathtt{t}_1,\mathtt{t}_2\} \longrightarrow \{\mathtt{t}_1',\mathtt{t}_2\}} \qquad \text{(E-Pair1)}$$

$$\frac{\mathtt{t}_2 \longrightarrow \mathtt{t}_2'}{\{\mathtt{v}_1,\mathtt{t}_2\} \longrightarrow \{\mathtt{v}_1,\mathtt{t}_2'\}} \qquad \text{(E-Pair2)}$$

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

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathsf{T}_1 \qquad \Gamma \vdash \mathtt{t}_2 : \mathsf{T}_2}{\Gamma \vdash \{\mathtt{t}_1,\mathtt{t}_2\} : \mathsf{T}_1 \times \mathsf{T}_2} \qquad \text{(T-Pair)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathsf{T}_{11} \times \mathsf{T}_{12}}{\Gamma \vdash \mathtt{t}_1\mathtt{.1} : \mathsf{T}_{11}} \qquad \text{(T-Proj1)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathsf{T}_{11} \times \mathsf{T}_{12}}{\Gamma \vdash \mathtt{t}_1\mathtt{.2} : \mathsf{T}_{12}} \qquad \text{(T-Proj2)}$$

Figure 11-5: Pairs

11.6 Pairs

Most programming languages provide a variety of ways of building compound data structures. The simplest of these is *pairs,* or more generally *tuples,* of values. We treat pairs in this section, then do the more general cases of tuples and labeled records in §11.7 and §11.8.[3]

The formalization of pairs is almost too simple to be worth discussing—by this point in the book, it should be about as easy to read the rules in Figure 11-5 as to wade through a description in English conveying the same information. However, let's look briefly at the various parts of the definition to emphasize the common pattern.

Adding pairs to the simply typed lambda-calculus involves adding two new forms of term—pairing, written $\{\mathtt{t}_1,\mathtt{t}_2\}$, and projection, written `t.1` for the

3. The `fullsimple` implementation does not actually provide the pairing syntax described here, since tuples are more general anyway.

first projection from `t` and `t.2` for the second projection—plus one new type constructor, $\mathsf{T}_1 \times \mathsf{T}_2$, called the *product* (or sometimes the *cartesian product*) of T_1 and T_2. Pairs are written with curly braces[4] to emphasize the connection to records in the §11.8.

For evaluation, we need several new rules specifying how pairs and projection behave. E-PairBeta1 and E-PairBeta2 specify that, when a fully evaluated pair meets a first or second projection, the result is the appropriate component. E-Proj1 and E-Proj2 allow reduction to proceed under projections, when the term being projected from has not yet been fully evaluated. E-Pair1 and E-Pair2 evaluate the parts of pairs: first the left part, and then—when a value appears on the left—the right part.

The ordering arising from the use of the metavariables `v` and `t` in these rules enforces a left-to-right evaluation strategy for pairs. For example, the compound term

```
{pred 4, if true then false else false}.1
```

evaluates (only) as follows:

```
    {pred 4, if true then false else false}.1
⟶  {3, if true then false else false}.1
⟶  {3, false}.1
⟶  3
```

We also need to add a new clause to the definition of values, specifying that $\{\mathsf{v}_1,\mathsf{v}_2\}$ is a value. The fact that the components of a pair value must themselves be values ensures that a pair passed as an argument to a function will be fully evaluated before the function body starts executing. For example:

```
    (λx:Nat×Nat. x.2) {pred 4, pred 5}
⟶  (λx:Nat×Nat. x.2) {3, pred 5}
⟶  (λx:Nat×Nat. x.2) {3,4}
⟶  {3,4}.2
⟶  4
```

The typing rules for pairs and projections are straightforward. The introduction rule, T-Pair, says that $\{\mathsf{t}_1,\mathsf{t}_2\}$ has type $\mathsf{T}_1 \times \mathsf{T}_2$ if t_1 has type T_1 and t_2 has type T_2. Conversely, the elimination rules T-Proj1 and T-Proj2 tell us that, if t_1 has a product type $\mathsf{T}_{11} \times \mathsf{T}_{12}$ (i.e., if it will evaluate to a pair), then the types of the projections from this pair are T_{11} and T_{12}.

4. The curly brace notation is a little unfortunate for pairs and tuples, since it suggests the standard mathematical notation for sets. It is more common, both in popular languages like ML and in the research literature, to enclose pairs and tuples in parentheses. Other notations such as square or angle brackets are also used.

$\rightarrow$ $\{\}$ *Extends $\lambda_\rightarrow$ (9-1)*

New syntactic forms

$\mathtt{t}$	::=	...	*terms:*
		$\{\mathtt{t}_i{}^{\,i\in 1..n}\}$	*tuple*
		$\mathtt{t.i}$	*projection*
$\mathtt{v}$	::=	...	*values:*
		$\{\mathtt{v}_i{}^{\,i\in 1..n}\}$	*tuple value*
$\mathtt{T}$	::=	...	*types:*
		$\{\mathtt{T}_i{}^{\,i\in 1..n}\}$	*tuple type*

New evaluation rules $\boxed{\mathtt{t} \longrightarrow \mathtt{t}'}$

$$\{\mathtt{v}_i{}^{\,i\in 1..n}\}.\mathtt{j} \longrightarrow \mathtt{v}_j \qquad \text{(E-ProjTuple)}$$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{t}_1.\mathtt{i} \longrightarrow \mathtt{t}_1'.\mathtt{i}} \qquad \text{(E-Proj)}$$

$$\frac{\mathtt{t}_j \longrightarrow \mathtt{t}_j'}{\{\mathtt{v}_i{}^{\,i\in 1..j-1},\mathtt{t}_j,\mathtt{t}_k{}^{\,k\in j+1..n}\} \longrightarrow \{\mathtt{v}_i{}^{\,i\in 1..j-1},\mathtt{t}_j',\mathtt{t}_k{}^{\,k\in j+1..n}\}} \qquad \text{(E-Tuple)}$$

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

$$\frac{\text{for each } i \quad \Gamma \vdash \mathtt{t}_i : \mathtt{T}_i}{\Gamma \vdash \{\mathtt{t}_i{}^{\,i\in 1..n}\} : \{\mathtt{T}_i{}^{\,i\in 1..n}\}} \qquad \text{(T-Tuple)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \{\mathtt{T}_i{}^{\,i\in 1..n}\}}{\Gamma \vdash \mathtt{t}_1.\mathtt{j} : \mathtt{T}_j} \qquad \text{(T-Proj)}$$

Figure 11-6: Tuples

11.7 Tuples

It is easy to generalize the binary products of the previous section to *n*-ary products, often called *tuples*. For example, `{1,2,true}` is a 3-tuple containing two numbers and a boolean. Its type is written `{Nat,Nat,Bool}`.

The only cost of this generalization is that, to formalize the system, we need to invent notations for uniformly describing structures of arbitrary arity; such notations are always a bit problematic, as there is some inevitable tension between rigor and readability. We write $\{\mathtt{t}_i{}^{\,i\in 1..n}\}$ for a tuple of n terms, $\mathtt{t}_1$ through $\mathtt{t}_n$, and $\{\mathtt{T}_i{}^{\,i\in 1..n}\}$ for its type. Note that n here is allowed to be 0; in this case, the range *1..n* is empty and $\{\mathtt{t}_i{}^{\,i\in 1..n}\}$ is `{}`, the empty tuple. Also, note the difference between a bare value like `5` and a one-element tuple like `{5}`: the only operation we may legally perform on the latter is projecting its first component.

Figure 11-6 formalizes tuples. The definition is similar to the definition of products (Figure 11-5), except that each rule for pairing has been generalized to the *n*-ary case, and each pair of rules for first and second projections has become a single rule for an arbitrary projection from a tuple. The only rule that deserves special comment is E-Tuple, which combines and generalizes the rules E-Pair1 and E-Pair2 from Figure 11-5. In English, it says that, if we

→ {} Extends λ→ (9-1)

New syntactic forms

$\mathtt{t}$	::=	...	*terms:*
		$\{\mathtt{l}_i\mathtt{=t}_i\ ^{i\in 1..n}\}$	*record*
		$\mathtt{t.l}$	*projection*
$\mathtt{v}$	::=	...	*values:*
		$\{\mathtt{l}_i\mathtt{=v}_i\ ^{i\in 1..n}\}$	*record value*
$\mathtt{T}$	::=	...	*types:*
		$\{\mathtt{l}_i\mathtt{:T}_i\ ^{i\in 1..n}\}$	*type of records*

New evaluation rules $\boxed{\mathtt{t} \longrightarrow \mathtt{t}'}$

$$\{\mathtt{l}_i\mathtt{=v}_i\ ^{i\in 1..n}\}\mathtt{.l}_j \longrightarrow \mathtt{v}_j \qquad \text{(E-ProjRcd)}$$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{t}_1\mathtt{.l} \longrightarrow \mathtt{t}_1'\mathtt{.l}} \qquad \text{(E-Proj)}$$

$$\frac{\mathtt{t}_j \longrightarrow \mathtt{t}_j'}{\begin{array}{l}\{\mathtt{l}_i\mathtt{=v}_i\ ^{i\in 1..j-1}\mathtt{,l}_j\mathtt{=t}_j\mathtt{,l}_k\mathtt{=t}_k\ ^{k\in j+1..n}\}\\ \longrightarrow \{\mathtt{l}_i\mathtt{=v}_i\ ^{i\in 1..j-1}\mathtt{,l}_j\mathtt{=t}_j'\mathtt{,l}_k\mathtt{=t}_k\ ^{k\in j+1..n}\}\end{array}} \qquad \text{(E-Rcd)}$$

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

$$\frac{\text{for each } i \quad \Gamma \vdash \mathtt{t}_i : \mathtt{T}_i}{\Gamma \vdash \{\mathtt{l}_i\mathtt{=t}_i\ ^{i\in 1..n}\} : \{\mathtt{l}_i\mathtt{:T}_i\ ^{i\in 1..n}\}} \qquad \text{(T-Rcd)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \{\mathtt{l}_i\mathtt{:T}_i\ ^{i\in 1..n}\}}{\Gamma \vdash \mathtt{t}_1\mathtt{.l}_j : \mathtt{T}_j} \qquad \text{(T-Proj)}$$

Figure 11-7: Records

have a tuple in which all the fields to the left of field j have already been reduced to values, then that field can be evaluated one step, from $\mathtt{t}_j$ to $\mathtt{t'}_j$. Again, the use of metavariables enforces a left-to-right evaluation strategy.

11.8 Records

The generalization from n-ary tuples to labeled records is equally straightforward. We simply annotate each field $\mathtt{t}_i$ with a *label* $\mathtt{l}_i$ drawn from some predetermined set $\mathcal{L}$. For example, `{x=5}` and `{partno=5524,cost=30.27}` are both record values; their types are `{x:Nat}` and `{partno:Nat,cost:Float}`. We require that all the labels in a given record term or type be distinct.

The rules for records are given in Figure 11-7. The only one worth noting is E-ProjRcd, where we rely on a slightly informal convention. The rule is meant to be understood as follows: If $\{\mathtt{l}_i\mathtt{=v}_i\ ^{i\in 1..n}\}$ is a record and $\mathtt{l}_j$ is the label of its j^{th} field, then $\{\mathtt{l}_i\mathtt{=v}_i\ ^{i\in 1..n}\}\mathtt{.l}_j$ evaluates in one step to the j^{th} value, $\mathtt{v}_j$. This convention (and the similar one that we used in E-ProjTuple) could be eliminated by rephrasing the rule in a more explicit form; however, the cost in terms of readability would be fairly high.

11.8.1 Exercise [★ ↛]: Write E-ProjRcd more explicitly, for comparison. □

Note that the same "feature symbol," {}, appears in the list of features on the upper-left corner of the definitions of both tuples and products. Indeed, we can obtain tuples as a special case of records, simply by allowing the set of labels to include both alphabetic identifiers and natural numbers. Then when the i^{th} field of a record has the label `i`, we omit the label. For example, we regard `{Bool,Nat,Bool}` as an abbreviation for `{1:Bool,2:Nat,3:Bool}`. (This convention actually allows us to mix named and positional fields, writing `{a:Bool,Nat,c:Bool}` as an abbreviation for `{a:Bool,2:Nat,c:Bool}`, though this is probably not very useful in practice.) In fact, many languages keep tuples and records notationally distinct for a more pragmatic reason: they are implemented differently by the compiler.

Programming languages differ in their treatment of the order of record fields. In many languages, the order of fields in both record values and record types has no affect on meaning—i.e., the terms `{partno=5524,cost=30.27}` and `{cost=30.27,partno=5524}` have the same meaning and the same type, which may be written either `{partno:Nat,cost:Float}` or `{cost:Float, partno:Nat}`. Our presentation chooses the other alternative: `{partno=5524, cost=30.27}` and `{cost=30.27,partno=5524}` are *different* record values, with types `{partno:Nat,cost:Float}` and `{cost:Float, partno:Nat}`, respectively. In Chapter 15, we will adopt a more liberal view of ordering, introducing a subtype relation in which the types `{partno:Nat,cost:Float}` and `{cost:Float,partno:Nat}` are *equivalent*—each is a subtype of the other—so that terms of one type can be used in any context where the other type is expected. (In the presence of subtyping, the choice between ordered and unordered records has important effects on performance; these are discussed further in §15.6. Once we have decided on unordered records, though, the choice of whether to consider records as unordered from the beginning or to take the fields primitively as ordered and then give rules that allow the ordering to be ignored is purely a question of taste. We adopt the latter approach here because it allows us to discuss both variants.)

11.8.2 Exercise [★★★]: In our presentation of records, the projection operation is used to extract the fields of a record one at a time. Many high-level programming languages provide an alternative *pattern matching* syntax that extracts all the fields at the same time, allowing some programs to be expressed much more concisely. Patterns can also typically be nested, allowing parts to be extracted easily from complex nested data structures.

We can add a simple form of pattern matching to an untyped lambda calculus with records by adding a new syntactic category of *patterns,* plus one new case (for the pattern matching construct itself) to the syntax of terms. (See Figure 11-8.)

→ {} let p (untyped) *Extends 11-7 and 11-4*

New syntactic forms

$$\begin{array}{llll} \mathtt{p} & ::= & \mathtt{x} & \textit{variable pattern} \\ & & \{\mathtt{l}_i\mathtt{=p}_i{}^{\,i\in 1..n}\} & \textit{record pattern} \end{array}$$

$$\begin{array}{llll} \mathtt{t} & ::= & \ldots & \textit{terms:} \\ & & \mathtt{let\ p=t\ in\ t} & \textit{pattern binding} \end{array}$$

Matching rules:

$$match(\mathtt{x}, \mathtt{v}) = [\mathtt{x} \mapsto \mathtt{v}] \qquad \text{(M-Var)}$$

$$\frac{\text{for each } i \quad match(\mathtt{p}_i, \mathtt{v}_i) = \sigma_i}{match(\{\mathtt{l}_i\mathtt{=p}_i{}^{\,i\in 1..n}\}, \{\mathtt{l}_i\mathtt{=v}_i{}^{\,i\in 1..n}\}) = \sigma_1 \circ \cdots \circ \sigma_n} \qquad \text{(M-Rcd)}$$

New evaluation rules $\boxed{\mathtt{t} \longrightarrow \mathtt{t}'}$

$$\mathtt{let\ p=v_1\ in\ t_2} \longrightarrow match(\mathtt{p}, \mathtt{v}_1)\, \mathtt{t}_2 \qquad \text{(E-LetV)}$$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{let\ p=t_1\ in\ t_2} \longrightarrow \mathtt{let\ p=t_1'\ in\ t_2}} \qquad \text{(E-Let)}$$

Figure 11-8: (Untyped) record patterns

The computation rule for pattern matching generalizes the let-binding rule from Figure 11-4. It relies on an auxiliary "matching" function that, given a pattern p and a value v, either fails (indicating that v does not match p) or else yields a substitution that maps variables appearing in p to the corresponding parts of v. For example, *match*({x,y}, {5,true}) yields the substitution [x ↦ 5, y ↦ true] and *match*(x, {5,true}) yields [x ↦ {5,true}], while *match*({x}, {5,true}) fails. E-LetV uses *match* to calculate an appropriate substitution for the variables in p.

The *match* function itself is defined by a separate set of inference rules. The rule M-Var says that a variable pattern always succeeds, returning a substitution mapping the variable to the whole value being matched against. The rule M-Rcd says that, to match a record pattern $\{\mathtt{l}_i\mathtt{=p}_i{}^{\,i\in 1..n}\}$ against a record value $\{\mathtt{l}_i\mathtt{=v}_i{}^{\,i\in 1..n}\}$ (of the same length, with the same labels), we individually match each sub-pattern $\mathtt{p}_i$ against the corresponding value $\mathtt{v}_i$ to obtain a substitution σ_i, and build the final result substitution by composing all these substitutions. (We require that no variable should appear more than once in a pattern, so this composition of substitutions is just their union.)

Show how to add types to this system.

1. Give typing rules for the new constructs (making any changes to the syntax you feel are necessary in the process).

2. Sketch a proof of type preservation and progress for the whole calculus. (You do not need to show full proofs—just the statements of the required lemmas in the correct order.) □

$\rightarrow$ $+$ *Extends $\lambda_{\rightarrow}$ (9-1)*

New syntactic forms

$\texttt{t}$ ::=	...		*terms:*
	$\texttt{inl t}$		*tagging (left)*
	$\texttt{inr t}$		*tagging (right)*
	$\texttt{case t of inl x}\Rightarrow\texttt{t} \mid \texttt{inr x}\Rightarrow\texttt{t}$		*case*
$\texttt{v}$ ::=	...		*values:*
	$\texttt{inl v}$		*tagged value (left)*
	$\texttt{inr v}$		*tagged value (right)*
$\texttt{T}$ ::=	...		*types:*
	$\texttt{T+T}$		*sum type*

New evaluation rules $\boxed{\texttt{t} \longrightarrow \texttt{t}'}$

$$\begin{array}{l}\texttt{case (inl v}_0\texttt{)}\\ \texttt{of inl x}_1\Rightarrow\texttt{t}_1 \mid \texttt{inr x}_2\Rightarrow\texttt{t}_2\\ \qquad\longrightarrow [\texttt{x}_1 \mapsto \texttt{v}_0]\texttt{t}_1\end{array} \quad \text{(E-CaseInl)}$$

$$\begin{array}{l}\texttt{case (inr v}_0\texttt{)}\\ \texttt{of inl x}_1\Rightarrow\texttt{t}_1 \mid \texttt{inr x}_2\Rightarrow\texttt{t}_2\\ \qquad\longrightarrow [\texttt{x}_2 \mapsto \texttt{v}_0]\texttt{t}_2\end{array} \quad \text{(E-CaseInr)}$$

$$\frac{\texttt{t}_0 \longrightarrow \texttt{t}_0'}{\begin{array}{l}\texttt{case t}_0\texttt{ of inl x}_1\Rightarrow\texttt{t}_1 \mid \texttt{inr x}_2\Rightarrow\texttt{t}_2\\ \longrightarrow \texttt{case t}_0'\texttt{ of inl x}_1\Rightarrow\texttt{t}_1 \mid \texttt{inr x}_2\Rightarrow\texttt{t}_2\end{array}} \quad \text{(E-Case)}$$

$$\frac{\texttt{t}_1 \longrightarrow \texttt{t}_1'}{\texttt{inl t}_1 \longrightarrow \texttt{inl t}_1'} \quad \text{(E-Inl)}$$

$$\frac{\texttt{t}_1 \longrightarrow \texttt{t}_1'}{\texttt{inr t}_1 \longrightarrow \texttt{inr t}_1'} \quad \text{(E-Inr)}$$

New typing rules $\boxed{\Gamma \vdash \texttt{t} : \texttt{T}}$

$$\frac{\Gamma \vdash \texttt{t}_1 : \texttt{T}_1}{\Gamma \vdash \texttt{inl t}_1 : \texttt{T}_1\texttt{+T}_2} \quad \text{(T-Inl)}$$

$$\frac{\Gamma \vdash \texttt{t}_1 : \texttt{T}_2}{\Gamma \vdash \texttt{inr t}_1 : \texttt{T}_1\texttt{+T}_2} \quad \text{(T-Inr)}$$

$$\frac{\begin{array}{c}\Gamma \vdash \texttt{t}_0 : \texttt{T}_1\texttt{+T}_2\\ \Gamma, \texttt{x}_1\texttt{:T}_1 \vdash \texttt{t}_1 : \texttt{T} \qquad \Gamma, \texttt{x}_2\texttt{:T}_2 \vdash \texttt{t}_2 : \texttt{T}\end{array}}{\Gamma \vdash \texttt{case t}_0\texttt{ of inl x}_1\Rightarrow\texttt{t}_1 \mid \texttt{inr x}_2\Rightarrow\texttt{t}_2 : \texttt{T}} \quad \text{(T-Case)}$$

Figure 11-9: Sums

11.9 Sums

Many programs need to deal with *heterogeneous* collections of values. For example, a node in a binary tree can be either a leaf or an interior node with two children; similarly, a list cell can be either `nil` or a `cons` cell carrying a head and a tail,[5] a node of an abstract syntax tree in a compiler can represent a variable, an abstraction, an application, etc. The type-theoretic mechanism that supports this kind of programming is *variant types*.

Before introducing variants in full generality (in §11.10), let us consider the

5. These examples, like most real-world uses of variant types, also involve *recursive types*—the tail of a list is itself a list, etc. We will return to recursive types in Chapter 20.

simpler case of binary *sum types*. A sum type describes a set of values drawn from exactly two given types. For example, suppose we are using the types

```
PhysicalAddr = {firstlast:String, addr:String};
VirtualAddr  = {name:String, email:String};
```

to represent different sorts of address-book records. If we want to manipulate both sorts of records uniformly (e.g., if we want to make a list containing records of both kinds), we can introduce the sum type[6]

```
Addr = PhysicalAddr + VirtualAddr;
```

each of whose elements is either a `PhysicalAddr` or a `VirtualAddr`.

We create elements of this type by *tagging* elements of the component types `PhysicalAddr` and `VirtualAddr`. For example, if `pa` is a `PhysicalAddr`, then `inl pa` is an `Addr`. (The names of the tags `inl` and `inr` arise from thinking of them as functions

```
inl  :  PhysicalAddr → PhysicalAddr+VirtualAddr
inr  :  VirtualAddr → PhysicalAddr+VirtualAddr
```

that "inject" elements of `PhysicalAddr` or `VirtualAddr` into the left and right components of the sum type `Addr`. Note, though, that they are *not* treated as functions in our presentation.)

In general, the elements of a type T_1+T_2 consist of the elements of T_1, tagged with the token `inl`, plus the elements of T_2, tagged with `inr`.

To *use* elements of sum types, we introduce a `case` construct that allows us to distinguish whether a given value comes from the left or right branch of a sum. For example, we can extract a name from an `Addr` like this:

```
getName = λa:Addr.
  case a of
    inl x ⇒ x.firstlast
  | inr y ⇒ y.name;
```

When the parameter `a` is a `PhysicalAddr` tagged with `inl`, the case expression will take the first branch, binding the variable `x` to the `PhysicalAddr`; the body of the first branch then extracts the `firstlast` field from `x` and returns it. Similarly, if `a` is a `VirtualAddr` value tagged with `inr`, the second branch will be chosen and the `name` field of the `VirtualAddr` returned. Thus, the type of the whole `getName` function is `Addr→String`.

The foregoing intuitions are formalized in Figure 11-9. To the syntax of terms, we add the left and right injections and the `case` construct; to types,

6. The `fullsimple` implementation does not actually support the constructs for binary sums that we are describing here—just the more general case of variants described below.

we add the sum constructor. For evaluation, we add two "beta-reduction" rules for the `case` construct—one for the case where its first subterm has been reduced to a value v_0 tagged with `inl`, the other for a value v_0 tagged with `inr`; in each case, we select the appropriate body and substitute v_0 for the bound variable. The other evaluation rules perform evaluation in the first subterm of `case` and under the `inl` and `inr` tags.

The typing rules for tagging are straightforward: to show that `inl` t_1 has a sum type T_1+T_2, it suffices to show that t_1 belongs to the left summand, T_1, and similarly for `inr`. For the `case` construct, we first check that the first subterm has a sum type T_1+T_2, then check that the bodies t_1 and t_2 of the two branches have the same result type T, assuming that their bound variables x_1 and x_2 have types T_1 and T_2, respectively; the result of the whole `case` is then T. Following our conventions from previous definitions, Figure 11-9 does not state explicitly that the scopes of the variables x_1 and x_2 are the bodies t_1 and t_2 of the branches, but this fact can be read off from the way the contexts are extended in the typing rule T-CASE.

11.9.1 EXERCISE [★★]: Note the similarity between the typing rule for `case` and the rule for `if` in Figure 8-1: `if` can be regarded as a sort of degenerate form of `case` where no information is passed to the branches. Formalize this intuition by defining `true`, `false`, and `if` as derived forms using sums and `Unit`. □

Sums and Uniqueness of Types

Most of the properties of the typing relation of pure $\lambda_{\rightarrow}$ (cf. §9.3) extend to the system with sums, but one important one fails: the Uniqueness of Types theorem (9.3.3). The difficulty arises from the tagging constructs `inl` and `inr`. The typing rule T-INL, for example, says that, once we have shown that t_1 is an element of T_1, we can derive that `inl` t_1 is an element of T_1+T_2 for *any* type T_2. For example, we can derive both `inl 5 : Nat+Nat` and `inl 5 : Nat+Bool` (and infinitely many other types). The failure of uniqueness of types means that we cannot build a typechecking algorithm simply by "reading the rules from bottom to top," as we have done for all the features we have seen so far. At this point, we have various options:

1. We can complicate the typechecking algorithm so that it somehow "guesses" a value for T_2. Concretely, we hold T_2 indeterminate at this point and try to discover later what its value should have been. Such techniques will be explored in detail when we consider type reconstruction (Chapter 22).

2. We can refine the language of types to allow *all* possible values for T_2 to somehow be represented uniformly. This option will be explored when we discuss subtyping (Chapter 15).

$\to\ +$ *Extends* $\lambda_\to$ *(11-9)*

New syntactic forms

$$\begin{array}{llll} \mathtt{t} & ::= & \ldots & \textit{terms:} \\ & & \mathtt{inl}\ \mathtt{t}\ \boxed{\mathtt{as}\ \mathtt{T}} & \textit{tagging (left)} \\ & & \mathtt{inr}\ \mathtt{t}\ \boxed{\mathtt{as}\ \mathtt{T}} & \textit{tagging (right)} \\ \\ \mathtt{v} & ::= & \ldots & \textit{values:} \\ & & \mathtt{inl}\ \mathtt{v}\ \boxed{\mathtt{as}\ \mathtt{T}} & \textit{tagged value (left)} \\ & & \mathtt{inr}\ \mathtt{v}\ \boxed{\mathtt{as}\ \mathtt{T}} & \textit{tagged value (right)} \end{array}$$

New evaluation rules $\boxed{\mathtt{t} \longrightarrow \mathtt{t}'}$

$$\begin{array}{l} \mathtt{case}\ (\mathtt{inl}\ \mathtt{v}_0\ \mathtt{as}\ \mathtt{T}_0) \\ \mathtt{of}\ \mathtt{inl}\ \mathtt{x}_1 \Rightarrow \mathtt{t}_1 \mid \mathtt{inr}\ \mathtt{x}_2 \Rightarrow \mathtt{t}_2 \\ \qquad \longrightarrow [\mathtt{x}_1 \mapsto \mathtt{v}_0]\mathtt{t}_1 \end{array} \qquad \text{(E-CaseInl)}$$

$$\begin{array}{l} \mathtt{case}\ (\mathtt{inr}\ \mathtt{v}_0\ \mathtt{as}\ \mathtt{T}_0) \\ \mathtt{of}\ \mathtt{inl}\ \mathtt{x}_1 \Rightarrow \mathtt{t}_1 \mid \mathtt{inr}\ \mathtt{x}_2 \Rightarrow \mathtt{t}_2 \\ \qquad \longrightarrow [\mathtt{x}_2 \mapsto \mathtt{v}_0]\mathtt{t}_2 \end{array} \qquad \text{(E-CaseInr)}$$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{inl}\ \mathtt{t}_1\ \mathtt{as}\ \mathtt{T}_2 \longrightarrow \mathtt{inl}\ \mathtt{t}_1'\ \mathtt{as}\ \mathtt{T}_2} \qquad \text{(E-Inl)}$$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{inr}\ \mathtt{t}_1\ \mathtt{as}\ \mathtt{T}_2 \longrightarrow \mathtt{inr}\ \mathtt{t}_1'\ \mathtt{as}\ \mathtt{T}_2} \qquad \text{(E-Inr)}$$

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

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1}{\Gamma \vdash \mathtt{inl}\ \mathtt{t}_1\ \mathtt{as}\ \mathtt{T}_1\mathtt{+}\mathtt{T}_2 : \mathtt{T}_1\mathtt{+}\mathtt{T}_2} \qquad \text{(T-Inl)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_2}{\Gamma \vdash \mathtt{inr}\ \mathtt{t}_1\ \mathtt{as}\ \mathtt{T}_1\mathtt{+}\mathtt{T}_2 : \mathtt{T}_1\mathtt{+}\mathtt{T}_2} \qquad \text{(T-Inr)}$$

Figure 11-10: Sums (with unique typing)

3. We can demand that the programmer provide an explicit *annotation* to indicate which type $\mathtt{T}_2$ is intended. This alternative is the simplest—and it is not actually as impractical as it might at first appear, since, in full-scale language designs, these explicit annotations can often be "piggybacked" on other language constructs and so made essentially invisible (we'll come back to this point in the following section). We take this option for now.

Figure 11-10 shows the needed extensions, relative to Figure 11-9. Instead of writing just `inl t` or `inr t`, we write `inl t as T` or `inr t as T`, where `T` specifies the whole sum type to which we want the injected element to belong. The typing rules T-Inl and T-Inr use the declared sum type as the type of the injection, after checking that the injected term really belongs to the appropriate branch of the sum. (To avoid writing $\mathtt{T}_1\mathtt{+}\mathtt{T}_2$ repeatedly in the rules, the syntax rules allow any type `T` to appear as an annotation on an injection. The typing rules ensure that the annotation will always be a sum type, if the injection is well typed.) The syntax for type annotations is meant to suggest the ascription construct from §11.4: in effect these annotations can be viewed as syntactically required ascriptions.

$\rightarrow$ <> *Extends* $\lambda_{\rightarrow}$ *(9-1)*

New syntactic forms

$$
\begin{array}{lll}
\mathtt{t} ::= & \ldots & \textit{terms:} \\
 & \mathtt{<l=t>\ as\ T} & \textit{tagging} \\
 & \mathtt{case\ t\ of\ <l_i=x_i>\Rightarrow t_i}\ ^{i\in 1..n} & \textit{case} \\
 & & \\
\mathtt{T} ::= & \ldots & \textit{types:} \\
 & \mathtt{<l_i:T_i}\ ^{i\in 1..n}\mathtt{>} & \textit{type of variants}
\end{array}
$$

New evaluation rules $\boxed{\mathtt{t} \longrightarrow \mathtt{t'}}$

$$\mathtt{case\ (<l_j=v_j>\ as\ T)\ of\ <l_i=x_i>\Rightarrow t_i}\ ^{i\in 1..n} \longrightarrow [\mathtt{x_j} \mapsto \mathtt{v_j}]\mathtt{t_j} \qquad \text{(E-CaseVariant)}$$

$$\frac{\mathtt{t_0} \longrightarrow \mathtt{t_0'}}{\mathtt{case\ t_0\ of\ <l_i=x_i>\Rightarrow t_i}\ ^{i\in 1..n} \longrightarrow \mathtt{case\ t_0'\ of\ <l_i=x_i>\Rightarrow t_i}\ ^{i\in 1..n}} \qquad \text{(E-Case)}$$

$$\frac{\mathtt{t_i} \longrightarrow \mathtt{t_i'}}{\mathtt{<l_i=t_i>\ as\ T} \longrightarrow \mathtt{<l_i=t_i'>\ as\ T}} \qquad \text{(E-Variant)}$$

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

$$\frac{\Gamma \vdash \mathtt{t_j} : \mathtt{T_j}}{\Gamma \vdash \mathtt{<l_j=t_j>\ as\ <l_i:T_i}\ ^{i\in 1..n}\mathtt{>} : \mathtt{<l_i:T_i}\ ^{i\in 1..n}\mathtt{>}} \qquad \text{(T-Variant)}$$

$$\frac{\Gamma \vdash \mathtt{t_0} : \mathtt{<l_i:T_i}\ ^{i\in 1..n}\mathtt{>} \qquad \text{for each } i \quad \Gamma, \mathtt{x_i}:\mathtt{T_i} \vdash \mathtt{t_i} : \mathtt{T}}{\Gamma \vdash \mathtt{case\ t_0\ of\ <l_i=x_i>\Rightarrow t_i}\ ^{i\in 1..n} : \mathtt{T}} \qquad \text{(T-Case)}$$

Figure 11-11: Variants

11.10 Variants

Binary sums generalize to labeled *variants* just as products generalize to labeled records. Instead of $\mathtt{T_1+T_2}$, we write $\mathtt{<l_1:T_1,l_2:T_2>}$, where $\mathtt{l_1}$ and $\mathtt{l_2}$ are field labels. Instead of $\mathtt{inl\ t\ as\ T_1+T_2}$, we write $\mathtt{<l_1=t>\ as\ <l_1:T_1,l_2:T_2>}$. And instead of labeling the branches of the `case` with `inl` and `inr`, we use the same labels as the corresponding sum type. With these generalizations, the `getAddr` example from the previous section becomes:

```
  Addr = <physical:PhysicalAddr, virtual:VirtualAddr>;
  a = <physical=pa> as Addr;
▸ a : Addr

  getName = λa:Addr.
    case a of
      <physical=x> ⇒ x.firstlast
    | <virtual=y> ⇒ y.name;
▸ getName : Addr → String
```

The formal definition of variants is given in Figure 11-11. Note that, as with records in §11.8, the order of labels in a variant type is significant here.

Options

One very useful idiom involving variants is *optional values*. For example, an element of the type

```
OptionalNat = <none:Unit, some:Nat>;
```

is either the trivial `unit` value with the tag `none` or else a number with the tag `some`—in other words, the type `OptionalNat` is isomorphic to `Nat` extended with an additional distinguished value `none`. For example, the type

```
Table = Nat→OptionalNat;
```

represents finite mappings from numbers to numbers: the domain of such a mapping is the set of inputs for which the result is `<some=`*n*`>` for some *n*. The empty table

```
emptyTable = λn:Nat. <none=unit> as OptionalNat;
▸ emptyTable : Table
```

is a constant function that returns `none` for every input. The constructor

```
extendTable =
  λt:Table. λm:Nat. λv:Nat.
    λn:Nat.
      if equal n m then <some=v> as OptionalNat
      else t n;
▸ extendTable : Table → Nat → Nat → Table
```

takes a table and adds (or overwrites) an entry mapping the input `m` to the output `<some=v>`. (The `equal` function is defined in the solution to Exercise 11.11.1 on page 510.)

We can use the result that we get back from a `Table` lookup by wrapping a `case` around it. For example, if `t` is our table and we want to look up its entry for `5`, we might write

```
x = case t(5) of
      <none=u> ⇒ 999
    | <some=v> ⇒ v;
```

providing `999` as the default value of `x` in case `t` is undefined on `5`.

Many languages provide built-in support for options. OCaml, for example, predefines a type constructor `option`, and many functions in typical OCaml programs yield options. Also, the `null` value in languages like C, C++, and Java is actually an option in disguise. A variable of type `T` in these languages (where `T` is a "reference type"—i.e., something allocated in the heap) can actually contain either the special value `null` or else a pointer to a `T` value. That is, the type of such a variable is really `Ref(Option(T))`, where `Option(T) = <none:Unit,some:T>`. Chapter 13 discusses the `Ref` constructor in detail.

Enumerations

Two "degenerate cases" of variant types are useful enough to deserve special mention: enumerated types and single-field variants.

An *enumerated type* (or *enumeration*) is a variant type in which the field type associated with each label is `Unit`. For example, a type representing the days of the working week might be defined as:

```
Weekday = <monday:Unit, tuesday:Unit, wednesday:Unit,
           thursday:Unit, friday:Unit>;
```

The elements of this type are terms like `<monday=unit> as Weekday`. Indeed, since the type `Unit` has only `unit` as a member, the type `Weekday` is inhabited by precisely five values, corresponding one-for-one with the days of the week. The `case` construct can be used to define computations on enumerations.

```
nextBusinessDay = λw:Weekday.
  case w of <monday=x>    ⇒ <tuesday=unit> as Weekday
          | <tuesday=x>   ⇒ <wednesday=unit> as Weekday
          | <wednesday=x> ⇒ <thursday=unit> as Weekday
          | <thursday=x>  ⇒ <friday=unit> as Weekday
          | <friday=x>    ⇒ <monday=unit> as Weekday;
```

Obviously, the concrete syntax we are using here is not well tuned for making such programs easy to write or read. Some languages (beginning with Pascal) provide special syntax for declaring and using enumerations. Others—such as ML, cf. page 141—make enumerations a special case of the variants.

Single-Field Variants

The other interesting special case is variant types with just a single label `l`:

```
V = <l:T>;
```

Such a type might not seem very useful at first glance: after all, the elements of `V` will be in one-to-one correspondence with the elements of the field type `T`, since every member of `V` has precisely the form `<l=t>` for some `t : T`. What's important, though, is that the usual operations on `T` *cannot* be applied to elements of `V` without first unpackaging them: a `V` cannot be accidentally mistaken for a `T`.

For example, suppose we are writing a program to do financial calculations in multiple currencies. Such a program might include functions for converting between dollars and euros. If both are represented as `Float`s, then these functions might look like this:

```
dollars2euros = λd:Float. timesfloat d 1.1325;
▸ dollars2euros : Float → Float

euros2dollars = λe:Float. timesfloat e 0.883;
▸ euros2dollars : Float → Float
```

(where `timesfloat : Float→Float→Float` multiplies floating-point numbers). If we then start with a dollar amount

```
mybankbalance = 39.50;
```

we can convert it to euros and then back to dollars like this:

```
euros2dollars (dollars2euros mybankbalance);
▸ 39.49990125 : Float
```

All this makes perfect sense. But we can just as easily perform manipulations that make no sense at all. For example, we can convert my bank balance to euros twice:

```
dollars2euros (dollars2euros mybankbalance);
▸ 50.660971875 : Float
```

Since all our amounts are represented simply as floats, there is no way that the type system can help prevent this sort of nonsense. However, if we define dollars and euros as different variant types (whose underlying representations are floats)

```
DollarAmount = <dollars:Float>;
EuroAmount = <euros:Float>;
```

then we can define safe versions of the conversion functions that will only accept amounts in the correct currency:

```
dollars2euros =
  λd:DollarAmount.
    case d of <dollars=x> ⇒
      <euros = timesfloat x 1.1325> as EuroAmount;
▸ dollars2euros : DollarAmount → EuroAmount
```

```
euros2dollars =
  λe:EuroAmount.
    case e of <euros=x> ⇒
      <dollars = timesfloat x 0.883> as DollarAmount;
```

```
▸ euros2dollars : EuroAmount → DollarAmount
```

Now the typechecker can track the currencies used in our calculations and remind us how to interpret the final results:

```
mybankbalance = <dollars=39.50> as DollarAmount;
euros2dollars (dollars2euros mybankbalance);
```

```
▸ <dollars=39.49990125> as DollarAmount : DollarAmount
```

Moreover, if we write a nonsensical double-conversion, the types will fail to match and our program will (correctly) be rejected:

```
dollars2euros (dollars2euros mybankbalance);
```

```
▸ Error: parameter type mismatch
```

Variants vs. Datatypes

A variant type T of the form $\texttt{<}\mathtt{l}_i\texttt{:}\mathtt{T}_i\ ^{i\in 1..n}\texttt{>}$ is roughly analogous to the ML datatype defined by:[7]

$$
\begin{array}{l}
\texttt{type T = } \mathtt{l}_1 \texttt{ of } \mathtt{T}_1 \\
\qquad\quad \texttt{| } \mathtt{l}_2 \texttt{ of } \mathtt{T}_2 \\
\qquad\quad \texttt{| ...} \\
\qquad\quad \texttt{| } \mathtt{l}_n \texttt{ of } \mathtt{T}_n
\end{array}
$$

But there are several differences worth noticing.

1. One trivial but potentially confusing point is that the capitalization conventions for identifiers that we are assuming here are different from those of OCaml. In OCaml, types must begin with lowercase letters and datatype constructors (labels, in our terminology) with capital letters, so, strictly speaking, the datatype declaration above should be written like this:

   $\texttt{type t = } \mathtt{L}_1 \texttt{ of } \mathtt{t}_1 \texttt{ | ... | } \mathtt{L}_n \texttt{ of } \mathtt{t}_n$

7. This section uses OCaml's concrete syntax for datatypes, for consistency with implementation chapters elsewhere in the book, but they originated in early dialects of ML and can be found, in essentially the same form, in Standard ML as well as in ML relatives such as Haskell. Datatypes and pattern matching are arguably one of the most useful advantages of these languages for day to day programming.

To avoid confusion between terms `t` and types `T`, we'll ignore OCaml's conventions for the rest of this discussion and use ours instead.

2. The most interesting difference is that OCaml does *not* require a type annotation when a constructor $\mathtt{l}_i$ is used to inject an element of $\mathtt{T}_i$ into the datatype `T`: we simply write $\mathtt{l}_i\mathtt{(t)}$. The way OCaml gets away with this (and retains unique typing) is that the datatype `T` must be *declared* before it can be used. Moreover, the labels in `T` cannot be used by any other datatype declared in the same scope. So, when the typechecker sees $\mathtt{l}_i\mathtt{(t)}$, it knows that the annotation can only be `T`. In effect, the annotation is "hidden" in the label itself.

   This trick eliminates a lot of silly annotations, but it does lead to a certain amount of grumbling among users, since it means that labels cannot be shared between different datatypes—at least, not within the same module. In Chapter 15 we will see another way of omitting annotations that avoids this drawback.

3. Another convenient trick used by OCaml is that, when the type associated with a label in a datatype definition is just `Unit`, it can be omitted altogether. This permits enumerations to be defined by writing

```
type Weekday = monday | tuesday | wednesday | thursday | friday
```

   for example, rather than:

```
type Weekday = monday of Unit
             | tuesday of Unit
             | wednesday of Unit
             | thursday of Unit
             | friday of Unit
```

   Similarly, the label `monday` all by itself (rather than `monday` applied to the trivial value `unit`) is considered to be a value of type `Weekday`.

4. Finally, OCaml datatypes actually bundle variant types together with several additional features that we will be examining, individually, in later chapters.

   - A datatype definition may be *recursive*—i.e., the type being defined is allowed to appear in the body of the definition. For example, in the standard definition of lists of `Nat`s, the value tagged with `cons` is a pair whose second element is a `NatList`.

```
type NatList = nil
             | cons of Nat * NatList
```

- An OCaml datatype can be *[*parametric data type]parameterized*parametric!data type* on a type variable, as in the general definition of the List datatype:

```
type 'a List = nil
             | cons of 'a * 'a List
```

  Type-theoretically, List can be viewed as a kind of function—called a *type operator*—that maps each choice of ′a to a concrete datatype… Nat to NatList, etc. Type operators are the subject of Chapter 29.

Variants as Disjoint Unions

Sum and variant types are sometimes called *disjoint unions*. The type T_1+T_2 is a "union" of T_1 and T_2 in the sense that its elements include all the elements from T_1 and T_2. This union is disjoint because the sets of elements of T_1 or T_2 are tagged with inl or inr, respectively, before they are combined, so that it is always clear whether a given element of the union comes from T_1 or T_2. The phrase *union type* is also used to refer to *untagged* (non-disjoint) union types, described in §15.7.

Type Dynamic

Even in statically typed languages, there is often the need to deal with data whose type cannot be determined at compile time. This occurs in particular when the lifetime of the data spans multiple machines or many runs of the compiler—when, for example, the data is stored in an external file system or database, or communicated across a network. To handle such situations safely, many languages offer facilities for inspecting the types of values at run time.

One attractive way of accomplishing this is to add a type Dynamic whose values are pairs of a value v and a type tag T where v has type T. Instances of Dynamic are built with an explicit tagging construct and inspected with a type safe typecase construct. In effect, Dynamic can be thought of as an infinite disjoint union, whose labels are types. See Gordon (circa 1980), Mycroft (1983), Abadi, Cardelli, Pierce, and Plotkin (1991b), Leroy and Mauny (1991), Abadi, Cardelli, Pierce, and Rémy (1995), and Henglein (1994).

11.11 General Recursion

Another facility found in most programming languages is the ability to define recursive functions. We have seen (Chapter 5, p. 65) that, in the untyped lambda-calculus, such functions can be defined with the aid of the `fix` combinator.

Recursive functions can be defined in a typed setting in a similar way. For example, here is a function `iseven` that returns `true` when called with an even argument and `false` otherwise:

```
  ff = λie:Nat→Bool.
         λx:Nat.
           if iszero x then true
           else if iszero (pred x) then false
           else ie (pred (pred x));

▸ ff : (Nat→Bool) → Nat → Bool

  iseven = fix ff;

▸ iseven : Nat → Bool

  iseven 7;

▸ false : Bool
```

The intuition is that the higher-order function `ff` passed to `fix` is a *generator* for the `iseven` function: if `ff` is applied to a function `ie` that approximates the desired behavior of `iseven` up to some number n (that is, a function that returns correct results on inputs less than or equal to n), then it returns a better approximation to `iseven`—a function that returns correct results for inputs up to $n + 2$. Applying `fix` to this generator returns its fixed point—a function that gives the desired behavior for all inputs n.

However, there is one important difference from the untyped setting: `fix` itself cannot be defined in the simply typed lambda-calculus. Indeed, we will see in Chapter 12 that *no* expression that can lead to non-terminating computations can be typed using only simple types.[8] So, instead of defining `fix` as a term in the language, we simply add it as a new primitive, with evaluation rules mimicking the behavior of the untyped `fix` combinator and a typing rule that captures its intended uses. These rules are written out in Figure 11-12. (The `letrec` abbreviation will be discussed below.)

The simply typed lambda-calculus with numbers and `fix` has long been a favorite experimental subject for programming language researchers, since it is the simplest language in which a range of subtle semantic phenomena such as *full abstraction* (Plotkin, 1977, Hyland and Ong, 2000, Abramsky, Jagadeesan, and Malacaria, 2000) arise. It is often called *PCF*.

8. In later chapters—Chapter 13 and Chapter 20—we will see some extensions of simple types that recover the power to define `fix` within the system.

→ fix *Extends $\lambda_{\rightarrow}$ (9-1)*

New syntactic forms

$$\begin{array}{lll} \texttt{t} ::= & \dots & \textit{terms:} \\ & \texttt{fix t} & \textit{fixed point of } \texttt{t} \end{array}$$

New evaluation rules $\boxed{\texttt{t} \longrightarrow \texttt{t}'}$

$$\begin{array}{l}\texttt{fix } (\lambda \texttt{x:T}_1\texttt{.t}_2) \\ \longrightarrow [\texttt{x} \mapsto (\texttt{fix } (\lambda \texttt{x:T}_1\texttt{.t}_2))]\texttt{t}_2\end{array} \quad \text{(E-FixBeta)}$$

$$\frac{\texttt{t}_1 \longrightarrow \texttt{t}_1'}{\texttt{fix t}_1 \longrightarrow \texttt{fix t}_1'} \quad \text{(E-Fix)}$$

New typing rules $\boxed{\Gamma \vdash \texttt{t : T}}$

$$\frac{\Gamma \vdash \texttt{t}_1 : \texttt{T}_1 \to \texttt{T}_1}{\Gamma \vdash \texttt{fix t}_1 : \texttt{T}_1} \quad \text{(T-Fix)}$$

New derived forms

$$\begin{array}{l}\texttt{letrec x:T}_1\texttt{=t}_1 \texttt{ in t}_2 \\ \stackrel{\text{def}}{=} \texttt{let x = fix } (\lambda \texttt{x:T}_1\texttt{.t}_1) \texttt{ in t}_2\end{array}$$

Figure 11-12: General recursion

11.11.1 Exercise [★★]: Define `equal`, `plus`, `times`, and `factorial` using `fix`. □

The `fix` construct is typically used to build functions (as fixed points of functions from functions to functions), but it is worth noticing that the type `T` in rule T-Fix is not restricted to function types. This extra power is sometimes handy. For example, it allows us to define a *record* of mutually recursive functions as the fixed point of a function on records (of functions). The following implementation of `iseven` uses an auxiliary function `isodd`; the two functions are defined as fields of a record, where the definition of this record is abstracted on a record `ieio` whose components are used to make recursive calls from the bodies of the `iseven` and `isodd` fields.

```
ff = λieio:{iseven:Nat→Bool, isodd:Nat→Bool}.
       {iseven = λx:Nat.
                    if iszero x then true
                    else ieio.isodd (pred x),
        isodd = λx:Nat.
                    if iszero x then false
                    else ieio.iseven (pred x)};
```

```
▸ ff : {iseven:Nat→Bool,isodd:Nat→Bool} →
       {iseven:Nat→Bool, isodd:Nat→Bool}
```

Forming the fixed point of the function `ff` gives us a record of two functions

```
r = fix ff;
```

```
▸ r : {iseven:Nat→Bool, isodd:Nat→Bool}
```

and projecting the first of these gives us the `iseven` function itself:

```
iseven = r.iseven;
```

```
▸ iseven : Nat → Bool
```

```
iseven 7;
```

```
▸ false : Bool
```

The ability to form the fixed point of a function of type `T→T` for any `T` has some surprising consequences. In particular, it implies that *every* type is inhabited by some term. To see this, observe that, for every type `T`, we can define a function $\mathtt{diverge}_T$ as follows:

```
divergeT = λ_:Unit. fix (λx:T.x);
```

```
▸ divergeT : Unit → T
```

Whenever $\mathtt{diverge}_T$ is applied to a `unit` argument, we get a non-terminating evaluation sequence in which E-FixBeta is applied over and over, always yielding the same term. That is, for every type `T`, the term $\mathtt{diverge}_T$ `unit` is an *undefined element* of `T`.

One final refinement that we may consider is introducing more convenient concrete syntax for the common case where what we want to do is to bind a variable to the result of a recursive definition. In most high-level languages, the first definition of `iseven` above would be written something like this:

```
letrec iseven : Nat→Bool =
  λx:Nat.
    if iszero x then true
    else if iszero (pred x) then false
    else iseven (pred (pred x))
in
  iseven 7;
```

```
▸ false : Bool
```

The recursive binding construct `letrec` is easily defined as a derived form:

$$\mathtt{letrec}\ \mathtt{x}\mathtt{:}\mathtt{T}_1\mathtt{=}\mathtt{t}_1\ \mathtt{in}\ \mathtt{t}_2 \quad \stackrel{\text{def}}{=} \quad \mathtt{let}\ \mathtt{x} = \mathtt{fix}\ (\lambda\mathtt{x}\mathtt{:}\mathtt{T}_1\mathtt{.}\mathtt{t}_1)\ \mathtt{in}\ \mathtt{t}_2$$

11.11.2 Exercise [★]: Rewrite your definitions of `plus`, `times`, and `factorial` from Exercise 11.11.1 using `letrec` instead of `fix`. □

Further information on fixed point operators can be found in Klop (1980) and Winskel (1993).

11.12 Lists

The typing features we have seen can be classified into *base types* like `Bool` and `Unit`, and *type constructors* like $\to$ and $\times$ that build new types from old ones. Another useful type constructor is `List`. For every type `T`, the type `List T` describes finite-length lists whose elements are drawn from `T`.

Figure 11-13 summarizes the syntax, semantics, and typing rules for lists. Except for syntactic differences (`List T` instead of `T list`, etc.) and the explicit type annotations on all the syntactic forms in our presentation,[9] these lists are essentially identical to those found in ML and other functional languages. The empty list (with elements of type `T`) is written `nil[T]`. The list formed by adding a new element t_1 (of type `T`) to the front of a list t_2 is written `cons[T]` t_1 t_2. The head and tail of a list `t` are written `head[T] t` and `tail[T] t`. The boolean predicate `isnil[T] t` yields `true` iff `t` is empty.[10]

11.12.1 Exercise [★★★]: Verify that the progress and preservation theorems hold for the simply typed lambda-calculus with booleans and lists. □

11.12.2 Exercise [★★]: The presentation of lists here includes many type annotations that are not really needed, in the sense that the typing rules can easily derive the annotations from context. Can *all* the type annotations be deleted? □

9. Most of these explicit annotations could actually be omitted (Exercise [★, ↛]: which cannot); they are retained here to ease comparison with the encoding of lists in §23.4.

10. We adopt the "`head`/`tail`/`isnil` presentation" of lists here for simplicity. From the perspective of language design, it is arguably better to treat lists as a datatype and use `case` expressions for destructing them, since more programming errors can be caught as type errors this way.

$\rightarrow \mathbb{B}$ `List` *Extends $\lambda_\rightarrow$ (9-1) with booleans (8-1)*

New syntactic forms

$\mathtt{t}$	::=	...	*terms:*
		$\mathtt{nil[T]}$	*empty list*
		$\mathtt{cons[T]\ t\ t}$	*list constructor*
		$\mathtt{isnil[T]\ t}$	*test for empty list*
		$\mathtt{head[T]\ t}$	*head of a list*
		$\mathtt{tail[T]\ t}$	*tail of a list*
$\mathtt{v}$	::=	...	*values:*
		$\mathtt{nil[T]}$	*empty list*
		$\mathtt{cons[T]\ v\ v}$	*list constructor*
$\mathtt{T}$	::=	...	*types:*
		$\mathtt{List\ T}$	*type of lists*

New evaluation rules $\boxed{\mathtt{t} \longrightarrow \mathtt{t}'}$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{cons[T]}\ \mathtt{t}_1\ \mathtt{t}_2 \longrightarrow \mathtt{cons[T]}\ \mathtt{t}_1'\ \mathtt{t}_2} \qquad \text{(E-Cons1)}$$

$$\frac{\mathtt{t}_2 \longrightarrow \mathtt{t}_2'}{\mathtt{cons[T]}\ \mathtt{v}_1\ \mathtt{t}_2 \longrightarrow \mathtt{cons[T]}\ \mathtt{v}_1\ \mathtt{t}_2'} \qquad \text{(E-Cons2)}$$

$$\mathtt{isnil[S]\ (nil[T])} \longrightarrow \mathtt{true} \qquad \text{(E-IsnilNil)}$$

$$\mathtt{isnil[S]\ (cons[T]}\ \mathtt{v}_1\ \mathtt{v}_2) \longrightarrow \mathtt{false} \qquad \text{(E-IsnilCons)}$$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{isnil[T]}\ \mathtt{t}_1 \longrightarrow \mathtt{isnil[T]}\ \mathtt{t}_1'} \qquad \text{(E-Isnil)}$$

$$\mathtt{head[S]\ (cons[T]}\ \mathtt{v}_1\ \mathtt{v}_2) \longrightarrow \mathtt{v}_1 \qquad \text{(E-HeadCons)}$$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{head[T]}\ \mathtt{t}_1 \longrightarrow \mathtt{head[T]}\ \mathtt{t}_1'} \qquad \text{(E-Head)}$$

$$\mathtt{tail[S]\ (cons[T]}\ \mathtt{v}_1\ \mathtt{v}_2) \longrightarrow \mathtt{v}_2 \qquad \text{(E-TailCons)}$$

$$\frac{\mathtt{t}_1 \longrightarrow \mathtt{t}_1'}{\mathtt{tail[T]}\ \mathtt{t}_1 \longrightarrow \mathtt{tail[T]}\ \mathtt{t}_1'} \qquad \text{(E-Tail)}$$

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

$$\Gamma \vdash \mathtt{nil}\ [\mathtt{T}_1] : \mathtt{List}\ \mathtt{T}_1 \qquad \text{(T-Nil)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{T}_1 \qquad \Gamma \vdash \mathtt{t}_2 : \mathtt{List}\ \mathtt{T}_1}{\Gamma \vdash \mathtt{cons}[\mathtt{T}_1]\ \mathtt{t}_1\ \mathtt{t}_2 : \mathtt{List}\ \mathtt{T}_1} \qquad \text{(T-Cons)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{List}\ \mathtt{T}_{11}}{\Gamma \vdash \mathtt{isnil}[\mathtt{T}_{11}]\ \mathtt{t}_1 : \mathtt{Bool}} \qquad \text{(T-Isnil)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{List}\ \mathtt{T}_{11}}{\Gamma \vdash \mathtt{head}[\mathtt{T}_{11}]\ \mathtt{t}_1 : \mathtt{T}_{11}} \qquad \text{(T-Head)}$$

$$\frac{\Gamma \vdash \mathtt{t}_1 : \mathtt{List}\ \mathtt{T}_{11}}{\Gamma \vdash \mathtt{tail}[\mathtt{T}_{11}]\ \mathtt{t}_1 : \mathtt{List}\ \mathtt{T}_{11}} \qquad \text{(T-Tail)}$$

Figure 11-13: Lists