142 11 Simple Extensions

• 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:

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. 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.

144 11 Simple Extensions

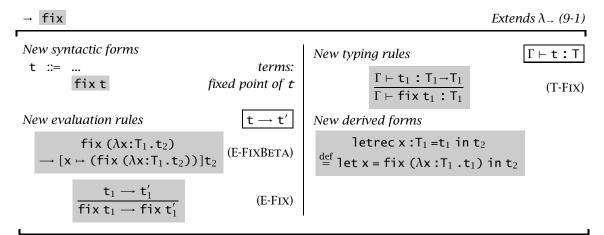


Figure 11-12: General recursion

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.

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;
riseven : Nat → Bool
iseven 7;
rfalse : Bool
```

The ability to form the fixed point of a function of type $T \rightarrow T$ 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 $diverge_T$ as follows:

```
diverge_T = \lambda_-:Unit. fix (\lambda x:T.x);
 diverge_T : Unit \rightarrow T
```

Whenever $diverge_T$ 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 $diverge_T$ 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:

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

```
letrec x:T_1=t_1 in t_2 \stackrel{\text{def}}{=} \text{let x} = \text{fix } (\lambda x:T_1.t_1) \text{ in } 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).