8 ◆ HASKELL

Haskell was designed by a committee whose job was to create a pure functional language. The language is named for Haskell Curry (whose name came up earlier in the discussion of ML) and is based loosely on Miranda. It uses a type mechanism similar to ML, has higher-order functions, allows polymorphism, and uses pattern matching. Unlike ML, Haskell is frequently compiled. Haskell has various input forms; my examples use syntactically correct code for the Gofer Haskell interpreter. Nonetheless, I use `in:` and `out:` to indicate values of expressions as if Haskell were interpreted. Because Haskell is a pure functional language, it has no pointers. Scoping is dynamic. Any function with more than one parameter is automatically curried. (ML also provides a mechanism for automatic currying that I didn't describe in the preceding section.) Some other differences between the two languages will become clear in the following examples.

In Haskell, we define functions in a way that is similar to ML, although Haskell has an unusual syntax in which indentation determines scope.[12] A typical Haskell definition looks like the one in Figure 1.54.

Figure 1.54

```
Max :: Int -> Int -> Int                                    1
Max a b                                                     2
      | a >= b     = a                                      3
      | otherwise  = b                                      4
```

We may specify the type of the function `Max` before defining it, as in line 1. Haskell can infer the type of a function from its definition, just like ML. This example shows the standard structure of Haskell function definitions: a series of conditional expressions and conclusions. When `Max` is invoked, it evaluates the conditional expressions until it finds one that holds; the value returned by the function is the associated conclusion. The expression **`otherwise`** is always true.

8.1 Lazy evaluation and enumeration

Haskell provides some extensions to ML. Haskell always uses **lazy evaluation** to evaluate expressions. I discuss lazy evaluation in more detail in Chapter 4; for now, let me just say that no term of an expression is evaluated until its value is actually needed, and then the term is evaluated as little as possible to provide any needed values. Figure 1.55

[12] This convention is not unique to Haskell; Miranda and ABC also use indentation to indicate grouping.

illustrates lazy evaluation by defining a function that only evaluates one of two parameters, depending on a conditional parameter.

Figure 1.55

```
Cond boolValue x y                                          1
     | boolValue   = x                                      2
     | otherwise   = y                                      3

in:  let x = 0 in Cond (x == 0) 0 (1/x)                     4
out: 0                                                      5
```

On line 4, the expression `(1/x)` should raise an exception. However, because its value is never needed, the expression is never evaluated and no exception is raised.

Haskell provides an operator that enumerates the values of certain types. For example, to create a list of integers from 1 to 10, we can use the notation in Figure 1.56.

Figure 1.56

```
in:  [1..10]                                                1
out: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]                        2
```

The type of an integer lists is denoted `[Int]`. The empty list is expressed as `[]`, and the **cons** operator is denoted by `:`. Lazy evaluation and enumerations make is easy to define infinite lists, which are not possible in ML.

Figure 1.57

```
naturalNumbers = [0..]                                      3
ones = 1 : ones                                             4
```

Both lists in Figure 1.57 have type `[Int]`. The list `naturalNumbers` contains all natural numbers, and `ones` is an infinite list all of whose elements are `1`. If we try to print out either list, Haskell will print as much of the list as we can tolerate before cancelling the display.

8.2 Comprehensions

Lists are often used to represent sets. Zermelo-Fraenkel set theory builds sets by extracting elements of other sets and manipulating the remaining elements. For example, the set of cubes of odd natural numbers is built from the set of natural numbers by extracting the odd numbers and then cubing them. Haskell **comprehensions** provide exactly this facility, as shown in Figure 1.58. A comprehension is an expression (which performs the manipulation), a list of **generators** (which build the base sets), and Boolean expressions (extractors). A generator is of the form `a <- [.....]`.

Figure 1.58

```
Squares = [n*n | n <- [1..5] ]                                    1

Fermat = [(a,b,c,n) | a <- [3..], b <- [3..], c <- [3..],         2
     n <- [3..], ((a^n) + (b^n) == (c^n))]                        3

QuickSort [] = []                                                 4
QuickSort (a:rest) =                                              5
     QuickSort [b | b <- rest, b <= a] ++                         6
     [a] ++                                                       7
     QuickSort [b | b <- rest, b > a]                             8
```

The list `Squares` (line 1) evaluates to `[1,4,9,16,25]`. The list `Fermat` evaluates to `[]`, but evaluation will never terminate! On lines 4–8 is a definition of Quicksort that uses list comprehensions and pattern matching (as in ML); `++` is the `append` operator.

The standard recursive algorithm for calculating Fibonacci numbers takes factorial time and is only good as an example of recursion. Instead, we should use dynamic programming (also called memoization), caching previous results and using them in later calculations. Lazy evaluation, comprehension, and infinite lists let us create an infinite cache that is only evaluated as needed. Figure 1.59 gives the code for generating the Fibonacci sequence.

Figure 1.59

```
Cache = [Fib x | x <- [0..]]                                      1

Fib :: Int -> Int                                                 2
Fib 0 = 1                                                         3
Fib 1 = 1                                                         4
Fib n = Cache!!(n-1) + Cache!!(n-2)                               5
```

The list `Cache` (line 1) is a list of all Fibonacci numbers. The `Fib` function (lines 2–5) looks up the appropriate values in the cache and returns their sum. The `!!` operator (line 5) is the subscript operator. The chart in Figure 1.60 shows the order of events in evaluating `Fib 4`.

Figure 1.60

```
Fib 4                                                    1
     Cache!!3                                            2
          Cache!!0                                       3
               Fib 0 returns 1; Cache!!0 becomes 1       4
          Cache!!1                                       5
               Fib 1 returns 1; Cache!!1 becomes 1       6
          Cache!!2                                       7
               Fib 2                                     8
                    Cache!!1 returns 1                   9
                    Cache!!0 returns 1                   10
                    Fib 2 returns 2; Cache!!2 becomes 2  11
          Fib 3                                          12
               Cache!!2 returns 2                        13
               Cache!!1 returns 1                        14
               Fib 3 returns 3; Cache!!3 becomes 3       15
     Cache!!2 returns 2                                  16
     returns 5                                           17
```

Another way to express dynamic programming for computing Fibonacci numbers is described in Chapter 9 in the section on mathematics languages.

8.3 Polymorphic types

Haskell allows polymorphic data types much like those in ML. Some examples are shown in Figure 1.61.

Figure 1.61

```
data Tree a = Nil | Node a (Tree a) (Tree a)          1
bigTree = Node 7 bigTree bigTree                      2

type BinOp a = a -> (a -> a)                          3

f :: BinOp Int                                        4
f x y = x + y                                         5

type Matrix a = [[a]]                                 6
type BoolMatrix = Matrix Bool                         7

aMatrix :: BoolMatrix                                 8
aMatrix = [[True,False], [False,False]]               9

firstRow :: BoolMatrix -> [Bool]                      10
firstRow [] = []                                      11
firstRow (rowOne:otherRows) = rowOne                  12
```

```
in:  firstRow aMatrix                                    13
out: [True, False]                                       14
```

Tree (line 1) is a polymorphic type with a type parameter a. The definition of the infinite bigTree in line 2 looks much like the definition of ones in Figure 1.57. BinOp (line 3) is a polymorphic type with type parameter a. The function f (lines 4–5) is a function of type BinOp Int. A matrix can be defined as a list of lists (line 6) and can be specialized to a Boolean matrix (line 7). The function firstRow (lines 10–12) uses the underlying definition of BoolMatrix to return the first row of a matrix. If we didn't specify the type of firstRow in line 10, Haskell would infer its type as [[a]]->[a], which is a generalization of the type we want. ML would infer the same type (denoted 'a **list list** -> 'a **list**).

8.4 Interfaces

What should be the inferred type of QuickSort in Figure 1.58 on page 47? ML requires that QuickSort be given an explicit type, because ML can't unify the use of the <= operator, which is only defined on integers and floats and cannot be unified. Haskell uses a different type scheme that allows it to infer the type of QuickSort completely.

Haskell lets the programmer define an **interface**, which is a set of types.[13] An interface is defined by its **signature**, which is a list of functions whose parameters and return values match the types in the interface. Each member of the interface must satisfy its signature, and any type that satisfies the signature is automatically a member of the interface. For example, any type for which equality (==) is defined is a member of the Eq interface. The definition of Eq is shown in Figure 1.62.

Figure 1.62

```
interface Eq t where                                     1
     (==), (/=) :: t -> t -> Bool                        2
     a /= b = not (a==b)                                 3

member Eq Bool where                                     4
     True  == True  = True                               5
     False == False = True                               6
     _     == _     = False                              7
```

Eq has two functions (line 2): equality (==) and inequality (/=), both of

[13] Unfortunately, interfaces are called "classes" in Haskell, which conflicts with the use of the term in object-oriented programming languages (Chapter 5). I will use the term "interface" here for clarity; I am borrowing the term from Java, which uses it for a similar purpose.

type t -> t -> Bool. The functions are placed in parentheses to indicate that they are infix operators. An interface can include default definitions for some functions. The /= operator (line 3) has a default definition based on == . In lines 4–7, I make Bool a member of the Eq interface by defining the == operator. Integers and floats are automatically members of the Eq interface. Figure 1.63 illustrates more basic definitions.

Figure 1.63

```
member x [] = False                                              1
member y (first:rest) = (y == first) || member y rest            2
-- inferred type: member :: Eq a => a -> [a] -> Bool             3

interface Pair t where                                           4
      PairFn :: t -> t -> t                                      5

PairTester x y = (PairFn x y) + x                                6
-- inferred type:                                                7
--         PairTester :: (Num a, Pair a) => a -> a -> a          8

instance Pair Int where                                          9
     PairFn x y = x * y                                          10

in: PairTester 3 4                                               11
out: 15                                                          12

multipleContext x y z = ((x == z), (y + y))                      13
-- inferred type: multipleContext :: (Num b, Eq a) =>            14
--         a -> b -> a -> (Bool, b)                              15
```

The member function (lines 1–3) takes a list and an element as parameters and returns True if and only if the element is in the list. Its inferred type is polymorphic, as we might expect, but a -> [a] -> Bool doesn't completely describe that polymorphic type. In addition, equality must be defined on type a; that is, a must be an member of interface Eq, as shown in line 3.

Haskell can infer multiple interfaces as the context. In lines 4–5, I define my own interface Pair, whose signature is a single binary function, PairFn. When I define the function PairTester (line 6), Haskell infers the context Pair a and Num a, because PairTester requires both PairFn and + . I then make Int a member of Pair by defining PairFn for integers (lines 9–10) and then applying PairTester to a pair of integers (lines 11–12). In lines 13–15, Haskell infers one context for type a and a different context for type b.

Now we can describe the type of QuickSort (Figure 1.58, page 47). The interface Ord provides the ordering operators > and <= . (Ord is a

subinterface of `Eq`; that is, all members of `Ord` also belong to `Eq`.) The type of `QuickSort` is therefore `QuickSort :: Ord a => [a] -> [a]`.

I can now use algebraic data types and interfaces to define my own integer type, `MyInt`. (Algebraic data types are discussed in more detail in Chapter 8.) I'll define an integer as either `0`, the predecessor of an integer, or the successor of an integer, as shown in Figure 1.64.

Figure 1.64

```
data MyInt = Zero | Pred MyInt | Succ MyInt                    1

reduce (Pred (Succ n))     = n                                 2
reduce (Succ (Pred n))     = n                                 3
reduce (Pred n)            = Pred (reduce n)                   4
reduce (Succ n)            = Succ (reduce n)                   5
reduce n                   = n                                 6
-- inferred type: reduce :: MyInt -> MyInt                     7

in:  reduce (Pred (Pred (Succ (Zero))))                        8
out: Pred Zero                                                 9

intToMyInt a                                                   10
     | a == 0   = Zero                                         11
     | a < 0    = Pred (intToMyInt (a+1))                      12
     | a > 0    = Succ (intToMyInt (a-1))                      13
-- inferred type: intToMyInt :: Int -> MyInt                   14

in: intToMyInt (-3)                                            15
out: Pred (Pred (Pred Zero))                                   16

instance Eq MyInt where                                        17
     Zero == Zero          = True                              18
     (Pred a) == (Pred b)  = (a == b)                          19
     (Succ a) == (Succ b)  = (a == b)                          20
     _        == _         = False                             21
```

```
instance Ord MyInt where                                              22
     Zero < a   = case (reduce a) of                                  23
          Succ x     -> True                                          24
          _          -> False                                         25
     a < b      = case (reduce a) of                                  26
          Zero       -> Zero < reduce b                               27
          Pred x     -> x    < reduce (Succ b)                        28
          Succ x     -> x    < reduce (Pred b)                        29
     a <= b     = (a == b) || (a < b)                                 30

instance Enum MyInt where                                             31
     enumFrom a = a : (enumFrom (reduce (Succ a)))                    32

in:  Zero > (Pred Zero)                                               33
out: False                                                            34

in:  [(intToMyInt (-2)) .. (intToMyInt (2))]                          35
out: [Pred (Pred Zero), Pred Zero, Zero, Succ Zero,                   36
          Succ (Succ Zero)]                                           37

in:  [(intToMyInt (0))..]                                             38
out: [Zero, Succ Zero, Succ (Succ Zero), ...                          39

in:  (Pred Zero) /= Zero                                              40
out: True                                                             41
```

The function `reduce` (lines 2–6) simplifies a value of type `MyInt`. Miranda, a predecessor of Haskell, lets a program specify simplification rules to be executed automatically every time a new member of the type is created; in Haskell, we invoke `reduce` explicitly when we need it, as in line 8. In lines 10–13, I define a conversion function from integer to `MyInt`.

The most interesting part of Figure 1.64 is lines 17–30, in which I make `MyInt` a member of the `Eq`, `Ord`, and `Enum` interfaces. You have seen the first two interfaces already. Members of the `Enum` interface allow enumerations, as in Figure 1.56. To make all the functions in the three signatures work, I only have to define a few primitive functions and take advantage of the default definitions of the other functions. The `[a..]` notation is a shorthand for the `enumFrom` function (line 32). The `Enum` interface includes a default definition of the `..` operator, used in lines 35 and 38, based on `enumFrom`.