A Gentle Introduction to Haskell

Paul Hudak
Yale University
Department of Computer Science

Joseph H. Fasel
University of California
Los Alamos National Laboratory

1 Introduction

This edition of the tutorial has been updated to reflect changes in the Haskell language that will appear in the next version of the Haskell report, 1.3. These changes have been implemented in version 2.2 of the Yale Haskell system and this particular edition of the tutorial has been altered from its original form to reflect these changes. The only significant change in this tutorial is that the section on Input/Output has been updated for the new monadic I/O system.

Our purpose in writing this tutorial is not to teach programming, nor even to teach functional programming. Rather, it is intended to serve as a supplement to the Haskell Report [3], which is otherwise a rather dense technical exposition. Our goal is to provide a gentle introduction to Haskell for someone who has experience with at least one other language, preferably a functional language (even if only an "almost-functional" language such as ML or Scheme). If the reader wishes to learn more about the functional programming style, we highly recommend Bird and Wadler's text *Introduction to Functional Programming* [1], which uses a language sufficiently similar to Haskell to make translation between the two quite easy. For a useful survey of functional programming languages and techniques, including some of the language design principles used in Haskell, see [2].

Our general strategy for introducing language features is this: motivate the idea, define some terms, give some examples, and then point to the Report for details. We suggest, however, that the reader completely ignore the details until this document has been completely read. On the other hand, Haskell's Standard Prelude (in Appendix A of the Report) contains lots of useful examples of Haskell code; we encourage a thorough reading once this tutorial is completed. This will not only give the reader a feel for what real Haskell code looks like, but will also familiarize her with Haskell's standard set of predefined functions and types.

[We have also taken the course of not laying out a plethora of lexical syntax rules at the outset. Rather, we introduce them incrementally as our examples demand, and enclose them in brackets, as with this paragraph. This is in stark contrast to the organization of the Report, although the Report remains the authoritative source for details (references such as "§2.1" refer to sections in the Report).]

Haskell is a *typeful* programming language[1]. Types are pervasive, and the newcomer is best off becoming well-aware of the full power and complexity of Haskell's type system from the outset. For

---

[1] A phrase due to Luca Cardelli.

those whose only experience is with relatively "untypeful" languages such as Basic or Lisp, this may be a difficult adjustment; for those familiar with Pascal, Modula, or even ML, the adjustment should be easier but still not insignificant, since Haskell's type system is different and somewhat richer than most. In any case, "typeful programming" is part of the Haskell programming experience, and cannot be avoided.

2 Values, Types, and Other Goodies

Because Haskell is a purely functional language, all computations are done via the evaluation of *expressions* (syntactic terms) to yield *values* (abstract entities that we regard as answers). Every value has an associated *type*. (Intuitively, we can think of types as sets of values.) Examples of expressions include atomic values such as the integer `5`, the character `'a'`, and the successor function `succ`, as well as structured values such as the list `[1,2,3]` and the pair `('b',4)`.

Just as expressions denote values, *type expressions* are syntactic terms that denote *type values* (or just *types*). Examples of type expressions include the atomic types `Int` (fixed-precision integers), `Char` (ASCII characters), `Int->Int` (functions mapping `Int` to `Int`), as well as the structured types `[Int]` (homogeneous lists of integers) and `(Char,Int)` (character/integer pairs).

All Haskell values are "first-class"—they may be passed as arguments to functions, returned as results, placed in data structures, etc. Haskell types, on the other hand, are *not* first-class. Types in a sense *describe* values, and the association of a value with its type is called a *typing*. Using the examples of values and types above, we write typings as follows:

```
        5  :: Int
      'a'  :: Char
     succ  :: Int -> Int
  [1,2,3]  :: [Int]
  ('b',4)  :: (Char,Int)
```

The `::` can be read "has type."

Functions in Haskell are normally defined by a series of *equations*. For example, the successor function `succ` can be defined by the single equation:

```
succ n          = n+1
```

An equation is an example of a *declaration*. Another kind of declaration is a *type signature declaration* (§4.4.1), with which we can declare an explicit typing for `succ`:

```
succ            :: Int -> Int
```

We will have much more to say about function definitions in Section 3.

For pedagogical purposes, when we wish to indicate that an expression e_1 evaluates, or "reduces," to another expression or value e_2, we will write:

$$e_1 \quad \Rightarrow \quad e_2$$

For example, note that:

$$\texttt{succ (succ 3)} \quad \Rightarrow \quad \texttt{5}$$

Haskell's *static type system* defines the formal relationship between types and values (§4.1.3). The static type system ensures that Haskell programs are *type safe*; that is, that the programmer has not mismatched types in some way. For example, we cannot generally add together two characters, so the expression `'a'+'b'` is ill-typed. The main advantage of statically typed languages is well-known: All type errors are detected at compile-time. This not only aids the user in reasoning about programs, but also permits a compiler to generate more efficient code (for example, no run-time type tags or tests are required).

The type system also ensures that user-supplied type signatures are correct. In fact, Haskell's type system is powerful enough to allow us to avoid writing any type signatures at all,[2] in which case we say that the type system *infers* the correct types for us. Nevertheless, judicious placement of type signatures is a good idea, as we did for `succ`, since it improves readability and helps bring programming errors to light.

[The reader will note that we have capitalized identifiers that denote specific types, such as `Int` and `Char`, but not identifiers that denote values, such as `succ`. This is not just a convention: it is enforced by Haskell's lexical syntax. In fact, the case of the other characters matters, too: `foo`, `fOo`, and `fOO` are all distinct identifiers.]

2.1 Polymorphic Types

Haskell also incorporates *polymorphic* types—types that are universally quantified in some way over *all* types. Polymorphic type expressions essentially describe *families* of types. For example, `(∀a)[a]` is the family of types consisting of, for every type `a`, the type of lists of `a`. Lists of integers (e.g. `[1,2,3]`), lists of characters (`['a','b','c']`), even lists of lists of integers, etc., are all members of this family. (Note, however, that `[2,'b']` is *not* a valid example, since there is no single type that contains both `2` and `'b'`.)

[Identifiers such as `a` above are called *type variables*, and are uncapitalized to distinguish them from specific types such as `Int`. Furthermore, since Haskell has only universally quantified types, there is no need to explicitly write out the symbol for universal quantification, and thus we simply write `[a]` in the example above. In other words, all type variables are implicitly universally quantified.]

Lists are a commonly used data structure in functional languages, and are a good vehicle for explaining the principles of polymorphism. The list `[1,2,3]` in Haskell is actually shorthand for the list `1:(2:(3:[]))`, where `[]` is the empty list and `:` is the infix operator that adds its first argument to the front of its second argument (a list).[3] Since `:` is right associative, we can also write this list as `1:2:3:[]`.

As an example of a user-defined function that operates on lists, consider the problem of counting the number of elements in a list:

---

[2] With a few exceptions to be described later.

[3] `:` and `[]` are like Lisp's `cons` and `nil`, respectively.

```
length                  :: [a] -> Int
length  []              = 0
length (x:xs)           = 1 + length xs
```

This definition is almost self-explanatory. We can read the equations as saying: "The length of the empty list is 0, and the length of a list whose first element is `x` and remainder is `xs` is 1 plus the length of `xs`." (Note the naming convention used here; `xs` is the plural of `x`, and should be read that way.)

Although intuitive, this example highlights an important aspect of Haskell that is yet to be explained: *pffttffrn mfftffffffnff*. The left-hand sides of the equations contain *pffttffrns* such as `[]` and `x:xs`. In a function application these patterns are *mfftffffffff* against actual parameters in a fairly intuitive way (`[]` only matches the empty list, and `x:xs` will successfully match any list with at least one element, binding `x` to the first element and `xs` to the rest of the list). If the match succeeds, the right-hand side is evaluated and returned as the result of the application. If it fails, the next equation is tried, and if all equations fail, an error results.

Defining functions by pattern matching is quite common in Haskell, and the user should become familiar with the various kinds of patterns that are allowed; we will return to this issue in Section 3.14.

`length` is also an example of a *polymorpffffff ffunfftffon*. It can be applied to a list containing elements of any type. For example:

```
length [1,2,3]          ⇒     3
length ['a','b','c']    ⇒     3
length [[],[],[]]       ⇒     3
```

Here are two other useful polymorphic functions on lists that will be used later:

```
head                    :: [a] -> a
head (x:xs)             =  x

tail                    :: [a] -> [a]
tail (x:xs)             =  xs
```

With polymorphic types, we find that some types are in a sense strictly *morff ffffnffrffl* than others. For example, the type `[a]` is more general than `[Char]`. In other words, the latter type can be derived from the former by a suitable substitution for `a`. With regard to this generalization ordering, Haskell's type system possesses two important properties: First, every well-typed expression is guaranteed to have a unique *prffnffffpffl* type (explained below), and second, the principal type can be *ffnffffrrffff* automatically (§4.1.3). In comparison to a *monomorpffffffffflly typffff* language such as Pascal, the reader will find that *polymorpffffsm* improves expressiveness, and *typff ffnffffrffnffff* lessens the burden of types on the programmer.

An expression's or function's principal type is the least general type that, intuitively, "contains all instances of the expression." For example, the principal type of `head` is `[a]->a`; the types `[b]->a`, `a->a`, or even `a` are *too general*, whereas something like `[Int]->Int` is *too specific*. The existence of unique principal types is the hallmark feature of the *Hindley-Milner type system*, which forms the basis of the type systems of Haskell, ML, Miranda,[4] and several other (mostly functional) languages.

2.2 User-Defined Types

We can define our own types in Haskell using a `data` declaration, which we introduce via a series of examples (§4.2.1).

An important predefined type in Haskell is that of truth values:

```
data Bool               = False | True
```

The type being defined here is `Bool`, and it has exactly two values: `True` and `False`. `Bool` is an example of a (nullary) *type constructor*, and `True` and `False` are (also nullary) *data constructors* (or just *constructors*, for short).

Similarly, we might wish to define a color type:

```
data Color              = Red | Green | Blue | Indigo | Violet
```

Both `Bool` and `Color` are examples of *enumerated types*, since they consist of a finite number of nullary data constructors.

Here is an example of a type with just one data constructor:

```
data Point a            = Pt a a
```

Because of the single constructor, a type like `Point` is often called a *tuple type*, since it is essentially just a cartesian product (in this case binary) of other types.[5] In contrast, multi-constructor types, such as `Bool` and `Color`, are called (disjoint) *union* types.

More importantly, however, `Point` is an example of a *polymorphic* type: for any type t, it defines the type of cartesian points that use t as the coordinate type. `Point` can now be seen clearly as a unary type constructor, since from the type t it constructs a new type `Point` t. (In the same sense, using the list example given earlier, `[_]` is also a type constructor (where we have used "`_`" to denote the missing argument): given any type t we can "apply" `[_]` to yield a new type `[`t`]`. Similarly, `_->_` is a type constructor: given two types t and u, t`->`u is the type of functions mapping elements of type t to elements of type u.)

Note that the type of the binary constructor `Pt` is `a -> a -> Point a`, and thus the following typings are valid:

---

[4] "Miranda" is a trademark of Research Software, Ltd.

[5] Tuples are somewhat like *records* in other languages, except that the elements are positional, rather than having names (labels) associated with them.

```
Pt  2.0  3.0            :: Point Float
Pt  'a'  'b'            :: Point Char
Pt True False           :: Point Bool
```

On the other hand, an expression such as `Pt 'a' 1` is ill-typed.

It is important to distinguish between applying a *ffonstrufftor* to yield a *vffluff*, and applying a *typff ffonstrufftor* to yield a *typff*; the former happens at run-time and is how we compute things in Haskell, whereas the latter happens at compile-time and is part of the type system's process of ensuring type safety.

2.3 Recursive Types

Types can also be recursive, as in:

```
data Tree a             = Leaf a | Branch (Tree a) (Tree a)
```

Here we have defined a polymorphic binary tree type whose elements are either leaf nodes containing a value of type `a`, or internal nodes ("branches") containing (recursively) two sub-trees.

When reading data declarations such as this, remember that `Tree` is a type constructor, whereas `Branch` and `Leaf` are data constructors. Aside from establishing a connection between these constructors, the above declaration is essentially defining the following types for `Branch` and `Leaf`:

```
Branch                  :: Tree a -> Tree a -> Tree a
Leaf                    :: a -> Tree a
```

With this example we have defined a type sufficiently rich to allow defining some interesting (recursive) functions that use it. For example, suppose we wish to define a function `fringe` that returns a list of all the elements in the leaves of a tree from left to right. It's usually helpful to write down the type of new functions first; in this case we see that the type should be `Tree a -> [a]`. That is, `fringe` is a polymorphic function that, for any type `a`, maps trees of `a` into lists of `a`. A suitable definition follows:

```
fringe :: Tree a -> [a]
fringe (Leaf x)            = [x]
fringe (Branch left right) = fringe left ++ fringe right
```

Where `++` is the infix operator that concatenates two lists (its full definition will be given in Section 3.2). As with the `length` example given earlier, `fringe` is defined using pattern matching, except that here we see patterns involving user-defined constructors: `Leaf` and `Branch`. [Note that the formal parameters are easily identified as the ones beginning with lower-case letters.]

2.4 Type Synonyms

For convenience, Haskell provides a way to define *typff synonyms*; i.e. names for commonly used types. Type synonyms are created using a `type` declaration (§4.2.2). Here are several examples:

```
type String             = [Char]
type Person             = (Name,Address)
type Name               = String
data Address            = None | Addr String
```

Type synonyms do not define new types, but simply give new names for existing types. For example, the type `Person -> Name` is precisely equivalent to `(String,Address) -> String`. The new names are often shorter than the types they are synonymous with, but this is not the only purpose of type synonyms: they can also improve readability of programs by being more mnemonic; indeed, the above examples highlight this. We can even give new names to polymorphic types:

```
type AssocList a b              = [(a,b)]
```

This is the type of "association lists" which associate values of type `a` with those of type `b`.

2.5 Built-in Types Are Not Special

Earlier we introduced several "built-in" types such as lists, tuples, integers, and characters. We have also shown how new user-defined types can be defined. Aside from special syntax, are the built-in types in any way more special than the user-defined ones? The answer is *no*. The special syntax is for convenience and for consistency with historical convention, but has no semantic consequence.

We can emphasize this point by considering what the type declarations would look like for these built-in types if in fact we were allowed to use the special syntax in defining them. For example, the `Char` type might be written as:

```
data Char       = ’a’ | ’b’ | ’c’ | ...         -- This is not valid
                | ’A’ | ’B’ | ’C’ | ...         -- Haskell code!
                | ’1’ | ’2’ | ’3’ | ...
                ...
```

These constructor names are not syntactically valid; to fix them we would have to write something like:

```
data Char       = Ca | Cb | Cc | ...
                | CA | CB | CC |
```

the above definitions make abundantly clear. In particular, note the recursive nature of the list type whose elements are homogeneous and of arbitrary length, and the non-recursive nature of a (particular) tuple type whose elements are heterogeneous and of fixed length. The typing rules for tuples and lists should now also be clear:

> For $(e_1,e_2,\ldots,e_n)$, $n \geq 2$, if t_i is the type of e_i, then the type of the tuple is $(t_1,t_2,\ldots,t_n)$.
>
> For $[e_1,e_2,\ldots,e_n]$, $n \geq 0$, each e_i must have the same type t, and the type of the list is $[t]$.

2.5.1 List Comprehensions and Arithmetic Sequences

As with Lisp dialects, lists are pervasive in Haskell, and as with other functional languages, there is yet more syntactic sugar to aid in their creation. Aside from the constructors for lists just discussed, Haskell provides an expression known as a *list comprehension* that is best explained by example:

```
[ f x | x <- xs ]
```

This expression can intuitively be read as "the list of all `f x` such that `x` is drawn from `xs`." The similarity to set notation is not a coincidence. The phrase `x<-xs` is called a *generator*, of which more than one is allowed, as in:

```
[ (x,y) | x<-xs, y<-ys ]
```

This list comprehension forms the cartesian product of the two lists `xs` and `ys`. The elements are selected as if the generators were "nested" from left to right (with the rightmost generator varying fastest); thus, if `xs` is `[1,2]` and `ys` is `[3,4]`, the result is `[(1,3),(1,4),(2,3),(2,4)]`.

Besides generators, boolean expressions called *guards* are permitted. Guards place constraints on the elements generated. For example, here is a concise definition of everybody's favorite sorting algorithm:

```
quicksort  []           =  []
quicksort (x:xs)        =  quicksort [y | y <- xs, y<x ]
                        ++ [x]
                        ++ quicksort [y | y <- xs, y>=x]
```

To further support the use of lists, Haskell has special syntax for *arithmetic sequences*, which are best explained by a series of examples:

```
[1..10]     ⇒    [1,2,3,4,5,6,7,8,9,10]
[1,3..10]   ⇒    [1,3,5,7,9]
[1,3..]     ⇒    [1,3,5,7,9, ...    (infinite sequence)
```

More will be said about arithmetic sequences in Section 5.2, and "infinite lists" in Section 3.4.

2.5.2 Strings

As another example of syntactic sugar for built-in types, we note that the literal string `"hello"` is actually shorthand for the list of characters `['h','e','l','l','o']`. Indeed, the type of `"hello"` is `String`, where `String` is a predefined type synonym (that we gave as an earlier example):

```
type String             = [Char]
```

This means we can use predefined polymorphic list functions to operate on strings. For example:

`"hello" ++ " world"` ⇒ `"hello world"`

3 Functions

Since Haskell is a functional language, one would expect functions to play a major role, and indeed they do. In this section, we look at several aspects of functions in Haskell.

First, consider this definition of a function which adds its two arguments:

```
add                     :: Int -> Int -> Int
add x y                 = x + y
```

This is an example of a *curried* function[6]. An application of `add` has the form `add` e_1 e_2, and is equivalent to `(add` e_1`)` e_2, since function application associates to the *left*. In other words, applying `add` to one argument yields a new function which is then applied to the second argument. This is consistent with the type of `add`, `Int->Int->Int`, which is equivalent to `Int->(Int->Int)`; i.e. `->` associates to the *right*. Indeed, using `add`, we can define `succ` in a different way from what we did earlier:

```
succ                    = add 1
```

This is an example of the *partial application* of a curried function, and is one way that a function can be returned as a value. Let's consider a case in which it's useful to pass a function as an argument. The well-known `map` function is a perfect example:

```
map                     :: (a->b) -> [a] -> [b]
map f  []               = []
map f (x:xs)            = f x : map f xs
```

[Function application has higher precedence than any infix operator, and thus the right-hand side

---

[6] The name *curry* derives from the person who popularized the idea: Haskell Curry. To get the effect of an *uncurried* function, we could use a *tuple*, as in:

```
add (x,y)               = x + y
```

But then we see that this version of `add` is really just a function of one argument!

of the second equation parses as `(f x) : (map f xs)`.] `map` is a polymorphic function, and its type indicates clearly that its first argument is a function; note also that the two `a`'s must be instantiated with the same type (likewise for the `b`'s). As an example of the use of `map`, we can increment the elements in a list:

```
map (add 1) [1,2,3]    ⇒    [2,3,4]
```

These examples demonstrate the first-class nature of functions, which when used in this way are usually called *higher-order* functions.

3.1 Lambda Abstractions

Instead of using equations to define functions, we can also define them "anonymously" via a *lambda abstraction*. For example, a function equivalent to `succ` could be written as `\x -> x+1`. Similarly, the function `add` is equivalent to `\x -> \y -> x+y`. Nested lambda abstractions such as this may be written using the equivalent shorthand notation `\x y -> x+y`. In fact, the equations:

```
succ x                  = x+1
add  x y                = x+y
```

are really shorthand for:

```
succ                    = \x   -> x+1
add                     = \x y -> x+y
```

We will have more to say about such equivalences later.

In general, given that `x` has type t_1 and `exp` has type t_2, then `\x->exp` has type t_1`->`t_2.

3.2 Infix Operators

Infix operators are really just functions, and can also be defined using equations. For example, here is the definition of Haskell's list concatenation operator:

```
(++)                    :: [a] -> [a] -> [a]
[]     ++ ys            =  ys
(x:xs) ++ ys            =  x : (xs++ys)
```

[Lexically, infix operators consist entirely of "symbols," as opposed to normal identifiers which are alphanumeric (§2.3). Haskell has no prefix operators, with the exception of minus (`-`), which is both infix and prefix.]

As another example, an important infix operator on functions is that for *function composition*:

```
(.)                     :: (b->c) -> (a->b) -> (a->c)
f . g                   = \ x -> f (g x)
```

3 Fortunately, most ASCII t

the typefiont used in this manuscript.

---

[8] Fixity declarations must only appear at the very beginning of a Haskell *module*, as will be described in Section 6.

Both of these specify right-associativity, the first with a precedence level of 5, the other 9. Left associativity is specified via `infixl`, and non-associativity by `infix`. Also, the fixity of more than one operator may be specified with the same fixity declaration. If no fixity declaration is given for a particular operator, it defaults to `infixl 9`. (See §5.7 for a detailed definition of the associativity rules.)

3.3 Functions are Non-strict

Suppose `bot` is defined by:

```
bot                     = bot
```

In other words, `bot` is a non-terminating expression. Abstractly, we denote the *value* of a non-terminating expression as ⊥ (read "bottom"). Expressions that result in some kind of a run-time error, such as `1/0`, also have this value.

A function `f` is said to be *strict* if, when applied to a nonterminating expression, it also fails to terminate. In other words, `f` is strict iff the value of `f bot` is ⊥. For most programming languages, *all* functions are strict. But this is not so in Haskell. As a simple example, consider `const1`, the constant 1 function, defined by:

```
const1 x                = 1
```

The value of `const1 bot` in Haskell is `1`. Operationally speaking, since `const1` does not "need" the value of its argument, it never attempts to evaluate it, and thus never gets caught in a nonterminating computation. For this reason, non-strict functions are also called "lazy functions," and are said to evaluate their arguments "lazily," or "by need."

Since error and nonterminating values are semantically the same in Haskell, the above argument also holds for errors. For example, `const1 (1/0)` also evaluates properly to `1`.

Non-strict functions are extremely useful in a variety of contexts. The main advantage is that they free the programmer from many concerns about evaluation order. Computationally expensive values may be passed as arguments to functions without fear of them being computed if they are not needed. An important example of this is a possibly *infinite* data structure.

3.4 "Infinite" Data Structures

One advantage of the non-strict nature of Haskell is that data constructors are non-strict, too. This should not be surprising, since constructors are really just a special kind of function (the distinguishing feature being that they can be used in pattern matching). For example, the constructor for lists, `(:)`, is non-strict.

Non-strict constructors permit the definition of (conceptually) *infinite* data structures. Here is an infinite list of ones:

```
ones                    = 1 : ones
```

Perhaps more interesting is the function `numsFrom`:

```
numsFrom n              = n : numsFrom (n+1)
```

Thus `numsFrom n` is the infinite list of successive integers beginning with `n`. From it we can construct an infinite list of squares:

```
squares                 = map (^2) (numsfrom 0)
```

(Note the use of a section; `^` is the infix exponentiation operator.)

Of course, eventually we expect to extract some finite portion of the list for actual computation, and there are lots of predefined functions in Haskell that do this sort of thing: `take`, `takeWhile`, `filter`, and others (see the portion of the Standard Prelude called `PreludeList`). For example, `take` removes the first `n` elements from a list:

`take 5 squares` ⇒ `[0,1,4,9,16]`

The definition of `ones` above is an example of a *circular list*. In most circumstances this has an important impact on efficiency, since an implementation can be expected to implement the list as a true circular structure, thus saving space.

For another example of the use of circularity, the Fibonacci sequence can be computed efficiently as the following infinite sequence:

```
fib             = 1 : 1 : [ a+b | (a,b) <- zip fib (tail fib) ]
```

where `zip` is a Standard Prelude function that returns the pairwise interleaving of its two list arguments:

```
zip (x:xs) (y:ys)       = (x,y) : zip xs ys
zip  xs     ys          = []
```

Note how `fib`, an infinite list, is defined in terms of itself, as if it were "chasing its tail." Indeed, we can draw a picture of this computation as shown in Figure 1a.

For another application of infinite lists, see Section 4.4.

3.5 The Error Function

Haskell has a built-in function called `error` whose type is `String->a`. This is a somewhat odd function: From its type it looks as if it is returning a value of a polymorphic type about which it knows nothing, since it never receives a value of that type as an argument!

In fact, there *is* one value "shared" by all types: $\perp$. Indeed, semantically that is exactly what value is always returned by `error` (recall that all errors have value $\perp$). However, we can expect that a reasonable implementation will print the string argument to `error` for diagnostic purposes. Thus this function is useful when we wish to terminate a program when something has "gone wrong."

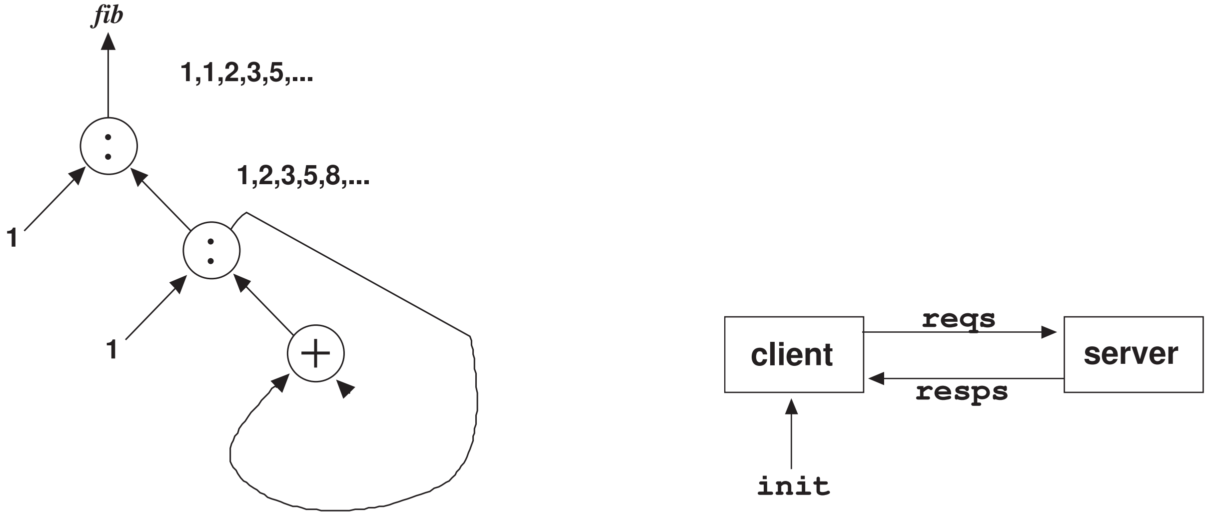

Figure 1: (a) Circular Fibonacci Sequence (b) Server/Client Simulation

For example, the actual definition of `head` taken from the Standard Prelude is:

```
head (x:xs)             =  x
head  []                =  error "head{PreludeList}: head []"
```

4 Case Expressions and Pattern Matching

Earlier we gave several examples of pattern matching in defining functions—for example `length` and `fringe`. In this section we will look at the pattern-matching process in much greater detail (§3.14).[9]

Patterns are not "first-class;" there is only a fixed set of different kinds of patterns. We have already seen several examples of *data constructor* patterns; both `length` and `fringe` defined earlier use such patterns, the former on the constructors of a "built-in" type (lists), the latter on a user-defined type (`Tree`). Indeed, matching is permitted using the constructors of any type, user-defined or not. This includes tuples, strings, numbers, characters, etc. For example, here's a contrived function that matches against a tuple of "constants:"

```
contrived :: ([a], Char, (Int, Float), String, Bool) -> Bool
contrived    ([],  'b',  (1,   2.0),   "hi",   True) = False
```

This example also demonstrates that *nesting* of patterns is permitted (to arbitrary depth).

---

[9] Pattern matching in Haskell is very different from that found in logic programming languages such as Prolog; in particular, it can be viewed as "one-way" matching, whereas Prolog allows "two-way" matching (via unification), along with implicit backtracking in its evaluation mechanism.

Technically speaking, *formal parameters*[10] are also patterns—it's just that they *never fail to match a value*. As a "side effect" of the successful match, the formal parameter is bound to the value it is being matched against. For this reason patterns in any one equation are not allowed to have more than one occurrence of the same formal parameter (a property called *linearity* §3.14, §3.2,§4.4.2).

Patterns such as formal parameters that never fail to match are said to be *irrefutable*, in contrast to *refutable* patterns such as the ones given in the `contrived` example above. There are three other kinds of irrefutable patterns, two of which we will introduce now (the other we will delay until Section 4.4).

As-patterns. Sometimes it is convenient to name a pattern for use on the right-hand side. For example, a function that duplicates the first element in a list might be written as:

```
f (x:xs)                = x:x:xs
```

(Recall that : associates to the right.) Note that `x:xs` appears both as a pattern on the left-hand side, and an expression on the right-hand side. To improve readability, we might prefer to write `x:xs` just once, which we can achieve using an *as-pattern* as follows:[11]

```
f s@(x:xs)              = x:s
```

Technically speaking, as-patterns always result in a successful match, although the sub-pattern (in this case `x:xs`) could, of course, fail.

Wild-cards. Another common situation is matching against a value we really care nothing about. For example, the functions `head` and `tail` defined in Section 2.1 can be rewritten as:

```
head (x:_)              = x
tail (_:xs)             = xs
```

in which we have "advertised" the fact that we don't care what a certain part of the input is. Each wild-card will independently match anything, but in contrast to a formal parameter, each will bind nothing; for this reason more than one are allowed in an equation.

4.1 Pattern-Matching Semantics

So far we have discussed how individual patterns are matched, how some are refutable, some are irrefutable, etc. But what drives the overall process? In what order are the matches attempted? What if none succeed? These are the questions addressed in this section.

---

[10] The Report calls these *variables*.

[11] Another advantage to doing this is that a naive implementation might completely reconstruct `x:xs` rather than re-use the value being matched against.

A particular match of a pattern to a value can actually yield one of three results: *fffffflurff*; *suffffffss* (returning a binding for each formal parameter in the pattern); or *ffffvffrffffnffff* (i.e. nontermination). The matching process itself occurs "top-down, left-to-right." Failure of a pattern anywhere in one equation results in failure of the whole equation, and the next equation is then tried. If all equations fail, the value of the function application is $\perp$, and results in a run-time error.

For example, if `[1,2]` is matched against `[0,bot]`, then `1` fails to match `0`, so the result is a failed match. But if `[1,2]` is matched against `[bot,0]`, then matching `1` against `bot` causes divergence (i.e. $\perp$).

The only other twist to this set of rules is that top-level patterns may also have a boolean *ffuffrff*, as in this definition of a function that forms an abstract version of a number's sign:

```
sign x |  x >  0        =   1
       |  x == 0        =   0
       |  x <  0        =  -1
```

Note that a sequence of guards may be provided for the same pattern; as with patterns, they are evaluated top-down, and the first that evaluates to `True` results in a successful match.

4.2 An Example

The pattern-matching rules can have subtle effects on the meaning of functions. For example, consider this definition of `take`:

```
take  0     _           =  []
take  _     []          =  []
take (n+1) (x:xs)       =  x : take n xs
```

and this slightly different version (the first 2 equations have been reversed):

```
take1  _     []         =  []
take1  0     _          =  []
take1 (n+1) (x:xs)      =  x : take1 n xs
```

Now note the following:

$$
\begin{array}{lcl}
\texttt{take 0 bot} & \Rightarrow & \texttt{[]} \\
\texttt{take1 0 bot} & \Rightarrow & \perp \\[1ex]
\texttt{take bot []} & \Rightarrow & \perp \\
\texttt{take1 bot []} & \Rightarrow & \texttt{[]}
\end{array}
$$

We see that `take` is "more defined" with respect to its second argument, whereas `take1` is more defined with respect to its first. It is difficult to say in this case which definition is better. Just remember that in certain applications, it may make a difference. (The Standard Prelude includes a definition corresponding to `take`.)

4.3 Case Expressions

Pattern matching provides a way to "dispatch control" based on structural properties of a value. However, in many circumstances we don't wish to define a *function* every time we need to do this, but so far we have only shown how to do pattern matching in function definitions. Haskell's *case expression* provides a way to solve this problem. Indeed, the meaning of pattern matching in function definitions is specified in the Report in terms of case expressions, which are considered more primitive. In particular, a function definition of the form:

$$\begin{array}{l}\texttt{f}\ p_{11}\ \ldots\ p_{1k}\ \texttt{=}\ e_1\\ \ldots\\ \texttt{f}\ p_{n1}\ \ldots\ p_{nk}\ \texttt{=}\ e_n\end{array}$$

where each p_{ij} is a pattern, is semantically equivalent to:

$$\begin{array}{ll}\texttt{f x1 x2 ... xk = case (x1, ..., xk) of} & (p_{11},\ \ldots,\ p_{1k})\ \texttt{->}\ e_1\\ & \ldots\\ & (p_{n1},\ \ldots,\ p_{nk})\ \texttt{->}\ e_n\end{array}$$

where the `xi` are new identifiers. (For a more general translation that includes guards, see §4.4.2.) For example, the definition of `take` given earlier is equivalent to:

```
take m ys               = case (m,ys) of
                            (0,_)       ->  []
                            (_,[])      ->  []
                            (n+1,x:xs)  ->  x : take n xs
```

A point not made earlier is that, for type correctness, the types of the right-hand sides of a case expression or set of equations comprising a function definition must all be the same; more precisely, they must all share a common principal type.

The pattern-matching rules for case expressions are the same as we have given for function definitions, so there is really nothing new to learn here, other than to note the convenience that case expressions offer. Indeed, there's one use of a case expression that is so common that it has special syntax: the *conditional expression*. In Haskell, conditional expressions have the familiar form:

$$\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3$$

which is really short-hand for:

$$\begin{array}{ll}\texttt{case}\ e_1\ \texttt{of} & \texttt{True}\ \ \texttt{->}\ e_2\\ & \texttt{False}\ \texttt{->}\ e_3\end{array}$$

From this expansion it should be clear that e_1 must have type `Bool`, and e_2 and e_3 must have the same (but otherwise arbitrary) type. In other words, `if_then_else_` when viewed as a function has type `Bool->a->a->a`.

4.4 Lazy Patterns

There is one other kind of pattern allowed in Haskell. It is called a *lazy pattern*, and has the form *~pat*. Lazy patterns are *irrefutable*: matching a value v against *~pat* always succeeds, regardless of *pat*. Operationally speaking, if an identifier in *pat* is later "used" on the right-hand-side, it will be bound to that portion of the value that would result if v were to successfully match *pat*, and $\perp$ otherwise.

Lazy patterns are useful in contexts where infinite lists are being defined recursively. For example, infinite lists are an excellent vehicle for writing *simulation* programs, and in this context the infinite lists are often called *streams*. Consider the simple case of simulating the interactions between a server process `server` and a client process `client`, where `client` sends a sequence of *requests* to `server`, and `server` replies to each request with some kind of *response*. This situation is shown pictorially in Figure 1b, just as we did with the Fibonacci example. (Note that `client` also takes an initial message as argument.) Using streams to simulate the message sequences, the Haskell code corresponding to this diagram is:

```
reqs                     = client init resps
resps                    = server reqs
```

These recursive equations are a direct lexical transliteration of the diagram.

Let us further assume that the structure of the server and client look something like this:

```
client init (resp:resps) = init : client (next resp) resps
server      (req:reqs)   = process req : server reqs
```

where we assume that `next` is a function that, given a response from the server, determines the next request, and `process` is a function that processes a request from the client, returning an appropriate response.

Unfortunately, this program has a serious problem: it will not produce any output! The problem is that `client`, as used in the recursive setting of `reqs` and `resps`, attempts a match on the response list before it has submitted its first request! In other words, the pattern matching is being done "too early." One way to fix this is to redefine `client` as follows:

```
client init resps         = init : client (next (head resps)) (tail resps)
```

Although workable, this solution does not read as well as that given earlier. A better solution is to use a lazy pattern:

```
client init ~(resp:resps) = init : client (next resp) resps
```

Because lazy patterns are irrefutable, the match will immediately succeed, allowing the initial request to be "submitted," in turn allowing the first response to be generated; the engine is now "primed," and the recursion takes care of the rest.

As an example of this program in action, if we define:

```
init                    = 0
next resp               = resp
process req             = req+1
```

then we see that:

```
take 10 reqs   ⇒   [0,1,2,3,4,5,6,7,8,9]
```

As another example of the use of lazy patterns, consider the definition of Fibonacci given earlier:

```
fib             = 1 : 1 : [ a+b | (a,b) <- zip fib (tail fib) ]
```

We might try rewriting this using an as-pattern:

```
fib@(1:tfib)    = 1 : 1 : [ a+b | (a,b) <- zip fib tfib ]
```

This version of `fib` has the (small advantage) of not using `tail` on the right-hand side, since it is available in "destructured" form on the left-hand side as `tfib`.

[This kind of equation is called a *pfttffrn ffffnffffnff* because it is a top-level equation in which the entire left-hand side is a pattern; i.e. both `fib` and `tfib` become bound within the scope of the declaration.]

Now, using the same reasoning we did earlier, we should be led to believe that this program will not generate any output. Curiously, however, it *ffoffs*, and the reason is simple: in Haskell, pattern bindings are assumed to have an implicit `~` in front of them, reflecting the most common behavior expected of pattern bindings, and avoiding some anomalous situations which are beyond the scope of this tutorial. Thus we see that lazy patterns play an important role in Haskell, if only implicitly.

4.5 Lexical Scoping and Nested Forms

It is often desirable to create a nested scope within an expression, for the purpose of creating local bindings not seen elsewhere—i.e. some kind of "block-structuring" form. In Haskell there are two ways to achieve this:

Let Expressions. Haskell's *lfft ffxprffssffons* are useful whenever a nested set of bindings is required. As a simple example, consider:

```
let y   = a*b
    f x = (x+y)/y
in f c + f d
```

The set of bindings created by a `let` expression is *mutufflly rffffursffvff*, and pattern bindings are treated as lazy patterns (i.e. they carry an implicit `~`). The only kind of declarations permitted are *typff sffffnffturffs*, *ffunfftffon ffffnffffnffs*, and *pfttffrn ffffnffffnffs*.

Where Clauses. Sometimes it is convenient to scope bindings over several guarded equations, which requires a *where clause*:

```
f x y  |  y>z           =  ...
       |  y==z          =  ...
       |  y<z           =  ...
                        where z = x*x
```

Note that this cannot be done with a `let` expression, which only scopes over the expression which it encloses. A `where` clause is only allowed at the top level of a set of equations or case expression. The same properties and constraints on bindings in `let` expressions apply to those in `where` clauses.

These two forms of nested scope seem very similar, but remember that a `let` expression is an *expression*, whereas a `where` clause is not—it is part of the syntax of function declarations and case expressions.

4.6 Layout

The reader may have been wondering how it is that Haskell programs avoid the use of semicolons, or some other kind of line terminator, to mark the end of equations, declarations, etc. For example, consider this `let` expression from the last section:

```
let y   = a*b
    f x = (x+y)/y
in f c + f d
```

How does the parser know not to parse this as:

```
let y   = a*b f
    x   = (x+y)/y
in f c + f d
```

?

The answer is that Haskell uses a two-dimensional syntax called *layout* that essentially relies on declarations being "lined up in columns." In the above example, note that `y` and `f` begin in the same column. The rules for layout are spelled out in detail in the Report (§1.5,§B.4), but in practice, use of layout is rather intuitive. Just remember two things:

First, the next character following any of the keywords `where`, `let`, or `of` is what determines the starting column for the declarations in the where, let, or case expression being written (the rule also applies to `where` used in the class and instance declarations to be defined in Section 5). Thus we can begin the declarations on the same line as the keyword, the next line, etc.

Second, just be sure that the starting column is further to the right than the starting column associated with the immediately surrounding clause (otherwise it would be ambiguous). The "termination" of a declaration happens when something appears at or to the left of the starting column associated with that binding form.[12]

Layout is actually shorthand for an *explicit* grouping mechanism, which deserves mention because it can be useful under certain circumstances. The `let` example above is equivalent to:

```
let { y   = a*b;
      f x = (x+y)/y }
in f c + f d
```

Note the explicit curly braces and semicolons. One way in which this explicit notation is useful is when more than one declaration is desired on a line; for example, this is a valid expression:

```
let y   = a*b;  z = a/b
    f x = (x+y)/z
in f c + f d
```

For another example of the expansion of layout into explicit delimiters, see §1.5.

The use of layout greatly reduces the syntactic clutter associated with declaration lists, thus enhancing readability. It is easy to learn, and its use is encouraged.

5 Type Classes, Overloading, and “OOP”

There is one final feature of Haskell's type system that sets it apart from other languages, and is probably the most innovative aspect of Haskell's design. The kind of polymorphism that we have talked about so far is commonly called *parametric* polymorphism. There is another kind called *ad hoc* polymorphism, better known as *overloading*. Here are some examples of ad hoc polymorphism:

- The literals `1`, `2`, etc. are often used to represent both fixed and arbitrary precision integers.
- Numeric operators such as `+` are often defined to work on many different kinds of numbers.
- The equality operator (`==` in Haskell) usually works on numbers and many other (but not all) types.

Note that these overloaded behaviors are *different* for each type (in fact the behavior is sometimes undefined, or error), whereas in parametric polymorphism the type truly does not matter (`fringe`, for example, really doesn't care what kind of elements are found in the leaves of a tree). In Haskell, *type classes* provide a structured way to control ad hoc polymorphism, or overloading.

Let's start with a simple, but important, example: *equality*. There are many types for which we would like equality defined, but some for which we would not. For example, comparing the equality of functions is generally considered computationally intractable, whereas we often want to compare

---

[12] Haskell observes the convention that tabs count as 8 blanks; thus care must be taken when using an editor which may observe some other convention.

two lists for equality.[13] To highlight the issue, consider this definition of the function `elem` which tests for membership in a list:

```
x `elem`  []            = False
x `elem` (y:ys)         = x==y || (x `elem` ys)
```

[For the stylistic reason we discussed in Section 3.1, we have chosen to define `elem` in infix form. `==` and `||` are the infix operators for equality and logical or, respectively.]

Intuitively speaking, the type of `elem` "ought" to be: `a->[a]->Bool`. But this would imply that `==` has type `a->a->Bool`, even though we just said that we don't expect `==` to be defined for all types.

Furthermore, as we have noted earlier, even if `==` *were* defined on all types, comparing two lists for equality is *very different* from comparing two integers. In this sense, we expect `==` to be *overloaded* to carry on these various tasks.

Type classes conveniently solve both of these problems by allowing us to declare which types are *instances* of which class, and to provide definitions of the overloaded *operations* associated with a class. For example, let's define a type class containing an equality operator:

```
class Eq a where
  (==)                  :: a -> a -> Bool
```

Here `Eq` is the name of the class being defined, and `==` is the single operation in the class. This declaration may be read "a type `a` is an instance of the class `Eq` if there is an (overloaded) operation `==`, of the appropriate type, defined on it." (Note that `==` is only defined on pairs of objects of the same type.)

The constraint that a type `a` must be an instance of the class `Eq` is written `Eq a`. Thus `Eq a` is not a type expression, but rather it expresses a constraint on a type, and is called a *context*. Contexts are appended to the front of type expressions. For example, the effect of the above class declaration is to assign the following type to `==`:

```
(==)                    :: (Eq a) => a -> a -> Bool
```

This should be read, "For every type `a` that is an instance of the class `Eq`, `==` has type `a->a->Bool`." This is the type that would be used for `==` in the `elem` example, and indeed the constraint imposed by the context propagates to the principal type for `elem`:

```
elem                    :: (Eq a) => a -> [a] -> Bool
```

This should be read, "For every type `a` that is an instance of the class `Eq`, `elem` has type `a->[a]->Bool`." This is just what we want—it expresses the fact that `elem` is not defined on *all* types, just those for which we know how to compare its elements for equality.

---

[13] The kind of equality we are referring to here is "value equality," and opposed to the "pointer equality" found, for example, in Lisp's `eq`. Pointer equality is not referentially transparent, and thus does not sit well in a purely functional language.

So far so good. But how do we specify which types are instances of the class `Eq`, and the actual behavior of `==` on each of those types? This is done with an *instance declaration*. For example:

```
instance Eq Int where
  x == y                =  intEq x y
```

The definition of `==` is called a *method*. `intEq` happens to be the primitive function that compares integers for equality, but in general any valid expression is allowed on the right-hand side, just as for any other function definition. The overall declaration is essentially saying: "The type `Int` is an instance of the class `Eq`, and here is the definition of the method corresponding to the operation `==`." Given this declaration, we can now compare fixed precision integers for equality using `==`. Similarly:

```
instance Eq Float where
  x == y                =  floatEq x y
```

allows us to compare floating point numbers using `==`.

Recursive types such as `Tree` defined earlier can also be handled:

```
instance (Eq a) => Eq (Tree a) where
  Leaf a         == Leaf b          =  a == b
  (Branch l1 r1) == (Branch l2 r2)  =  (l1==l2) && (r1==r2)
  _              == _               =  False
```

Note the context `Eq a` in the first line—this is necessary because the elements in the leaves (of type `a`) are compared for equality in line 2. The additional constraint is essentially saying that we can compare trees of `a`'s for equality as long as we know how to compare `a`'s for equality. If the context were omitted, a static type error would result.

The Haskell Report, especially the Standard Prelude, contains a wealth of useful examples of type classes. Indeed, a class `Eq` is defined that is slightly larger than the one defined earlier:

```
class  Eq a  where
  (==), (/=)            :: a -> a -> Bool
  x /= y                =  not (x == y)
```

This is an example of a class with *two* operations, one for equality, the other for inequality. It also demonstrates the use of a *default method*, in this case for the inequality operation `/=`. If a method for a particular operation is omitted in an instance declaration, then the default one defined in the class declaration, if it exists, is used instead. For example, the three instances of `Eq` defined earlier will work perfectly well with the above class declaration, yielding just the right definition of inequality that we want: the logical negation of equality.

Haskell also supports a notion of *class inclusion*. For example, we may wish to define a class `Ord` which *inherits* all of the operations in Eq, but in addition has a set of comparison operations and minimum and maximum functions:

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

Note the context in the `class` declaration. We say that `Eq` is a *superclass* of `Ord` (conversely, `Ord` is a *subclass* of `Eq`), and any instance of `Ord` must also be an instance of `Eq`. (In the next Section we give a fuller definition or `Ord` taken from the Standard Prelude.)

One benefit of such class inclusions is shorter contexts: A type expression for a function that uses operations from both the `Eq` and `Ord` classes can use the context `(Ord a)`, rather than `(Eq a, Ord a)`, since `Ord` "implies" `Eq`. More importantly, methods for subclass operations can assume the existence of methods for superclass operations. For example, the `Ord` declaration in the Standard Prelude contains this default method for `(<)`:

```
    x < y               =  x <= y && x /= y
```

As an example of the use of `Ord`, the principal typing of `quicksort` defined in Section 2.5.1 is:

```
quicksort               ::  (Ord a) => [a] -> [a]
```

In other words, `quicksort` only operates on lists of values of *ordered* types. This typing for `quicksort` arises because of the use of the comparison operators `<` and `>=` in its definition.

Haskell also permits *multiple inheritance*, since classes may have more than one superclass. Name conflicts are avoided by having the constraint that a particular operation can be a member of at most one class in any given scope.

Contexts are also allowed in `data` declarations; see §4.2.1.

A Different Perspective. Before going on to further examples of the use of type classes, it is worth pointing out two other viewpoints of Haskell's type classes. The first is by analogy to object-oriented programming (OOP). In the following general statement about OOP, simply substituting *type class* for class, and *type* for object, yields a valid summary of Haskell's type class mechanism:

"*Classes* capture common sets of *operations*. A particular *object* may be an *instance* of a class, and will have a *method* corresponding to each operation. Classes may be arranged hierarchically, forming notions of *superclasses* and *subclasses*, and permitting *inheritance* of operations/methods. A *default method* may also be associated with an operation."

In *contrast* to OOP, it should be clear that types are not objects, and in particular there is no notion of an object's or type's internal mutable state. An advantage over OOP is that methods in Haskell are completely type-safe: any attempt to apply a method to a value whose type is not in the required class will be detected at compile time instead of at runtime. In other words, methods are not "looked up" at runtime but are simply passed as higher-order functions.

A different perspective can be gotten by considering the relationship between parametric and ad hoc polymorphism. We have shown how parametric polymorphism is useful in defining families of types by universally quantifying over *ffll* types. Sometimes, however, that universal quantification is too broad—we wish to quantify over some smaller set of types, such as those types whose elements can be compared for equality. Type classes can be seen as providing a structured way to do just this. Indeed, we can think of parametric polymorphism as a kind of overloading too! It's just that the overloading occurs implicitly over all types instead of a constrained set of types (i.e. a type class).

In the remainder of this section we introduce the several predefined *stffnffffrff* type classes in Haskell.

5.1 Equality and Ordered Classes

Haskell's standard classes form the somewhat frightening inclusion structure shown in Figure 2. At the top of the figure, we see `Eq` with its subclass `Ord` below it. These were defined in the previous section. Continuing down, we encounter two subclasses of `Ord`, `Enum` and `Ix`.

5.2 Enumeration and Index Classes

`Enum` has a set of operations that underlie the syntactic sugar of arithmetic sequences; for example, the arithmetic sequence expression `[1,3..]` stands for `enumFromThen 1 3` (see §3.9 for the formal translation). Arithmetic sequences are ordered, so naturally, `Enum` is a subclass of `Ord`. We can now see that arithmetic sequence expressions can be used to generate lists of any type that is an instance of `Enum`. This includes not only most numeric types, but also `Char`, so that, for instance, `['a'..'z']` denotes the list of lower-case letters in alphabetical order. Furthermore, user-defined enumerated types like `Color` can easily be given `Enum` instance declarations. If so:

```
[Red..Violet]   ⇒   [Red, Green, Blue, Indigo, Violet]
```

Note that such a sequence is *ffrffttffmffttffff* in the sense that the increment between values is constant, even though the values are not numbers.

The other immediate standard subclass of `Ord` is `Ix`, which is the class of types that can be used as array indices. Again, it is natural that `Ix` should be a subclass of `Ord`, since we expect the elements of an array to have an index order. We deal with class `Ix` in Section 6.9.

5.3 Text Class

The instances of class `Text` are those types that can be converted to character strings (typically for I/O) and back; thus, this class provides operations for parsing character strings to obtain the values they may represent and for producing the canonical textual representation of a printable value. As these primitive operations are somewhat esoteric, let's begin with one of the higher-level functions that is defined in terms of them:

```
show                    :: (Text a) => a -> String
```

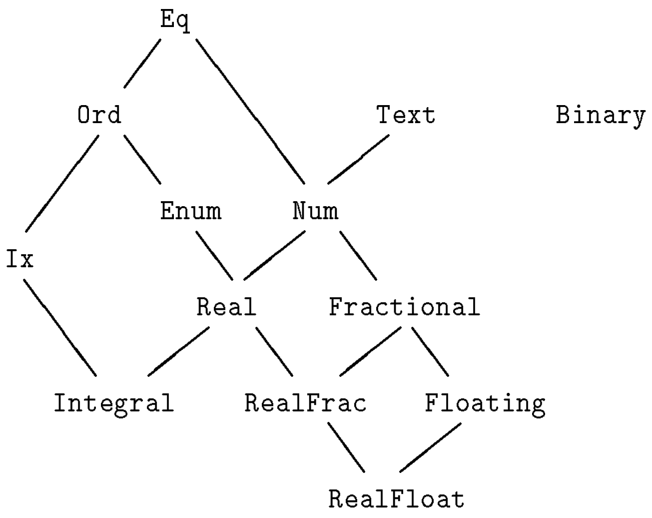

Figure 2: Haskell's Standard Type Class Hierarchy

Naturally enough, `show` takes any value of an appropriate type and returns its representation as a character string (list of characters), as in `show (2+2)`, which results in `"4"`. This is fine as far as it goes, but we typically need to produce more complex strings that may have the representations of many values in them, as in

```
"The sum of " ++ show x ++ " and " ++ show y ++ " is " ++ show (x+y) ++ "."
```

and after a while, all that concatenation gets to be a bit inefficient. Specifically, let's consider a function to represent the binary trees of Section 2.3 as a string, with suitable markings to show the nesting of subtrees and the separation of left and right branches (provided the element type is representable as a string):

```
showTree                :: (Text a) => Tree a -> String
showTree (Leaf x)       =  show x
showTree (Branch l r)   =  "<" ++ showTree l ++ "|" ++ showTree r ++ ">"
```

Because `(++)` has time complexity linear in the length of its left argument, `showTree` is quadratic in the size of the tree.

To restore linear complexity, the function `shows` is provided:

```
shows                   :: (Text a) => a -> String -> String
```

`shows` takes a printable value and a string and returns that string with the value's representation concatenated at the front. The second argument serves as a sort of string *accumulator*, and `show` can now be defined as `shows` with the null accumulator:

```
show x                   =  shows x ""
```

We can use `shows` to define a more efficient version of `showTree`, which also has a string accumulator argument:

```
showsTree                :: (Text a) => Tree a -> String -> String
showsTree (Leaf x) s     =  shows x s
showsTree (Branch l r) s=  '<' : showsTree l ('|' : showsTree r ('>' : s))
```

This solves our efficiency problem (`showsTree` has linear complexity), but the presentation of this function (and others like it) can be improved. First, let's create a type synonym:

```
type ShowS               =  String -> String
```

This is the type of a function that returns a string representation of something followed by an accumulator string. Second, we can avoid carrying accumulators around, and also avoid amassing parentheses at the right end of long constructions, by using functional composition:

```
showsTree                :: (Text a) => Tree a -> ShowS
showsTree (Leaf x)       =  shows x
showsTree (Branch l r)   =  ('<':) . showsTree l . ('|':) . showsTree r . ('>':)
```

Something more important than just tidying up the code has come about by this transformation: We have raised the presentation from an *offfffffft lffvffl* (in this case, strings) to a *ffunfftffon lffvffl.* We can think of the typing as saying that `showsTree` maps a tree into a *sffowffnff ffunfftffon.* Functions like `('<' :)` or `("a string" ++)` are primitive showing functions, and we build up more complex functions by composition.

Now that we can turn trees into strings, let's turn to the inverse problem. The basic idea is a *pffrsffr* for a type `a`, which is a function that takes a string and returns a list of `(a, String)` pairs[7]. The Standard Prelude provides a type synonym for such functions:

```
type ReadS a             =  String -> [(a,String)]
```

Normally, a parser returns a singleton list, containing a value of type `a` that was read from the input string and the remaining string that follows what was parsed. If no parse was possible, however, the result is the empty list, and if there is more than one possible parse (an ambiguity), the resulting list contains more than one pair. The standard function `reads` is a parser for any instance of `Text`:

```
reads                    :: (Text a) => ReadS a
```

We can use this function to define a parsing function for the string representation of binary trees produced by `showsTree`. List comprehensions give us a convenient idiom for constructing such parsers:

```
readsTree               :: (Text a) => ReadS (Tree a)
readsTree ('<':s)       =  [(Branch l r, u) | (l, '|':t) <- readsTree s,
                                              (r, '>':u) <- readsTree t ]
readsTree s             =  [(Leaf x, t)     | (x,t)      <- reads s]
```

Let's take a moment to examine this function definition in detail. There are two main cases to consider: If the first character of the string to be parsed is `'<'`, we should have the representation of a branch; otherwise, we have a leaf. In the first case, calling the rest of the input string following the opening angle bracket `s`, any possible parse must be a tree `Branch l r` with remaining string `u`, subject to the following conditions:

1. The tree `l` can be parsed from the beginning of the string `s`.
2. The string remaining (following the representation of `l`) begins with `'|'`. Call the tail of this string `t`.
3. The tree `r` can be parsed from the beginning of `t`.
4. The string remaining from *tfffft* parse begins with `'>'`, and `u` is the tail.

Notice the expressive power we get from the combination of pattern matching with list comprehension: The form of a resulting parse is given by the main expression of the list comprehension, the first two conditions above are expressed by the first generator ("`(l, '|':t)` is drawn from the list of parses of `s`."), and the remaining conditions are expressed by the second generator.

The second defining equation above just says that to parse the representation of a leaf, we parse a representation of the element type of the tree and apply the constructor `Leaf` to the value thus obtained.

We'll accept on faith for the moment that there is a `Text` instance of `Int` (among many other types), providing a `reads` that behaves as one would expect, e.g.:

```
(reads "5 golden rings") :: [(Int,String)]    ⇒    [(5, " golden rings")]
```

With this understanding, the reader should verify the following evaluations:

```
readsTree "<1|<2|3>>"     ⇒     [(Branch (Leaf 1) (Branch (Leaf 2) (Leaf 3)), "")]
readsTree "<1|2"          ⇒     []
```

There are a couple of shortcomings in our definition of readsTree. One is that the parser is quite rigid, allowing no white space before or between the elements of the tree representation; the other is that the way we parse our punctuation symbols is quite different from the way we parse leaf values and subtrees, this lack of uniformity making the function definition harder to read. We can address both of these problems by using the lexical analyzer provided by the Standard Prelude:

```
lex                     :: ReadS String
```

`lex` normally returns a singleton list containing a pair of strings: the first lexeme in the input string and the remainder of the input. The lexical rules are those of Haskell programs, including comments, which `lex` skips, along with whitespace. If the input string is empty or contains only whitespace and comments, `lex` returns `[("","")]`; if the input is not empty in this sense, but also does not begin with a valid lexeme after any leading whitespace and comments, `lex` returns `[]`.

Using the lexical analyzer, our tree parser now looks like this:

```
readsTree               :: (Text a) => ReadS (Tree a)
readsTree s             =  [(Branch l r, x) | ("<", t) <- lex s,
                                              (l,   u) <- readsTree t,
                                              ("|", v) <- lex u,
                                              (r,   w) <- readsTree v,
                                              (">", x) <- lex w         ]
                           ++
                           [(Leaf x, t)     | (x,   t) <- reads s       ]
```

We may now wish to use `readsTree` and `showsTree` to declare `(Text a) => Tree a` an instance of `Text`. This would allow us to use the generic overloaded functions from the Prelude to parse and display trees. Moreover, we would automatically then be able to parse and display many other types containing trees as components, for example, `[Tree Int]`. As it turns out, `readsTree` and `showsTree` are of almost the right types to be `Text` methods, needing only the addition of an extra parameter each that has do do with parenthesization of forms with infix constructors. We refer the interested reader to §E.2 for details.

We can test such a `Text` instance by applying `(read . show)` (which should be the identity) to some trees, where `read` is a specialization of `reads`:

```
read                    :: (Text a) => String -> a
```

This function fails if there is not a unique parse or if the input contains anything more than a representation of one value of type `a` (and possibly, comments and whitespace).

5.4 Derived Instances

Recall the `Eq` instance for trees we presented in Section 5; such a declaration is simple—and boring—to produce: We require that the element type in the leaves be an equality type; then, two leaves are equal iff they contain equal elements, and two branches are equal iff their left and right subtrees are equal, respectively. Any other two trees are unequal:

```
instance  (Eq a) => Eq (Tree a)  where
    (Leaf x)     == (Leaf y)         =  x == y
    (Branch l r) == (Branch l' r')   =  l == l' && r == r'
    _            == _                =  False
```

Fortunately, we don't need to go through this tedium every time we need equality operators for a new type; the `Eq` instance can be *derived automatically* from the `data` declaration if we so specify:

```
data  Tree a            =  Leaf a | Branch (Tree a) (Tree a)  deriving Eq
```

The `deriving` clause implicitly produces an `Eq` instance declaration just like the one in Section 5. Instances of `Ord`, `Enum`, `Ix`, and `Text` can also be generated by the `deriving` clause. [More than one class name can be specified, in which case the list of names must be parenthesized and the names, separated by commas.]

The derived `Ord` instance for `Tree` is slightly more complicated than the `Eq` instance:

```
instance  (Ord a) => Ord (Tree a)  where
    (Leaf _)     <= (Branch _)      =  True
    (Leaf x)     <= (Leaf y)        =  x <= y
    (Branch _)   <= (Leaf _)        =  False
    (Branch l r) <= (Branch l' r')  =  l == l' && r <= r' || l <= l'
```

This specifies a *lexicographic* order: Constructors are ordered by the order of their appearance in the `data` declaration, and the arguments of a constructor are compared from left to right. Recall that the built-in list type is semantically equivalent to an ordinary two-constructor type. In fact, this is the full declaration:

```
data [a]        = [] | a : [a] deriving (Eq, Ord)     -- pseudocode
```

(Lists also have a `Text` instance, which is not derived.) The derived `Eq` and `Ord` instances for lists are the usual ones; in particular, character strings, as lists of characters, are ordered as determined by the underlying `Char` type, with an initial substring comparing less than a longer string; for example, `"cat" < "catalog"`.

In practice, `Eq` and `Ord` instances are almost always derived, rather than user-defined. In fact, we should provide our own definitions of equality and ordering predicates only with some trepidation, being careful to maintain the expected algebraic properties of equivalence relations and partial or total orders. An intransitive `(==)` predicate, for example, could be disastrous, confusing readers of the program and confounding manual or automatic program transformations that rely on the `(==)` predicate's being an approximation to definitional equality. Nevertheless, it is sometimes necessary to provide `Eq` or `Ord` instances different from those that would be derived; probably the most important example is that of an abstract data type in which different concrete values may represent the same abstract value.

An enumerated type can have a derived `Enum` instance, and here again, the ordering is that of the constructors in the `data` declaration. For example:

```
data Day = Sunday | Monday | Tuesday | Wednesday
         | Thursday | Friday | Saturday         deriving (Eq, Ord, Enum)
```

```
[Wednesday..Friday]      ⇒   [Wednesday, Thursday, Friday]
[Monday, Wednesday ..]   ⇒   [Monday, Wednesday, Friday]
```

An enumerated type or tuple (single-constructor) type all of whose component types have `Ix` instances can have a derived `Ix` instance. (See Section §6.9.)

Derived `Text` instances are possible for enumerated types and other types all of whose component types also have `Text` instances. (`Text` instances for most of the standard types, but not functions, are provided by the Standard Prelude.) The textual representation defined by a derived `Text` instance is consistent with the appearance of constant Haskell expressions of the type in question. For example, if we add `Text` to the `deriving` clause for type `Day`, above, we obtain

```
show [Monday..Wednesday]   ⇒   "[Monday,Tuesday,Wednesday]"
```

5.5 Numbers

Haskell provides a rich collection of numeric types, based on those of Scheme[5], which in turn are based on Common Lisp[6]. (Those languages, however, are dynamically typed.) The standard types include fixed- and arbitrary-precision integers, ratios (rational numbers) formed from each integer type, and single- and double-precision real and complex floating-point. We outline here the basic characteristics of the numeric type class structure and refer the reader to §6.8 for details.

5.5.1 Numeric Class Structure

Returning to Figure 2, we notice that the numeric type classes (class `Num` and those that lie below it) account for more than half of the standard classes. We also note that `Num` is a subclass of `Eq`, but not of `Ord`; this is because the order predicates do not apply to complex numbers. The subclass `Real` of `Num`, however, is a subclass of `Ord` as well.

The `Num` class provides several basic operations common to all numeric types; these include, among others, addition, subtraction, negation, multiplication, and absolute value:

```
(+), (-), (*)          :: (Num a) => a -> a -> a
negate, abs            :: (Num a) => a -> a
```

[`negate` is the function applied by Haskell's only prefix operator, minus; we can't call it `(-)`, because that is the subtraction function, so this name is provided instead. For example, `-x*y` is equivalent to `negate (x*y)`. (Prefix minus has the same syntactic precedence as infix minus, which, of course, is lower than that of multiplication.)]

Note that `Num` does *not* provide a division operator; two different kinds of division operators are provided in two non-overlapping subclasses of `Num`:

The class `Integral` provides whole-number division and remainder operations, as well as the `even` and `odd` predicates. The standard instances of `Integral` are `Integer` (unbounded or mathematical integers, also known as "bignums") and `Int` (bounded, machine integers, with a range equivalent to at least 29-bit signed binary). A particular Haskell implementation might provide other integral types in addition to these. Note that `Integral` is a subclass of `Real`, rather than of `Num` directly; this means that there is no attempt to provide Gaussian integers.

All other numeric types fall in the class `Fractional`, which provides the ordinary division operator `(/)`. The further subclass `Floating` contains trigonometric, logarithmic, and exponential functions.

The `RealFrac` subclass of `Fractional` and `Real` provides a function `properFraction`, which decomposes a number into its whole and fractional parts, and a collection of functions that round to integral values by differing rules:

```
properFraction          :: (Fractional a, Integral b) => a -> (b,a)
truncate, round,
floor, ceiling:         :: (Fractional a, Integral b) => a -> b
```

The `RealFloat` subclass of `Floating` and `RealFrac` provides some specialized functions for efficient access to the components of a floating-point number, the *exponent* and *significand*. The standard types `Float` and `Double` fall in class `RealFloat`.

5.5.2 Constructed Numbers

Of the standard numeric types, `Int`, `Integer`, `Float`, and `Double` are primitive. The others are made from these by type constructors:

`Complex` is a type constructor that makes a complex type in class `Floating` from a `RealFloat` type:

```
data (RealFloat a) => Complex a = a :+ a  deriving (Eq, Text)
```

Notice the context `(RealFloat a) =>`, which restricts the argument type; thus, the standard complex types are `Complex Float` and `Complex Double`. We can also see from the `data` declaration that a complex number is written x `:+` y; the arguments are the cartesian real and imaginary parts, respectively. Since `:+` is a data constructor, we can use it in pattern matching:

```
conjugate               :: (RealFloat a) => Complex a -> Complex a
conjugate (x:+y)        =  x :+ (-y)
```

Similarly, the type constructor `Ratio` makes a rational type in class `RealFrac` from an instance of `Integral`. (`Rational` is a type synonym for `Ratio Integer`.) `Ratio`, however, is an abstract type constructor, being defined in the Prelude module `PreludeRatio`, from which it is exported without its data constructor; thus, ratios cannot be pattern-matched. Instead, a function to form a ratio from two integers is provided, as well as component extraction functions:

```
(%)                        :: (Integral a) => a -> a -> Ratio a
numerator, denominator     :: (Integral a) => Ratio a -> a
```

Why the difference? Complex numbers in cartesian form are unique—there are no nontrivial identities involving `:+`. On the other hand, ratios are not unique, but have a canonical (reduced) form that the implementation of the abstract data type must maintain; it is not necessarily the case, for instance, that `numerator (x%y)` is equal to `x`, although the real part of `x:+y` is always `x`.

5.5.3 Numeric Coercions and Overloaded Literals

The Standard Prelude provides several overloaded functions that serve as explicit coercions:

```
fromInteger             :: (Num a) => Integer -> a
fromRational            :: (Fractional a) => Rational -> a
toInteger               :: (Integral a) => a -> Integer
toRational              :: (RealFrac a) => a -> Rational
fromIntegral            :: (Integral a, Num b) => a -> b
fromRealFrac            :: (RealFrac a, Fractional b) => a -> b

fromIntegral            =  fromInteger . toInteger
fromRealFrac            =  fromRational . toRational
```

Two of these are implicitly used to provide overloaded numeric literals: An integer numeral (without a decimal point) is actually equivalent to an application of `fromInteger` to the value of the numeral as an `Integer`. Similarly, a floating numeral (with a decimal point) is regarded as an application of `fromRational` to the value of the numeral as a `Rational`. Thus, `7` has the type `(Num a) => a`, and `7.3` has the type `(Fractional a) => a`. This means that we can use numeric literals in generic numeric functions, for example:

```
halve                   :: (Fractional a) => a -> a
halve x                 =  x * 0.5
```

This rather indirect way of overloading numerals has the additional advantage that the method of interpreting a numeral as a number of a given type can be specified in an `Integral` or `Fractional` instance declaration (since `fromInteger` and `fromRational` are operators of those classes, respectively). For example, the `Num` instance of `(RealFloat a) => Complex a` contains this method:

```
fromInteger x           =  fromInteger x :+ 0
```

This says that a `Complex` instance of `fromInteger` is defined to produce a complex number whose real part is supplied by an appropriate `RealFloat` instance of `fromInteger`. In this manner, even user-defined numeric types (say, quaternions) can make use of overloaded numerals.

As another example, recall our first definition of `succ` from Section 2:

```
succ :: Int -> Int
succ n = n+1
```

Ignoring the type signature, the most general type of `succ` is `(Num a) => a->a`. The explicit type signature is legal, however, since it is *more specific* than the principal type (a more general type signature would cause a static error). The type signature has the effect of restricting `succ`'s type, and in this case would cause something like `succ (1::Float)` to be ill-typed.

5.5.4 Default Numeric Types

Consider the following function definition:

```
rms                     :: (Floating a) => a -> a -> a
rms x y                 =  sqrt ((x^2 + y^2) * 0.5)
```

The exponentiation function `(^)` (one of three different standard exponentiation operators with different typings, see §6.8.5) has the type `(Num a, Integral b) => a -> b -> a`, and since 2 has the type `(Num a) => a`, the type of `x^2` is `(Num a, Integral b) => a`. This is a problem; there is no way to resolve the overloading associated with the type variable `b`, since it is in the context, but has otherwise vanished from the type expression. Essentially, the programmer has specified that `x` should be squared, but has not specified whether it should be squared with an `Int` or an `Integer` value of two. Of course, we can fix this:

```
rms x y                 =  sqrt ((x ^ (2::Int) + y ^ (2::Int)) * 0.5)
```

It's obvious that this sort of thing will soon grow tiresome, however.

In fact, this kind of overloading ambiguity is not restricted to numbers:

```
show (read "xyz")
```

As what type is the string supposed to be read? This is more serious than the exponentiation ambiguity, because there, any `Integral` instance will do, whereas here, very different behavior can be expected depending on what instance of `Text` is used to resolve the ambiguity.

Because of the difference between the numeric and general cases of the overloading ambiguity problem, Haskell provides a solution that is restricted to numbers: Each module may contain a *default declaration,* consisting of the keyword `default` followed by a parenthesized, comma-separated list of numeric monotypes (types with no variables). When an ambigous type variable is discovered (such as `b`, above), if at least one of its classes is numeric and all of its classes are standard, the default list is consulted, and the first type from the list that will satisfy the context of the type variable is used. For example, if the default declaration `default (Int, Float)` is in effect, the ambiguous exponent above will be resolved as type `Int`. (See §4.3.4 for more details.)

The "default default" is `(Int, Double)`, but `(Integer, Rational, Double)` may also be appropriate. Very cautious programmers may prefer `default ()`, which provides no defaults.

6 Modules

At the top level, a Haskell program consists of a collection of *modules*. A module in Haskell serves the dual purpose of controlling name-spaces and creating abstract data types.

The top level of a module contains any of the various declarations we have discussed: fixity declarations, data and type declarations, class and instance declarations, type signatures, function definitions, and pattern bindings. Except for the fact that fixity declarations and import declarations (to be described shortly) must appear first, the declarations may appear in any order (the top-level scope is mutually recursive).

Haskell's module design is relatively conservative: The namespace of modules is completely flat, and modules are in no way "first-class." Module names are alphanumeric and must begin with an uppercase letter. There is no formal connection between a Haskell module and the file system that would (typically) support it. In particular, there is no connection between module names and file names, and more than one module could conceivably reside in a single file (one module may even span several files). Of course, a particular implementation will most likely adopt conventions that make the connection between modules and files more stringent.

Technically speaking, a module is really just one big declaration which begins with the keyword `module`; here's an example for a module whose name is `Tree`:

```
module Tree ( Tree(Leaf,Branch), fringe ) where

data Tree a                = Leaf a | Branch (Tree a) (Tree a)

fringe :: Tree a -> [a]
fringe (Leaf x)            = [x]
fringe (Branch left right) = fringe left ++ fringe right
```

The type `Tree` and the function `fringe` should be familiar; they were given as examples in Section 2.3. [Because of the `where` keyword, layout is active at the top level of a module, and thus the declarations must all line up in the same column (typically the first). Also note that the module name is the same as that of the type; this is allowed.]

This module explicitly *exports* `Tree`, `Leaf`, `Branch`, and `fringe`. If the export list following the `module` keyword is omitted, *all* of the names bound at the top level of the module would be exported. (In the above example everything is explicitly exported, so the effect would be the same.) Note that the name of a type and its constructors must be grouped together, as in `Tree(Leaf,Branch)`. As short-hand, we could also write `Tree(..)` (exporting a subset of the constructors is not allowed).

The `Tree` module may now be *imported* into some other module:

```
module Main (main) where
import Tree ( Tree(Leaf,Branch), fringe )

main = putText (fringe (Branch (Leaf 1) (Leaf 2)))
```

The various items being imported into and exported out of a module are called *entities*. Note the explicit import list in the import declaration; omitting it would cause all entities exported from `Tree` to be imported.

6.1 Original Names and Renaming

Abstractly associated with every entity is its *original name*—a pair consisting of the name of the entity and the name of the module in which it was originally defined. This concept is useful in that it allows an entity with the same name to be imported from a variety of different modules, without conflict, as long as the original name is the same.

Entities being imported can also be *renamed*; for example:

```
import Tree ( Tree(Leaf,Branch), fringe )
       renaming (Leaf to Root, Branch to Twig)
```

6.2 Interfaces and Implementations

The modules `Tree` and `Main` shown above are in actuality module *implementations*—they contain all of the Haskell code to completely define a module. The information passed between modules during export and import, however, is more abstract, and is captured formally in a module *interface*. The interface for a module `M` contains all of the information about the entities exported from `M` that is needed to ensure proper typing of some other module `N` that may import `M`. The proper interfaces for the module `Tree` is:

```
interface Tree ( Tree(Leaf,Branch), fringe ) where
data Tree a = Leaf a | Branch (Tree a) (Tree a)
fringe :: Tree a -> [a]
```

Normally the user need not be concerned about module interfaces, since they are usually derived automatically from the module implementation. On the other hand, it is not unusual to want to statically debug a module implementation that imports some other module whose implementation does not exist yet; being able to explicitly write the interface for this imported module would permit this. Indeed, interfaces support top-down, successive refinement, software development methodologies, in which *all* interfaces are ideally written *first*.

6.3 Abstract Data Types

Aside from controlling namespaces, modules provide the only way to build abstract data types (ADTs) in Haskell. For example, the characteristic feature of an ADT is that the *representation type* is *hidden*; all operations on the ADT are done at an abstract level which does not depend on the representation. For example, although the `Tree` type is simple enough that we might not normally make it abstract, a suitable ADT for it might include the following operations, which we specify in the form of an interface:

```
interface TreeADT (Tree, leaf, branch, cell,
                       left, right, isLeaf, isBranch) where
data Tree a
leaf                     :: a -> Tree a
branch                   :: Tree a -> Tree a -> Tree a
cell                     :: Tree a -> a
left, right              :: Tree a -> Tree a
isLeaf                   :: Tree a -> Bool
```

A module implementation supporting this is:

```
module TreeADT (Tree, leaf, branch, cell,
                left, right, isLeaf) where

data Tree a              = Leaf a | Branch (Tree a) (Tree a)

leaf                     = Leaf
branch                   = Branch
cell  (Leaf a)           = a
left  (Branch l r)       = l
right (Branch l r)       = r
isLeaf   (Leaf _)        = True
isLeaf   _               = False
```

Note in the export list that the type name `Tree` appears alone (i.e. without its constructors); similarly, in the interface the `data` declaration for `Tree` does not include the constructors. Thus `Leaf` and `Branch` are not exported, and the only way to build or take apart trees outside of the module is by using the various (abstract) operations. Of course, the advantage of this information hiding is that at a later time we could *ffffffnffff* the representation type without affecting users of the type.

6.4 Rules, Rules, and More Rules

Although Haskell's module system is relatively conservative, there are many rules concerning the import and export of values. Most of these are obvious—for instance, it is illegal to import two different entities having the same name into the same scope. Other rules are not so obvious—for example, an `instance` declaration can only appear in the module in which the corresponding `data` or `class` declaration appears. The reader should read the Report for details (§5).

7 Typing Pitfalls

This short section give an intuitive description of a few common problems that novices run into using Haskell's type system.

7.1 Let-Bound Polymorphism

Any language using the Hindley-Milner type system has what is called *let-bound polymorphism*, because identifiers not bound using a `let` or `where` clause (or at the top level of a module) are limited with respect to their polymorphism. In particular, a *lambda-bound* function (i.e., one passed as argument to another function) cannot be instantiated in two different ways. For example, this program is illegal:

```
let f g  =  (g [], g 'a')                        -- ill-typed expression
in f (\x->x)
```

because `g`, bound to a lambda abstraction whose principal type is `a->a`, is used within `f` in two different ways: once with type `[a]->[a]`, and once with type `Char->Char`.

7.2 Numeric Overloading

It is easy to forget at times that numerals are *overloaded*, and *not implicitly coerced* to the various numeric types, as in many other languages. More general numeric expressions sometimes cannot be quite so generic. A common numeric typing error is something like the following:

```
average xs              =  sum xs / length xs           -- Wrong!
```

`(/)` requires fractional arguments, but `length`'s result is an `Int`. The type mismatch must be corrected with an explicit coercion:

```
average                 :: (Fractional a) => [a] -> a
average xs              =  sum xs / fromIntegral (length xs)
```

7.3 The Monomorphism Restriction

The Haskell type system contains a restriction related to type classes that is not found in ordinary Hindley-Milner type systems: the *monomorphism restriction*. The reason for this restriction is related to a subtle type ambiguity and is explained in full detail in the Report (§4.5.4). A simpler explanation follows:

The monomorphism restriction says that any identifier bound by a pattern binding (which includes bindings to a single identifier), and having no explicit type signature, must be *monomorphic*.

An identifier is monomorphic if is either not overloaded, or is overloaded but is used in at most one specific overloading and is not exported.

Violations of this restriction result in a static type error. The simplest way to avoid the problem is to provide an explicit type signature. Note that *any* type signature will do (as long it is type correct).

A common violation of the restriction happens with functions defined in a higher-order manner, as in this definition of `sum` from the Standard Prelude:

```
sum                     =  foldl (+) 0
```

As is, this would cause a static type error. We can fix the problem by adding the type signature:

```
sum                     :: (Num a) => [a] -> a
```

Also note that this problem would not have arisen if we had written:

```
sum xs                  =  foldl (+) 0 xs
```

because the restriction only applies to pattern bindings.

8 Input/Output

The I/O system in Haskell is purely functional, yet has all of the expressive power found in conventional programming languages. To achieve this, Haskell uses a *monad* to integrate I/O operations into a purely functional context.

The I/O monad used by Haskell mediates between the *values* natural to a functional language and the *actions* which characterize I/O operations and imperative programming in general. The order of evaluation of expressions in Haskell is constrained only by data dependencies; an implementation normally has a great deal of freedom in choosing this order. Actions, however, must be ordered in a well-defined manner for program execution – and I/O in particular – to be meaningful. Haskell's I/O monad provides the user with a way to specify the sequential chaining of actions, and an implementation is obliged to preserve this order.

The term *monad* comes from a branch of mathematics known as *category theory*. From the perspective of a Haskell programmer, however, it's best to think of a monad as an *abstract datatype*. In the case of the I/O monad, the abstract values are the *actions* mentioned above. And there are operations to create actions – corresponding to conventional I/O operations – and operations (mentioned above) to sequentially compose actions – corresponding to sequencers (such as the semi-colon) in imperative languages. Finally, the *hidden implementation* can be thought of as the *system state*; i.e. the state of the world.

8.1 The I/O Monad

Operations within the I/O monad are characterized by the type `IO`. An I/O operation which returns a value of type `a` has the type `IO a`. For example, the `getChar` primitive has the following type:

```
getChar                 :: IO Char
```

since its effect is to return one character from standard input. When an I/O operation does not return a value, the type `IO ()` is used. (Recall that `()`, the unit type, is simply an empty tuple.)

I/O operations may also *rffffffffvff* arguments, as for example `putChar` does (whose effect is to display its character argument on standard output):

```
putChar                 :: Char -> IO ()
```

Linear sequencing of I/O operations is achieved using two functions. The `>>` function corresponds to simple sequential execution:

```
(>>)                    :: IO a -> IO b -> IO b
```

This performs the first I/O action, ignoring any result produced, and then performs the second action. The value produced by the second action is returned from `>>`. For example, two calls to `putChar` can be sequenced as follows:

```
putChar 'a' >> putChar 'b'
```

It is perhaps useful to think of `>>` as semi-colon in an imperative language.

When an I/O action returns a value, this value can be transmitted to a subsequent I/O operation using `>>=`:

```
(>>=)                   :: IO a -> (a -> IO b) -> IO b
```

Here, an action returning a value of type `a` is performed and this value is passed as an argument to the function which performs the next I/O action. For example, to read a character and then print it:

```
getChar >>= (\c -> putChar c)
```

Or, even simpler:

```
getChar >>= putChar
```

The `>>` function is actually a special case of `>>=`, as can be seen from its definition:

```
f >> g                  = f >>= (\_ -> g)
```

The I/O monad also provides a *null* I/O function, `return`. This function takes a value and creates an action which will return that value when executed. Its type is:

```
return ::               a -> IO a
```

It is important to remember that all of these I/O functions are no different from any other Haskell functions. For example, `>>` can be used as an argument to `foldr`:

```
putStr                  :: String -> IO ()
putStr s                = foldr (>>) (return ()) (map putChar s)
```

One of the differences between monadic programming and conventional imperative programming can be seen here in `putStr`. In an imperative language, mapping an imperative version of `putChar` over the string would be sufficient to print it. In Haskell, however, the `map` function does not perform any action. Instead it creates a list of actions, one for each character in the string. To trigger these actions, the `>>` function must be used. The `return ()` used here is quite necessary – `foldr` needs a 'unit' or 'zero' to account for the end of the string of characters.

To see how to perform repetitive I/O operations, consider the `getLine` function:

```
getLine                 :: IO String
getLine                 = getChar >>= \c ->
                            if c == '\n' then return "" else
                               getLine >>= \l -> return (c:l)
```

The I/O monad does not preclude ordinary recursive function calling, although `>>=` is needed to make the recursive call from within the monad. Also, the `return` function is used here to define the value returned by `getLine`.

The reader may wonder at this point how overall program execution is defined in Haskell. This issue is resolved using a simple convention: the user must define `main` in module `Main` as the overall action associated with the program. The type of `main` must be `IO ()` – an action returning no value. For example, the following program:

```
module Main where
main =                  putStr "Hello World\n"
```

does as expected when compiled and executed. (The `putStr` function is defined in the standard prelude and need not be defined by the user.)

8.2 Error Handling

So far, we have avoided the issue of errors during I/O operations. What would happen, for example, if `getChar` encounters an end of file? To deal with errors within the I/O monad, an exception handling mechanism is used, similar in functionality to the one in standard ML. No special syntax or semantics is required; there are just a few extra operations in the I/O monad which deal with exceptions. It is important to remember that Haskell supports exceptions only within the I/O monad. Conditions such as division by 0 or arithmetic overflow do not use exception handling. Instead, they either terminate execution or produce some standard value to denote an error.[14]

Errors are encoded using a special data type, `IOError`. The definition of `IOError` is given by:

```
data IOError             = EOF
                         | ReadError String
                         | WriteError String
                         | SeekError String
                         | OtherError String
```

This type represents all possible errors which may occur within the I/O monad. An *exception handler* has type `IOError -> IO a`. The `try` function associates an exception handler with an action or set of actions:

```
try                      :: IO a -> (IOError -> IO a) -> IO a
```

The arguments to `try` are an action and a handler. If the action succeeds, its result is returned without invoking the handler. If an error occurs, it is passed to the handler as a value of type `IOError`, and the action associated with the handler is then invoked. For example, this version of `getChar` returns a newline when an error is encountered:

```
getChar'                 :: IO Char
getChar'                 = try getChar (\e ->
```

any other list. An implementation can be expected to implement this demand-driven behavior by reading one character at a time from the file as they are required by the computation.

In this example, a Haskell program copies one file to another:

```
module Main where

main = getAndOpenFile "Copy from: " ReadMode >>= \fromHandle ->
       getAndOpenFile "Copy to: " WriteMode >>= \toHandle ->
       getContents fromHandle >>= \contents ->
       hPutStr toHandle contents >> close toHandle >> putStr "Done."

getAndOpenFile          :: String -> IOMode -> IO Handle

getAndOpenFile prompt mode =
       putStr prompt >>
       getLine >>= \name ->
       try (openFile mode name)
           (putStr "Cannot open "++ name ++ "\n" >>
            getAndOpenFile prompt mode)
```

By using the lazy `getContents` function, the entire contents of the file need not be read into memory all at once. If `hPutStr` chooses to buffer the output by writing the string in fixed sized blocks of characters, only one block of the input file needs to be in memory at once.

8.4 Haskell and Imperative Programming

As a final note, monadic programming raises an important issue: this style looks suspiciously like ordinary imperative programming. For example, the `getLine` function:

```
getLine         = getChar >>= \c ->
                  if c == '\n' then return ""
                  else getLine >>= \l -> return (c:l)
```

bears a striking similarity to imperative code:

```
function getLine() {
  c := getChar();
  if c == ‘\n‘ then return ""
             else {l := getLine();
                   return c:l}}
```

So, in the end, has Haskell simply re-invented the imperative wheel?

In some sense, yes. The I/O monad constitutes a small imperative sub-language inside Haskell, and thus the I/O component of a program may appear similar to ordinary imperative code. But there is one important difference: There is no special syntax or semantics that the user needs to deal with. In particular, equational reasoning in Haskell is not compromised. The imperative feel of the monadic code in a program does not detract in any way from the functional aspect of Haskell. An experienced functional programmer should be able to minimize the imperative component of the program, only using the monad for a minimal amount of top level sequencing. A major advantage of the monad is that it cleanly separates the functional and imperative program components. In contrast, imperative languages with functional subsets do not generally have any well defined barrier between the purely functional and imperative worlds.

9 Arrays

Ideally, arrays in a functional language would be regarded simply as functions from indices to values, but pragmatically, in order to assure efficient access to array elements, we need to be sure we can take advantage of the special properties of the domains of these functions, which are isomorphic to finite contiguous subsets of the integers. Haskell, therefore, does not treat arrays as general functions with an application operation, but as abstract data types with a subscript operation.

Two main approaches to functional arrays may be discerned: *incremental* and *monolithic* definition. In the incremental case, we have a function that produces an empty array of a given size and another that takes an array, an index, and a value, producing a new array that differs from the old one only at the given index. Obviously, a naive implementation of such an array semantics would be intolerably inefficient, requiring a new copy of an array for each incremental redefinition; thus, serious attempts at using this approach employ sophisticated static analysis and clever run-time devices to avoid excessive copying. The monolithic approach, on the other hand, constructs an array all at once, without reference to intermediate array values. Although Haskell has an incremental array update operator, the main thrust of the array facility is monolithic.

9.1 Index types

The Standard Prelude defines a type class of array indices:

```
class  (Ord a) => Ix a  where
    range       :: (a,a) -> [a]
    index       :: (a,a) a -> Int
    inRange     :: (a,a) -> a -> Bool
```

Instance declarations are provided for `Int`, `Integer`, `Char`, `Bool`, and tuples of `Ix` types; in addition, instances may be automatically derived for enumerated and tuple types. We regard the primitive types as vector indices and tuples as indices of multidimensional rectangular arrays. Note that the first argument of each of the operations of class `Ix` is a pair of indices; these are typically the *bounds* (first and last indices) of an array. For example, the bounds of a 10-element, zero-origin vector with `Int` indices would be `(0,9)`, while a 100 by 100 1-origin matrix might have the bounds `((1,1),(100,100))`. (In many other languages, such bounds would be written in a form like `1:100, 1:100`, but the present form fits the type system better, since each bound is of the same type as a general index.)

The `range` operation takes a bounds pair and produces the list of indices lying between those bounds, in index order. For example,

`range (0,4)` ⇒ `[0,1,2,3,4]`

`range ((0,0),(1,2))` ⇒ `[(0,0), (0,1), (0,2), (1,0), (1,1), (1,2)]`

The `inRange` predicate determines whether an index lies between a given pair of bounds. (For a tuple type, this test is performed componentwise.) Finally, the `index` operation is what is needed to address a particular element of an array: Given a bounds pair and an in-range index, the operation yields the zero-origin ordinal of the index within the range; for example:

```
index (1,9) 2   ⇒   1
```

```
index ((0,0),(1,2)) (1,1)   ⇒   5
```

9.2 Array Creation

Haskell's monolithic array creation function forms an array from a pair of bounds and a list of index-value pairs (an *association list*):

```
array                   :: (Ix a) => (a,a) -> [(a,b)] -> Array a b
```

Here, for example, is a definition of an array of the squares of numbers from 1 to 100:

```
squares                 =  array (1,100) [(i, i*i) | i <- [1..100]]
```

This array expression is typical in using a list comprehension for the association list; in fact, this usage results in array expressions much like the *array comprehensions* of the language Id[4]. Array subscripting is performed with the infix operator `!`, and the bounds of an array can be extracted with the function `bounds`:

```
squares!7   ⇒   49
```

```
bounds squares   ⇒   (1,100)
```

We might generalize this example by parameterizing the bounds and the function to be applied to each index:

```
mkArray                 :: (Ix a) => (a -> b) -> (a,a) -> Array a b
mkArray f bnds          =  array bnds [(i, f i) | i <- range bnds]
```

Thus, we could define `squares` as `mkArray (\i -> i * i) (1,100)`.

Many arrays are defined recursively; that is, with the values of some elements depending on the values of others. Here, for example, we have a function returning an array of Fibonacci numbers:

```
fibs    :: Int -> Array Int Int
fibs n  =  a  where a = array (0,n) ([(0, 1), (1, 1)] ++
                                     [(i, a!(i-2) + a!(i-1)) | i <- [2..n]])
```

Another example of such a recurrence is the n by n *wavefront* matrix, in which elements of the first row and first column all have the value *1* and other elements are sums of their neighbors to the west, northwest, and north:

```
wavefront       :: Int -> Array (Int,Int) Int
wavefront n     =  a  where
                   a = array ((1,1),(n,n))
                        ([((1,j), 1) | j <- [1..n]] ++
                         [((i,1), 1) | i <- [2..n]] ++
                         [((i,j), a!(i,j-1) + a!(i-1,j-1) + a!(i-1,j))
                                     | i <- [2..n], j <- [2..n]])
```

The wavefront matrix is so called because in a parallel implementation, the recurrence dictates that the computation can begin with the first row and column in parallel and proceed as a wedge-shaped wave, travelling from northwest to southeast. It is important to note, however, that no order of computation is specified by the association list.

In each of our examples so far, we have given a unique association for each index of the array and only for the indices within the bounds of the array, and indeed, we must do this in general for an array be fully defined. An association with an out-of-bounds index results in an error; if an index is missing or appears more than once, however, there is no immediate error, but the value of the array at that index is then undefined, so that subscripting the array with such an index yields an error.

9.3 Accumulation

We can relax the restriction that an index appear at most once in the association list by specifying how to combine multiple values associated with a single index; the result is called an *ffffffumulfftffff ffrrffy*:

```
accumArray :: (Ix a) -> (b -> c -> b) -> b -> (a,a) -> [Assoc a c] -> Array a b
```

The first argument of `accumArray` is the *ffffffumulfftffnff ffunfftffon*, the second is an initial value (the same for each element of the array), and the remaining arguments are bounds and an association list, as with the `array` function. Typically, the accumulating function is `(+)`, and the initial value, zero; for example, this function takes a pair of bounds and a list of values (of an index type) and yields a histogram; that is, a table of the number of occurrences of each value within the bounds:

```
hist            :: (Ix a, Integral b) => (a,a) -> [a] -> Array a b
hist bnds is    =  accumArray (+) 0 bnds [(i, 1) | i <- is, inRange bnds i]
```

Suppose we have a collection of measurements on the interval *[ff, ff)*, and we want to divide the interval into decades and count the number of measurements within each:

```
decades         :: (RealFrac a) => a -> a -> [a] -> Array Int Int
decades a b     =  hist (0,9) . map decade
                   where decade x = floor ((x - a) * s)
                         s        = 10 / (b - a)
```

9.4 Incremental updates

In addition to the monolithic array creation functions, Haskell also has an incremental array update function, written as the infix operator `//`; the simplest case, an array `a` with element `i` updated to `v`, is written `a // [(i, v)]`. The reason for the square brackets is that the left argument of `(//)` is an association list, usually containing a proper subset of the indices of the array:

```
(//)            :: (Ix a) => Array a b -> [(a,b)] -> Array a b
```

As with the `array` function, the indices in the association list must be unique for the values to be defined. For example, here is a function to interchange two rows of a matrix:

```
swapRows :: (Ix a, Ix b, Enum b) => a -> a -> Array (a,b) c -> Array (a,b) c
swapRows i i' a =  a // ([((i ,j), a!(i',j)) | j <- [jLo..jHi]] ++
                         [((i',j), a!(i ,j)) | j <- [jLo..jHi]])
                   where ((iLo,jLo),(iHi,jHi)) = bounds a
```

The concatenation here of two separate list comprehensions over the same list of `j` indices is, however, a slight inefficiency; it's like writing two loops where one will do in an imperative language. Never fear, we can perform the equivalent of a loop fusion optimization in Haskell:

```
swapRows i i' a =  a // [assoc | j <- [jLo..jHi],
                                 assoc <- [((i ,j), a!(i',j)),
                                           ((i',j), a!(i, j))] ]
                   where ((iLo,jLo),(iHi,jHi)) = bounds a
```

9.5 An example: Matrix Multiplication

We complete our introduction to Haskell arrays with the familiar example of matrix multiplication, taking advantage of overloading to define a fairly general function. Since only multiplication and addition on the element type of the matrices is involved, we get a function that multiplies matrices of any numeric type unless we try hard not to. Additionally, if we are careful to apply only `(!)` and the operations of `Ix` to indices, we get genericity over index types, and in fact, the four row and column index types need not all be the same. For simplicity, however, we require that the left column indices and right row indices be of the same type, and moreover, that the bounds be equal:

```
matMult         :: (Ix a, Ix b, Ix c, Num d) =>
                   Array (a,b) d -> Array (b,c) d -> Array (a,c) d
matMult x y     =  array resultBounds
                         [((i,j), sum [x!(i,k) * y!(k,j) | k <- range (lj,uj)])
                                      | i <- range (li,ui),
                                        j <- range (lj',uj') ]
        where ((li,lj),(ui,uj))         =  bounds x
              ((li',lj'),(ui',uj'))     =  bounds y
              resultBounds
                | (lj,uj)==(li',ui')    =  ((li,lj'),(ui,uj'))
                | otherwise             = error "matMult: incompatible bounds"
```

As an aside, we can also define `matMult` using `accumArray`, resulting in a presentation that more closely resembles the usual formulation in an imperative language:

```
matMult x y     =  accumArray (+) 0 resultBounds
                              [((i,j), x!(i,k) * y!(k,j))
                                      | i <- range (li,ui),
                                        j <- range (lj',uj')
                                        k <- range (lj,uj)  ]
        where ((li,lj),(ui,uj))         =  bounds x
              ((li',lj'),(ui',uj'))     =  bounds y
              resultBounds
                | (lj,uj)==(li',ui')    =  ((li,lj'),(ui,uj'))
                | otherwise             = error "matMult: incompatible bounds"
```

We can generalize further by making the function higher-order, simply replacing `sum` and `(*)` by functional parameters:

```
genMatMult      :: (Ix a, Ix b, Ix c) =>
                   ([f] -> g) -> (d -> e -> f) ->
                   Array (a,b) d -> Array (b,c) e -> Array (a,c) g
genMatMult f g x y  =  array resultBounds
                        [((i,j), f [x!(i,k) `g` y!(k,j) | k <- range (lj,uj)])
                                 | i <- range (li,ui),
                                   j <- range (lj',uj') ]
        where ((li,lj),(ui,uj))         =  bounds x
              ((li',lj'),(ui',uj'))     =  bounds y
              resultBounds
                | (lj,uj)==(li',ui')    =  ((li,lj'),(ui,uj'))
                | otherwise             = error "matMult: incompatible bounds"
```

APL fans will recognize the usefulness of functions like the following:

```
genMatMult maximum (-)
genMatMult and (==)
```

With the first of these, the arguments are numeric matrices, and the (i,j)-th element of the result is the maximum difference between corresponding elements of the i-th row and j-th column of the inputs. In the second case, the arguments are matrices of any equality type, and the result is a Boolean matrix in which element (i,j) is `True` if and only if the i-th row of the first argument and j-th column of the second are equal as vectors.

Notice that the element types of `genMatMult` need not be the same, but merely appropriate for the function parameter `g`. We could generalize still further by dropping the requirement that the first column index and second row index types be the same; clearly, two matrices could be considered conformable as long as the lengths of the columns of the first and the rows of the second are equal. The reader may wish to derive this still more general version. (**Hint:** Use the `index` operation to determine the lengths.)

10 Acknowledgements

Thanks to Patricia Fasel and Mark Mundt at Los Alamos, and Nick Carriero, Charles Consel, Amir Kishon, Sandra Loosemore, Martin Odersky, John Peterson, and David Rochberg at Yale University for their quick readings of earlier drafts of this manuscript. Special thanks to John Peterson for getting the bugs out of our Haskell code.

References

[1] R. Bird and P. Wadler. *Introduction to Functional Programming*. Prentice Hall, New York, 1988.

[2] P. Hudak. Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, 21(3):359–411, 1989.

[3] P. Hudak, S. Peyton Jones, and P. Wadler (editors). Report on the Programming Language Haskell, A Non-strict Purely Functional Language (Version 1.2). *ACM SIGPLAN Notices*, 27(5), May 1992.

[4] R.S. Nikhil. Id (version 90.0) reference manual. Technical report, Massachusetts Institute of Technology, Laboratory for Computer Science, September 1990.

[5] J. Rees and W. Clinger (eds.). The revised^3 report on the algorithmic language Scheme. *SIGPLAN Notices*, 21(12):37–79, December 1986.

[6] G.L. Steele Jr. *Common Lisp: The Language*. Digital Press, Burlington, Mass., 1984.

[7] P. Wadler. How to replace failure by a list of successes. In *Proceedings of Conference on Functional Programming Languages and Computer Architecture, LNCS Vol. 201*, pages 113–128. Springer Verlag, 1985.