7 Data Types

7.2.4 The ML Type System

EXAMPLE 7.96
Fibonacci function in ML

The following is an ML version of the tail-recursive Fibonacci function introduced in Section 6.6.1:

```
1.  fun fib (n) =
2.      let fun fib_helper (f1, f2, i) =
3.          if i = n then f2
4.          else fib_helper (f2, f1+f2, i+1)
5.      in
6.          fib_helper (0, 1, 0)
7.      end;
```

The `let` construct introduces a nested scope: function `fib_helper` is nested inside `fib`. The body of `fib` is the expression `fib_helper (0, 1, 0)`. The body of `fib_helper` is an `if`... `then` ... `else` expression; it evaluates to either `f2` or to `fib_helper (f2, f1+f2, i+1)`, depending on whether the third argument to `fib_helper` is `n` or not.

Given this function definition, an ML compiler will reason roughly as follows: Parameter `i` of `fib_helper` must have type `int`, because it is added to 1 at line 4. Similarly, parameter `n` of `fib` must have type `int`, because it is compared to `i` at line 3. In the specific call to `fib_helper` at line 6, the types of all three arguments are `int`, so in this context at least, the types of `f1` and `f2` are `int`. Moreover the type of `i` is consistent with the earlier inference, namely `int`, and the types of the arguments to the recursive call at line 4 are similarly consistent. Since `fib_helper` returns `f2` at line 3, the result of the call at line 6 will be an `int`. Since `fib` immediately returns this result as its own result, the return type of `fib` is `int`.

EXAMPLE 7.97
Expression types

Because ML is a functional language, every construct in ML is an expression. The ML type system infers a type for every object and every expression. Because functions are first-class values, they too have types. The type of `fib` above is `int -> int`; that is, a function from integers to integers. The type of `fib_helper` is `int * int * int -> int`; that is, a function from integer triples to integers. In denotational terms, `int * int * int` is a three-way Cartesian product. ■

Type correctness in ML amounts to what we might call type *consistency*: a program is type correct if the type checking algorithm can reason out a unique type for every expression, with no contradictions and no ambiguous occurrences of overloaded names. (For built-in arithmetic and comparison operators, ML assumes that arguments are integers if it cannot determine otherwise. Haskell is a bit more general: it allows the arguments to be of any type that supports the required operations.) If the programmer uses an object inconsistently, the compiler will complain. In a program containing the following expressions,

EXAMPLE 7.98
Type inconsistency

```
fun circum (r) = r * 2.0 * 3.14159;
...
circum (7)
```

the compiler will infer that `circum`'s parameter is of type `real`, and will then complain when we attempt to pass an integer argument. ■

Though usually compiled instead of interpreted, ML is intended for interactive use. The programmer interacts with the ML system "on-line," giving it input a line at a time. The system compiles this input incrementally, binding machine language fragments to function names, and producing any appropriate compile-time error messages. This style of interaction blurs the traditional distinction between interpretation and compilation, but has more of the flavor of the latter. The language implementation remains active during program execution, but it does not actively manage the execution of program fragments: it *transfers* control to them and waits for them to return.

In comparison to languages in which programmers must declare all types explicitly, ML's type inference system has the advantage of brevity and convenience for interactive use. More important, it provides a powerful form of implicit parametric polymorphism more or less for free. While all uses of objects in an ML program must be consistent, they do not have to be completely specified:

EXAMPLE 7.99
Polymorphic functions

```
fun compare (x, p, q) =
    if x = p then
        if x = q then "both"
        else "first"
    else
        if x = q then "second"
        else "neither";
```

Here the equality test (`=`) is a built-in polymorphic function of type `'a * 'a -> bool`; that is, a function that takes a pair of arguments of the same type and produces a Boolean result. The token `'a` is called a *type variable*; it stands for any type, and takes, implicitly, the role of an explicit type parameter in a generic construct (Sections 8.4 and 9.4.4). Every instance of `'a` in a given call to `=` must represent the same type, but instances of `'a` in different calls can be different. Starting with the type of `=`, an ML compiler can reason that the type of compare is 'a * 'a * 'a -> string. Thus compare is polymorphic; it does not depend on the types of x, p, and q, so long as they are all the same. The key point to observe is that the programmer did not have to do anything special to make compare polymorphic: polymorphism is a natural consequence of ML-style type inference.

Type Checking

An ML compiler verifies type consistency with respect to a well-defined set of constraints. Specifically,

- All occurrences of the same identifier (subject to scope rules) must have the same type.
- In an if... then ... else expression, the condition must be of type bool, and the then and else clauses must have the same type.
- A programmer-defined function has type 'a -> 'b, where 'a is the type of the function's parameter, and 'b is the type of its result. As we shall see shortly, all functions have a single parameter. One obtains the *appearance* of multiple parameters by passing a *tuple* as argument.
- When a function is applied (called), the type of the argument that is passed must be the same as the type of the parameter in the function's definition. The type of the application (call) is the same as the type of the result in the function's definition.

In any case where two types *A* and *B* must be "the same," the ML compiler must *unify* what it knows about *A* and *B* to produce a (potentially more detailed) description of their common type. For example, if the compiler has determined that E1 is an expression of type 'a * int (that is, a two-element tuple whose second element is known to be an integer), and that E2 is an expression of type string * 'b, then in the expression if x then E1 else E2, it can infer that 'a is string and 'b is int. Thus x is of type bool, and E1 and E2 are of type string * int.

Lists

As in most functional languages, ML programmers tend to make heavy use of lists. In languages like Lisp and Scheme, which are dynamically typed (and also

DESIGN & IMPLEMENTATION

Unification

Unification is a powerful technique. In addition to its role in type inference (which also arises in the templates [generics] of C++), unification plays a central role in the computational model of Prolog and other logic languages. We will consider this latter role in Section 11.1. In the general case the cost of unifying the types of two expressions can be exponential [Mai90], but the pathological cases tend not to arise in practice.

EXAMPLE 7.100
Polymorphic list operators

implicitly polymorphic), lists may contain objects of arbitrary types. In ML, all elements of a given list must have the same type, but—and this is important—functions that manipulate lists without performing operations on their members can take any kind of list as argument:

```
fun append (l1, l2) =
    if l1 = nil then l2
    else hd (l1) :: append (tl (l1), l2);
fun member (x, l) =
    if l = nil then false
    else if x = hd (l) then true
    else member (x, tl (l));
```

Here append is of type `'a list * 'a list -> 'a list`; `member` is of type `'a * 'a list -> bool`. The reserved word `nil` represents the empty list. The built-in `::` constructor is analogous to `cons` in Lisp. It takes an element and a list and tacks the former onto the beginning of the latter; its type is `'a * 'a list -> 'a list`. The `hd` and `tl` functions are analogous to `car` and `cdr` in Lisp. They return the head and the remainder, respectively, of a list created by `::`. ■

EXAMPLE 7.101
List notation

Lists are most often written in ML using "square bracket" notation. The token `[]` is the same as `nil`. `[A, B, C]` is the same as `A :: B :: C :: nil`. Only "proper" lists—those that end with nil—can be represented with square brackets. The append function defined above is actually provided in ML as a built-in infix constructor, `@`. The expression `[a, b, c] @ [d, e, f, g]` evaluates to `[a, b, c, d, e, f, g]`. ■

Since ML lists are homogeneous (all elements have the same type), one might wonder about the type of `nil`. To allow it to take on the type of any list, `nil` is defined not as an object, but as a built-in polymorphic function of type `unit -> 'a list`. The built-in type `unit` is simply a placeholder, analogous to `void` in C. A function that takes no arguments is said to have a parameter of type `unit`. A function that is executed only for its side effects (ML is not purely functional) is said to return a result of type `unit`.

Overloading

We have already seen that the equality test (`=`) is a built-in polymorphic operator. The same is not true of ordering tests (`<`, `<=`, `>=`, `>`) or arithmetic operators (`+`, `-`, `*`). The equality test can be defined as a polymorphic function because it accepts arguments of *any* type. The relations and arithmetic operators work only on certain types. To avoid limiting them to a single type of argument (e.g., integers), ML defines them as overloaded names for a collection of built-in functions, each of which operates on objects of a different type (integers, floating-point numbers, strings, etc.). The programmer can define additional such functions for new types.

EXAMPLE 7.102
Resolving ambiguity with explicit types

Unfortunately, overloading sometimes interferes with type inference—there may not be enough information in an otherwise valid program to resolve which function is named by an overloaded operator:

```
fun square (x) = x * x;
```

Here the ML compiler cannot tell whether * is meant to refer to integer or floating-point multiplication; it assumes the former by default. If this is not what the programmer wants, the alternative must be specified explicitly:

```
fun square (x : real) = x * x;
```

In addition to allowing the resolution of overloaded symbols, explicit type declarations serve as "verified documentation" in ML programs. ML programmers often declare types for variables even when they aren't required, because the declarations make a program easier to read and understand. Readability could also be enhanced by comments, of course, but programmer-specified types have a very important advantage: the compiler understands their meaning, and ensures that all uses of an object are consistent with its declared type.

EXAMPLE 7.103
Type classes in Haskell

Haskell adopts a more general approach to overloading known as *type classes.* The equality functions, for example, are declared (but not defined) in a predefined class Eq:

```
class Eq a where
    (==), (/=) :: a -> a -> Bool    -- type signature
    x /= y     =  not (x == y)      -- default implementation of /=
```

Here a (written without a tick mark) is a type parameter: both == and /= take two parameters of the same type and return a Boolean result. Any value that is passed to one of these functions will be inferred to be of some type in class Eq. Any value that is passed to one of the ordering functions (<, <=, >=, >) will similarly be inferred to be of some type in class Ord:

```
class (Eq a) => Ord a where
    (<), (<=), (>=), (>) :: a -> a -> Bool
    max, min             :: a -> a -> a
```

The "(Eq a) =>" in the header of this declaration indicates that Ord is an extension of Eq; every type in class Ord must support the operations of class Eq as well. There is a strong analogy between type classes and the *interfaces* of languages with mix-in inheritance (Section ◎9.5.4).

Pattern Matching

In our discussion so far, we have been "glossing over" another key feature of ML and its types: namely, pattern matching. One of the simplest forms of pattern matching occurs in functions of more than one parameter. Strictly speaking, such functions do not exist. Every function in ML takes a *single* argument, but this argument may be a *tuple.* A tuple resembles the records (structures) of many other languages, except that its members are identified by position, rather than by name. As an example, the function compare defined above takes a three-element tuple as argument. All of the following are valid:

EXAMPLE 7.104
Pattern matching of argument tuples

```
compare (1, 2, 3);
let val t = ("larry", "moe", "curly") in compare (t) end;
let val d = (2, 3) in
    let val (a, b) = d in
        compare (1, a, b)
    end
end;
```

Here pattern matching occurs not only between the parameters and arguments of the call to `compare`, but also between the left- and right-hand sides of the `val` construct. (The reserved word `val` serves to declare a name. The construct `fun inc (n) = n+1;` is syntactic sugar for `val inc = (fn n => n+1);`.) ■

EXAMPLE 7.105
Swap in ML

As a somewhat more plausible example, we can define a highly useful function that reverses a two-element tuple:

```
fun swap (a, b) = (b, a);
```

Since ML is (mostly) functional, swap is not intended to exchange the value of objects; rather, it takes a two-element tuple as argument, and produces the symmetrical two-element tuple as a result. ■

Pattern matching in ML works not only for tuples, but for any built-in or user-defined *constructor* of composite values. Constructors include the parentheses used for tuples, the square brackets used for lists, several of the built-in operators (`::`, `@`, etc.), and user-defined constructors of `datatypes` (see below). Literal constants are even considered to be constructors, so the tuple `t` can be matched against the pattern `(1, x)`: the match will succeed only if `t`'s first element is `1`.

In a call like `compare (t)` or `swap (2, 3)`, an ML implementation can tell at compile time that the pattern match will succeed: it knows all necessary information about the structure of the value being matched against the pattern. In other cases, the implementation can tell that a match is doomed to fail, generally because the types of the pattern and the value cannot be unified. The more interesting cases are those in which the pattern and the value have the same type (i.e., could be unified), but the success of the match cannot be determined until run time. If `l` is of type `int list`, for example, then an attempt to "deconstruct" `l` into its head and tail may or may not succeed, depending on `l`'s value:

EXAMPLE 7.106
Run-time pattern matching

```
let val head :: rest = l in ...
```

If `l` is `nil`, the attempted match will produce an *exception* at run time (we will consider exceptions further in Section 8.5). ■

We have seen how pattern matching works in function calls and `val` constructs. It is also supported by a `case` expression. Using `case`, the `append` function above could have been written as follows:

EXAMPLE 7.107
ML `case` expression

```
fun append (l1, l2) =
    case l1 of
        nil => l2
      | h :: t => h :: append (t, l2);
```

Here the code generated for the `case` expression will pattern-match `l1` first against `nil` and then against `h :: t`. The `case` expression evaluates to the subexpression following the => in the first arm whose pattern matches. The compiler will issue a warning message at compile time if the patterns of the arms are not exhaustive, or if the pattern in a later arm is completely covered by one in an earlier arm (implying that the latter will never be chosen). ■

A useless arm is probably an error, but harmless, in the sense that it will never result in a dynamic semantic error message. Nonexhaustive cases may be intentional, if the programmer can predict that the pattern will always work at run time. Our `append` function would have generated such a warning if written as follows:

EXAMPLE 7.108
Coverage of `case` labels

```
fun append (l1, l2) =
    if l1 = nil then l2
    else let val h::t = l1 in h :: append (t, l2) end;
```

Here the compiler is unlikely to realize that the `let` construct in the `else` clause will be elaborated only if `l1` is nonempty. (This example looks easy enough to figure out, but the general case is uncomputable, and most compilers won't contain special code to recognize easy cases.) ■

EXAMPLE 7.109
Function as a series of alternatives

When the body of a function consists entirely of a `case` expression, it can also be written as a simple series of alternatives:

```
fun append (nil, l2) = l2
 |  append (h::t, l2) = h :: append (t, l2);
```
■

Pattern matching features prominently in other languages as well, particularly those (such as Snobol, Icon, and Perl) that place a heavy emphasis on strings. ML-style pattern matching differs from that of string-oriented languages in its integration with static typing and type inference. Snobol, Icon, and Perl are all dynamically typed.

By casting "multiargument" functions in terms of tuples, ML eliminates the asymmetry between the arguments and return values of functions in many other languages. As shown by `swap` above, a function can return a tuple just as easily as it can take a tuple argument. Pattern matching allows the elements of the tuple to be extracted by the caller:

EXAMPLE 7.110
Pattern matching of return tuple

```
let val (a, b) = swap (c, d) in ...
```

Here `a` will have the value given by `d`; `b` will have the value given by `c`. ■

`Datatype` *Constructors*

In addition to lists and tuples, ML provides built-in constructors for records, together with a `datatype` mechanism that allows the programmer to introduce other kinds of composite types. A record is a composite object in which the elements have names, but no particular order (the language implementation must choose an order for its internal representation, but this order is not visible to the programmer). Records are specified using a "curly brace" constructor: `{name = "Abraham Lincoln", elected = 1860}`. (The same value can be denoted `{elected = 1860, name = "Abraham Lincoln"}`.) ■

EXAMPLE 7.111 ML records

EXAMPLE 7.112 ML datatypes

ML's `datatype` mechanism introduces a type name and a collection of constructors for that type. In the simplest case, the constructors are all functions of zero arguments, and the type is essentially an enumeration:

```
datatype weekday = sun | mon | tue | wed | thu | fri | sat;
```

In more complicated examples, the constructors have arguments, and the type is essentially a union (variant record):

```
datatype yearday = mmdd of int * int | ddd of int;
```

This code defines `mmdd` as a constructor that takes a pair of integers as argument, and `ddd` as a constructor that takes a single integer as argument. The intent is to allow days of the year to be specified either as (month, day) pairs or as integers in the range 1 . . 366. In a non–leap year, the Fourth of July could be represented either as `mmdd (7, 4)` or as `ddd (188)`, though the equality test `mmdd (7, 4) = ddd (188)` would fail unless we made `yearday` an *abstract* type (similar to the Euclid module types of Section 3.3.4), with its own, special, equality operation. ■

EXAMPLE 7.113 Recursive `datatypes`

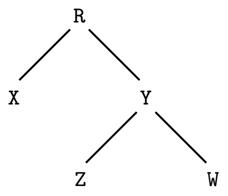
ML's `datatypes` can even be used to define recursive types, without the need for pointers. The canonical ML example is a binary tree:

```
datatype int_tree = empty | node of int * int_tree * int_tree;
```

By introducing an explicit type variable in the definition, we can even create a generic tree whose elements are of any homogeneous type:

```
datatype 'a tree = empty | node of 'a * 'a tree * 'a tree;
```

Given this definition, the tree



can be written `node (#"R", node (#"X", empty, empty), node (#"Y", node (#"Z", empty, empty), node (#"W", empty, empty)))`. Recursive types also appear in Lisp, Clu, Java, C#, and other languages with a reference model of variables; we will discuss them further in Section 7.7. ■

EXAMPLE 7.114 Type equivalence in ML

Because of its use of type inference, ML generally provides the effect of structural type equivalence. Definitions of `datatypes` can be used to obtain the effect of name equivalence when desired:

```
datatype celsius_temp = ct of int;
datatype fahrenheit_temp = ft of int;
```

A value of type `celsius_temp` can then be obtained by using the `ct` constructor:

```
val freezing = ct (0);
```

Unfortunately, `celsius_temp` does not automatically inherit the arithmetic operators and relations of `int`: unless the programmer defines these operators explicitly, the expression `ct (0) < ct (20)` will generate an error message along the lines of "operator not defined for type." ■

✓ **CHECK YOUR UNDERSTANDING**

54. Under what circumstances does an ML compiler announce a type clash?
55. Explain how the type inference of ML leads naturally to polymorphism.
56. What is a *type variable*? Give an example in which an ML programmer might use such a variable explicitly.
57. How do lists in ML differ from those of Lisp and Scheme?
58. Why do ML programmers often declare the types of variables, even when they don't have to?
59. What is *unification*? What is its role in ML?
60. List three contexts in which ML performs *pattern matching*.
61. Explain the difference between *tuples* and *records* in ML. How does an ML record differ from a record (structure) in languages like C or Pascal?
62. What are ML `datatype`s? What features do they subsume from imperative languages such as C and Pascal?