Exploring Languages with Interpreters and Functional Programming Chapter 5

H. Conrad Cunningham

8 September 2018

Contents

Professor of Computer and Information Science
University of Mississippi
211 Weir Hall

P.O. Box 1848
University, MS 38677
(662) 915-5358

Browser Advisory: The HTML version of this textbook requires a browser that supports the display of MathML. A good choice as of September 2018 is a recent version of Firefox from Mozilla.

5 Types

5.1 Chapter Introduction

The goals of this chapter are to:

- examine the general concepts of type systems
- explore Haskell's builtin types

5.2 Type System Concepts

The term *type* tends to be used in many different ways in programming languages. What is a type?

The chapter on object-based paradigms discusses the concept of type in the context of object-oriented languages. This chapter first examines the concept more generally and then examines Haskell's builtin types.

5.2.1 Types and subtypes

Conceptually, a *type* is a set of values (i.e. possible states or objects) and a set of operations defined on the values in that set.

Similarly, a type `S` is (a behavioral) *subtype* of type `T` if the set of values of type `S` is a "subset" of the values in set `T` and set of operations of type `S` is a "superset" of the operations of type `T`. That is, we can safely *substitute* elements of subtype `S` for elements of type `T` because `S`'s operations behave the "same" as `T`'s operations.

This is known as the *Liskov Substitution Principle* [Liskov 1987] [Wikipedia 2018a].

Consider a type representing all furniture and a type representing all chairs. In general, we consider the set of chairs to be a subset of the set of furniture. A chair should have all the general characteristics of furniture, but it may have additional characteristics specific to chairs.

If we can perform an operation on furniture in general, we should be able to perform the same operation on a chair under the same circumstances and get the same result. Of course, there may be additional operations we can perform on chairs that are not applicable to furniture in general.

Thus the type of all chairs is a subtype of the type of all furniture according to the Liskov Substitution Principle.

5.2.2 Constants, variables, and expressions

Now consider the types of the basic program elements.

A *constant* has whatever types it is defined to have in the context in which it is used. For example, the constant symbol `1` might represent an integer, a real number, a complex number, a single bit, etc., depending upon the context.

A *variable* has whatever types its value has in a particular context and at a particular time during execution.

An *expression* has whatever types its evaluation yields based on the types of the variables, constants, and operations from which it is constructed.

5.2.3 Static and dynamic

In a *statically typed language*, the types of a variable or expression can be determined from the program source code and checked at "compile time" (i.e. during the syntactic and semantic processing in the front-end of a language processor). Such languages may require at least some of the types of variables or expressions to be *declared* explicitly, while others may be *inferred* implicitly from the context.

Java, Scala, and Haskell are examples of statically typed languages.

In a *dynamically typed language*, the specific types of a variable or expression cannot be determined at "compile time" but can be checked at runtime.

Lisp, Python, JavaScript, and Lua are examples of dynamically typed languages.

Of course, most languages use a mixture of static and dynamic typing. For example, Java objects defined within an inheritance hierarchy must be bound dynamically to the appropriate operations at runtime. Also Java objects declared of type `Object` (the root class of all user-defined classes) often require explicit runtime checks or coercions.

5.2.4 Nominal and structural

In a language with *nominal typing*, the type of value is based on the type *name* assigned when the value is created. Two values have the same type if they have the same type name. A type `S` is a subtype of type `T` only if `S` is explicitly declared to be a subtype of `T`.

For example, Java is primarily a nominally typed language. It assigns types to an object based on the name of the class from which the object is instantiated and the superclasses extended and interfaces implemented by that class.

However, Java does not guarantee that subtypes satisfy the Liskov Substitution Principle. For example, a subclass might not implement an operation in a manner that is compatible with the superclass. (The behavior of subclass objects are this different from the behavior of superclass objects.) Ensuring that Java subclasses preserve the Substitution Principle is considered good programming practice in most circumstances.

In a language with *structural typing*, the type of a value is based on the *structure* of the value. Two values have the same type if they have the "same" structure; that is, they have the same *public* data attributes and operations and these are themselves of compatible types.

In structurally typed languages, a type `S` is a subtype of type `T` only if `S` has all the public data values and operations of type `T` and the data values and operations are themselves of compatible types. Subtype `S` may have additional data values and operations not in `T`.

Haskell is primarily a structurally typed language.

5.2.5 Polymorphic operations

Polymorphism refers to the property of having "many shapes". In programming languages, we are primarily interested in how *polymorphic* function names (or operator symbols) are associated with implementations of the functions (or operations).

In general, two primary kinds of polymorphism exist in programming languages:

1. *Ad hoc polymorphism*, in which the same function name (or operator symbol) can denote different implementations depending upon how it is used in an expression. That is, the implementation invoked depends upon the types of function's arguments and return value.

   There are two subkinds of ad hoc polymorphism.

   a. *Overloading* refers to ad hoc polymorphism in which the language's compiler or interpreter determines the appropriate implementation to invoke using information from the context. In statically typed languages, overloaded names and symbols can usually be bound to the intended implementation at *compile time* based on the declared types of the entities. They exhibit *early binding*.

      Consider the language Java. It overloads a few operator symbols, such as using the `+` symbol for both addition of numbers and concatenation of strings. Java also overloads calls of functions defined with the same name but different signatures (patterns of parameter types and return value). Java does not support user-defined operator overloading; C++ does.

      Haskell's *type class* mechanism, which we examine in a later chapter, implements overloading polymorphism in Haskell. There are similar mechanisms in other languages such as Scala and Rust.

b. *Subtyping* (also known as *subtype polymorphism* or *inclusion polymorphism*) refers to ad hoc polymorphism in which the appropriate implementation is determined by searching a hierarchy of types. The function may be defined in a supertype and redefined (overridden) in subtypes. Beginning with the actual types of the data involved, the program searches up the type hierarchy to find the appropriate implementation to invoke. This usually occurs at runtime, so this exhibits *late binding*.

   The object-oriented programming community often refers to inheritance-based subtype polymorphism as simply *polymorphism*. This the polymorphism associated with the class structure in Java.

   Haskell does not support subtyping. Its type classes do support *class extension*, which enables one class to inherit the properties of another. However, Haskell's classes are not types.

2. *Parametric polymorphism*, in which the same implementation can be used for many different types. In most cases, the function (or class) implementation is stated in terms of one or more type parameters. In statically typed languages, this binding can usually be done at compile time (i.e. exhibiting early binding).

   The object-oriented programming (e.g. Java) community often calls this type of polymorphism *generics* or *generic programming*.

   The functional programming (e.g. Haskell) community often calls this simply *polymorphism*.

5.2.6 Polymorphic variables

A *polymorphic variable* is a variable that can "hold" values of different types during program execution.

For example, a variable in a dynamically typed language (e.g. Python) is polymorphic. It can potentially "hold" any value. The variable takes on the type of whatever value it "holds" at a particular point during execution.

Also, a variable in a nominally and statically typed, object-oriented language (e.g. Java) is polymorphic. It can "hold" a value its declared type or of any of the subtypes of that type. The variable is declared with a static type; its value has a dynamic type.

A variable that is a parameter of a (parametrically) polymorphic function is polymorphic. It may be bound to different types on different calls of the function.

5.3 Surveying the Basic Types

The type system is an important part of Haskell; the compiler or interpreter uses the type information to detect errors in expressions and function definitions. To each expression Haskell assigns a *type* that describes the kind of value represented by the expression.

Haskell has both built-in types (defined in the language or its standard libraries) and facilities for defining new types. In the following we discuss the primary built-in types. As we have seen, a Haskell type name begins with a capital letter.

In this textbook, we sometimes refer to the types `Int`, `Float`, `Double`, `Bool`, and `Char` as being *primitive* because they likely have direct support in the host processor's hardware.

5.3.1 Integers: `Int` and `Integer`

The `Int` data type is usually an integer data type supported directly by the host processor (e.g. 32- or 64-bits on most current processors), but it is guaranteed to have the range of at least a 30-bit, two's complement integer.

The type `Integer` is an unbounded precision integer type. Unlike `Int`, host processors usually do not support this type directly. The Haskell library or runtime system typically supports this type in software.

Haskell integers support the usual literal formats (i.e. constants) and typical operations:

- Infix binary operators such as `+` (addition), `-` (subtraction), `*` (multiplication), and `^` (exponentiation)
- Infix binary comparison operators such as `==` (equality of values), `/=` (inequality of values), `<`, `<=`, `>`, and `>=`
- Unary operator `-` (negation)

For integer division, Haskell provides two-argument functions `div` and `rem` such that `div m n` returns the integral quotient from dividing `m` by `n` and `rem m n` returns the remainder.

Haskell also provides the useful two-argument functions `min` and `max`, which return the minimum and maximum of the two arguments, respectively.

Two-arguments functions such as `div`, `rem`, `min`, and `max` can be applied in infix form by including the function name between backticks as show below:

```
5 `div` 3   -- yields 1
5 `rem` 3   -- yields 2
5 `min` 3   -- yields 3
5 `max` 3   -- yields 5
```

5.3.2 Floating point numbers: `Float` and `Double`

The `Float` and `Double` data types are usually the single and double precision floating point numbers supported directly by the host processor.

Haskell floating point literals must include a decimal point; they may be signed or in scientific notation: `3.14159`, `2.0`, `-2.0`, `1.0e4`, `5.0e-2`, `-5.0e-2`.

Haskell supports the usual operations on floating point numbers. Division is denoted by `/` as usual.

5.3.3 Booleans: `Bool`

The `Bool` (i.e. Boolean) data type is usually supported directly by the host processor as one or more contiguous bits.

The `Bool` literals are `True` and `False`. Note that these begin with capital letters.

Haskell supports Boolean operations such as `&&` (and), `||` (or), and `not` (logical negation).

Functions can match against patterns using the Boolean constants. For example, we could define a function `myAnd` as follows:

```haskell
myAnd :: Bool -> Bool -> Bool
myAnd True  b = b
myAnd False _ = False
```

Above the pattern `_` is a wildcard that matches any value but does not bind a value that can be used on the right-hand-side of the definition.

The expressions in Haskell `if` conditions and guards on function definitions must be `Bool`-valued expressions. They can include calls to functions that return `Bool` values.

5.3.4 Characters: `Char`

The `Char` data type is usually supported directly by the host processor by one or more contiguous bytes.

Haskell uses Unicode for its character data type. Haskell supports character literals enclosed in single quotes—including both the graphic characters (e.g. `'a'`, `'0'`, and `'Z'`) and special codes entered following the escape character backslash \ (e.g. `'\n'` for newline, `'\t'` for horizontal tab, and `'\\'` for backslash itself).

In addition, a backslash character \ followed by a number generates the corresponding Unicode character code. If the first character following the backslash is `o`, then the number is in octal representation; if followed by `x`, then in hexadecimal notation; and otherwise in decimal notation.

For example, the exclamation point character can be represented in any of the following ways: `'!'`, `'\33'`, `'\o41'`, `'\x21'`

5.3.5 Functions: `t1 -> t2`

If `t1` and `t2` are types then `t1 -> t2` is the type of a function that takes an argument of type `t1` and returns a result of type `t2`.

Function and variable names begin with lowercase letters optionally followed by a sequences of characters each of which is a letter, a digit, an apostrophe (`'`) (sometimes pronounced "prime"), or an underscore (`_`).

Haskell functions are *first-class* objects. They can be arguments or results of other functions or be components of data structures. Multi-argument functions are *curried*—that is, treated as if they take their arguments one at a time.

For example, consider the integer addition operation `(+)`. (Surrounding the binary operator symbol with parentheses refers to the corresponding function.) In mathematics, we normally consider addition as an operation that takes a *pair* of integers and yields an integer result, which would have the type signature

```
(+) :: (Int,Int) -> Int
```

In Haskell, we give the addition operation the type

```
(+) :: Int -> (Int -> Int)
```

or just

```
(+) :: Int -> Int -> Int
```

since Haskell binds `->` from the right.

Thus `(+)` is a one argument function that takes some `Int` argument and returns a function of type `Int -> Int`. Hence, the expression `((+) 5)` denotes a function that takes one argument and returns that argument plus `5`.

We sometimes speak of this `(+)` operation as being *partially applied* (i.e. to one argument instead of two).

This process of replacing a structured argument by a sequence of simpler ones is called *currying*, named after American logician Haskell B. Curry who first described it.

The Haskell library, called the *standard prelude* (or just Prelude), contains a wide range of predefined functions including the usual arithmetic, relational, and Boolean operations. Some of these operations are predefined as *infix* operations.

5.3.6 Tuples: `(t1,t2,...,tn)`

If `t1`, `t2`, $\cdots$, `tn` are types, where `n` is finite and `n >= 2`, then is a type consisting of *n-tuples* where the various components have the type given for that position.

Each element in a tuple may have different types. The number of elements in a tuple is fixed.

Examples of tuple values with their types include the following:

```
('a',1) :: (Char,Int)
(0.0,0.0,0.0) :: (Double,Double,Double)
(('a',False),(3,4)) :: ((Char, Bool), (Int, Int))
```

We can also define a *type synonym* using the `type` declaration and the use the synonym in further declarations as follows:

```
type Complex = (Float,Float)
makeComplex :: Float -> Float -> Complex
makeComplex r i = (r,i)`
```

A type synonym does not define a new type, but it introduces an alias for an existing type. We can use `Complex` in declarations, but it has the same effect as using `(Float,Float)` expect that `Complex` provides better documentation of the intent.

5.3.7 Lists: `[t]`

The primary built-in data structure in Haskell is the *list*, a sequence of values. All the elements in a list must have the same type. Thus we declare lists with notation such as `[t]` to denote a list of zero or more elements of type `t`.

A list literal is a comma-separated sequence of values enclosed between `[` and `]`. For example, `[]` is an empty list and `[1,2,3]` is a list of the first three positive integers in increasing order.

We will look at programming with lists in a later chapter.

5.3.8 Strings: `String`

In Haskell, a *string* is just a list of characters. Thus Haskell defines the data type `String` as a *type synonym* :

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

We examine lists and strings in a later chapter, but, because we use strings in a few examples in this subsection, let's consider them briefly.

A `String` literal is a sequence of zero or more characters enclosed in double quotes, for example, `"Haskell programming"`.

Strings can contain any graphic character or any special character given as escape code sequence (using backslash). The special escape code `\&` is used to separate any character sequences that are otherwise ambiguous.

For example, the string literal `"Hotty\nToddy!\n"` is a string that has two newline characters embedded.

Also the string literal `"\12\&3"` represents the two-element list `['\12','3']`.

Because strings are represented as lists, all of the Prelude functions for manipulating lists also apply to strings. We look at manipulating lists and strings in later chapters of this textbook.

5.4 What Next?

In this chapter, we examined general type systems concepts and explored Haskell's builtin types.

In the next two chapters, we examine methods for developing Haskell programs using abstraction. We explore use of top-down stepwise refinement, modular design, and other methods in the context of Haskell.

5.5 Exercises

For each of the following exercises, develop and test a Haskell function or set of functions.

1. Develop a Haskell function `sumSqBig` that takes three `Double` arguments and returns the sum of the squares of the two larger numbers.

   For example, `sumSqBig 2.0 1.0 3.0` yields `13.0`.

2. Develop a Haskell function `prodSqSmall` that takes three `Double` arguments and returns the product of the squares of the two smaller numbers.

   For example, `prodSqSmall 2.0 4.0 3.0` yields `36.0`.

3. Develop a Haskell function `xor` that takes two Booleans and returns the "exclusive-or" of the two values. An exclusive-or operation returns `True` when exactly one of its arguments is `True` and returns `False` otherwise.

4. Develop a Haskell Boolean function `implies` that takes two Booleans `p` and `q` and returns the Boolean result `p` $\Rightarrow$ `q` (i.e. logical implication). That is, if `p` is `True` and `q` is `False`, then the result is `False`; otherwise, the result is `True`.

   Note: This function is sometimes called `nand`.

5. Develop a Haskell Boolean function `div23n5` that takes an `Int` and returns `True` if and only if the integer is divisible by 2 or divisible by 3, but is not divisible by 5.

   For example, `div23n5 4`, `div23n5 6`, and `div23n5 9` yield `True` and `div23n5 5`, `div23n5 7`, `div23n5 10`, `div23n5 15`, `div23n5 30` yield `False`.

6. Develop a Haskell function `notDiv` such that `notDiv n d` returns `True` if and only if integer `n` is not divisible by `d`.

   For example, `notDiv 10 5` yields `False` and `notDiv 11 5` yields `True`.

7. Develop a Haskell function `ccArea` that takes the *diameters* of two concentric circles (i.e. with the same center point) as `Double` values and returns the area of the space between the circles. That is, compute the area of the larger circle minus the area of the smaller circle. (Hint: Haskell has a builtin constant `pi`.)

   For example, `ccArea 2.0 4.0` yields approximately `9.42477796`.

8. Develop a Haskell function `mult` that takes two *natural numbers* (i.e. nonnegative integers in `Int`) and returns their product. The function must not use the multiplication (`*`) or division (`div`) operators. Hint: Multiplication can be done by repeated addition.

9. Develop a Haskell function `addTax` that takes two `Double` values such that `addTax c p` returns `c` with a sales tax of `p` *percent* added. For example, `addTax 2.0 9.0` returns `2.18`.

   Also develop a function `subTax` that is the inverse of `addTax`. That is, `subTax (addTax c p) p` yields `c`. For example, `subTax 2.18 9.0 = 2.0`.

10. The time of day can be represented by a tuple `(hours,minutes,m)` where `hours` and `minutes` are `Int` values with `1 <= hours <= 12` and `0 <= minutes <= 59`, and wehre `m` is either the string value `"AM"` or `"PM"`.

    Develop a Boolean Haskell function `comesBefore` that takes two time-of-day tuples and determines whether the first is an earlier time than the second.

11. A day on the calendar (usual Gregorian calendar used in the USA) can be represented as a tuple with three `Int` values `(month,day,year)` where the `year` is a positive integer, `1 <= month <= 12`, and `1 <= day <= days_in_month`. Here `days_in_month` is the number of days in the the given `month` (i.e. 28, 29, 30, or 31) for the given `year`.

    Develop a Boolean Haskell function `validDay d` that takes a date tuple `d` and returns `True` if and only if `d` represents a valid date.

    For example, `validDay (8,20,2018)` and `validDay(2,29,2016}` yield `True` and `validDay (2,29,2017)` and `validDay(0,0,0)` yield `False`.

Note: The Gregorian calendar [Wikipedia 2018c] was introduced by Pope Gregory of the Roman Catholic Church in October 1582. It replaced the Julian calendar system, which had been instituted in the Roman Empire by Julius Caesar in 46 BC. The goal of the change was to align the calendar year with the astronomical year.

Some countries adopted the Gregorian calendar at that time. Other countries adopted it later. Some countries may never have adopted it officially.

However, the Gregorian calendar system became the common calendar used worldwide for most civil matters. The *proleptic Gregorian calendar* [Wikipedia 2018c] extends the calendar backward in time from 1582. The year 1 BC becomes year 0, 2 BC becomes year -1, etc. The proleptic Gregorian calendar underlies the ISO 8601 standard used for dates and times in software systems [Wikipedia 2018c].

TODO: In the future, change the name of `validDay` to `validDate`, which is more accurate.

12. Develop a Haskell function `roman` that takes an `Int`) in the range from 0 to 3999 (inclusive) and returns the corresponding Roman numeral as a string (using capital letters). The function should halt with an appropriate `error` messages if the argument is below or above the range. Roman numbers use the following symbols and are combined by addition or subtraction of symbols.

I	1
V	5
X	10
L	50
C	100
D	500
M	1000

For the purposes of this exercise, we represent the Roman numeral for 0 as the empty string. The Roman numbers for integers 1-20 are `I`, `II`, `III`, `IV`, `V`, `VI`, `VII`, `VIII`, `IX`, `X`, `XI`, `XII`, `XIII`, `XIV`, `XV`, `XVI`, `XVII`, `XVII`, `XIX`, and `XX`. Integers 40, 90, 400, and 900 are `XL`, `XC`, `CD`, and `CM`.

13. Develop a Haskell function

```
minf :: (Int -> Int) -> Int
```

that takes a function `g` and returns the smallest integer `m` such that `0 <= m <= 10000000` and `g m == 0`. It should throw and `error` if there is no such integer.

5.6 Acknowledgements

In Summer 2016, I adapted and revised the discussion Surveying the Basic Types from chapter 5 of my *Notes on Functional Programming with Haskell* [Cunningham 2014]. In 2017, I incorporated the discussion into Section 2.4 in Chapter 2 Basic Haskell Functional Programming of my 2017 Haskell-based programming languages textbook.

In Spring and Summer 2018, I divided the previous Basic Haskell Functional Programming chapter into four chapters in the 2018 version of the textbook, now titled *Exploring Languages with Interpreters and Functional Programming*. Previous sections 2.1-2.3 became the basis for new Chapter 4, First Haskell Programs; previous Section 2.4 became Section 5.3 in the new Chapter 5, Types (this chapter); and previous sections 2.5-2.7 were reorganized into new Chapter 6, Procedural Abstraction, and Chapter 7, Data Abstraction.

In Spring 2018, I wrote the general Type System Concepts section as a part of a chapter that discusses the type system of Python 3 [Cunningham 2018]. In Summer 2018, I revised it to become Section 5.2 in this new Chapter 5. I also moved the "Kinds of Polymorphism" discussion from the 2017 List Programming chapter to the new subsection "Polymorphic Operations". This section had drawn on various Wikipedia articles [Wikipedia 2018b] and other sources.

I maintain this chapter as text in Pandoc's dialect of Markdown using embedded LaTeX markup for the mathematical formulas and then translate the document to HTML, PDF, and other forms as needed.

5.7 References

[Cunningham 2014]: H. Conrad Cunningham. *Notes on Functional Programming with Haskell*, 1993-2014.

[Cunningham 2018]: H. Conrad Cunningham. Basic Features Supporting Metaprogramming, Chapter 2, *Python 3 Reflexive Metaprogramming*, 2018.

[Liskov 1987]: Barbara Liskov. Keynote Address—Data Abstraction and Hierarchy, In the Addendum to the *Proceedings on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '87)*, Leigh Power and Zvi Weiss, Editors, ACM, 1987. [local]

[Wikipedia 2018a]: Wikipedia, Liskov Substitution Principle, accessed 30 August 2018.

[Wikipedia 2018b]: *Wikipedia* articles on "Polymorphism", "Ad Hoc Polymorphism", "Parametric Polymorphism", "Subtyping", and "Function Overloading", accessed 30 August 2018.

[Wikipedia 2018c]: *Wikipedia* articles on "Gregorian Calendar", "Proleptic Gregorian Calendar", and "ISO 8601", accessed 8 September 2018.

5.8 Terms and Concepts

Type, subtype, Liskov Substitution Principle, types of constants, variables, and expressions, static vs. dynamic types, nominal vs. structural types, polymorphic operations (ad hoc, overloading, subtyping, parametric/generic), early vs. late binding, compile time vs. runtime, polymorphic variables, basic Haskell types (`Int`, `Integer`, `Bool`, `Char`, functions, tuples, lists, `String`), `type` aliases, library (Prelude) functions, proleptic Gregorian calendar system, Roman numerals.