# Chapter 3

## Types and Clauses

Connor Baker

Compiled on December 21, 2019 at 9:04pm

## 3.1 Basic Concepts

> A type is a collection of related values.
>
> Excerpt From: Graham Hutton. “Programming in Haskell” (2nd ed.).

This means that the set-theoretic definition of the type Bool could be thought of as

```
Bool = {True, False}
```

and the type `Bool -> Bool` would be

```
Bool->Bool = {f: Bool -> Bool}
```

Haskell uses the notation `e :: T` to say that `e` is a value in the type `T` (that is, `e` has type `T`).

Every expression must have a type. Types are calculated prior to evaluating an expression via type inference. We get type errors when the inferred type is not the same as the explicit type.

## 3.2 Basic Types

| Type | Values |
|---|---|
| `Bool` | Logical values |
| `Char` | Single characters |
| `String` | A list of characters |
| `Int` | Fixed-precision integers |
| `Integer` | Arbitrary-precision integers |
| `Float` | Single-precision floating-point numbers |
| `Double` | Double-precision floating-point numbers |

## 3.3 List Types

> A list is a sequence of elements of the same type, with the elements being enclosed in square parentheses and separated by commas. We write `[T]` for the type of all lists whose elements have type `T`…
>
> Excerpt From: Graham Hutton. “Programming in Haskell” (2nd ed.).

The type of a list says nothing of its length (which may be infinite, because Haskell provides for lazy evaluation).

## 3.4 Tuple Types

> A tuple is a finite sequence of components of possibly different types, with the components being enclosed in round parentheses and separated by commas.
>
> Excerpt From: Graham Hutton. “Programming in Haskell” (2nd ed.).

The number of components of a tuple is called its arity. The arity of a tuple is zero or at least two (it must however, be finite). A tuple with arity one is not permitted because, as Graham notes, it would “conflict with the use of parentheses to make [arithmetic] evaluation order explicit.”

## 3.5 Function Types

> A function is a mapping from arguments of one type to results of another type.
>
> Excerpt From: Graham Hutton. “Programming in Haskell” (2nd ed.).

## 3.6 Curried Functions

Functions in Haskell are curried – that is, functions can return functions that take additional arguments.

> As well as being interesting in their own right, curried functions are also more flexible than functions on tuples, because useful functions can often be made by partially applying a curried function with less than its full complement of arguments.
>
> Excerpt From: Graham Hutton. “Programming in Haskell” (2nd ed.).

The arrow operator is right associative, which allows for excess parentheses when using curried functions. Graham notes that the following two types are equivalent:

```
Int -> Int -> Int -> Int
```

and

```
Int -> (Int -> (Int -> Int))
```

Graham also points out that, as a result, function application is left-associative:

```
must x y z
```

and

```
((must x)y)z
```

are equivalent.

> Unless tupling is explicitly required, all functions in Haskell with multiple arguments are normally defined as curried functions, and the two conventions above are used to reduce the number of parentheses that are required.

> Excerpt From: Graham Hutton. “Programming in Haskell” (2nd ed,).

## 3.7 Polymorphic Types

Polymorphic types are ones that can take any type as an argument. The `length` function is an example of a polymorphic type. It has the type

```
length :: [a] -> Int
```

## 3.8 Overloaded Types

A class constraint serves to restrict the application of a function to a certain instance of a type.

> Class constraints are written in the form C a, where C is the name of a class and a is a type variable. For example, the type of the addition operator + is as follows:
>
> ```
> (+):: Num a => a -> a -> a
> ```
>
> That is, for any type a that is an instance of the class Num of numeric types, the function `(+)` has type `a -> a -> a`.
>
> Excerpt From: Graham Hutton. “Programming in Haskell” (2nd ed.).

A type that includes class constraints is called overloaded.

> Numbers themselves are also overloaded. For example, `3 :: Num a => a` means that for any numeric type `a`, the value 3 has type `a`. In this manner, the value 3 could be an integer, a floating-point number, or more generally a value of any numeric type, depending on the context in which it is used.
>
> Excerpt From: Graham Hutton. “Programming in Haskell” (2nd ed.).

## 3.9 Basic Classes

> Recall that a type is a collection of related values. Building upon this notion, a class is a collection of types that support certain overloaded operations called methods.
>
> Excerpt From: Graham Hutton. “Programming in Haskell” (2nd ed.).

### Eq

> This class contains types whose values can be compared for equality and inequality using the following two methods:
>
> ```
> (==):: a -> a -> Bool
> ```

> `(/=):: a -> a -> Bool`
>
> All the basic types, `Bool`, `Char`, `String`, `Int`, `Integer`, `Float`, and `Double`, are instances of the `Eq` class, as are list and tuple types, provided that their element and component types are instances.
>
> Excerpt From: Graham Hutton. “Programming in Haskell” (2nd ed.).

Function types are not typically instances of Eq because it is not usually feasible to compare two functions for equality.

#### Ord

> This class contains types that are instances of the equality class `Eq`, but in addition whose values are totally (linearly) ordered, and as such can be compared and processed using the following six methods:
>
> `(<):: a -> a -> Bool`
>
> `(<=):: a -> a -> Bool`
>
> `(>):: a -> a -> Bool`
>
> `(>=):: a -> a -> Bool`
>
> `min :: a -> a -> a`
>
> `max :: a -> a -> a`
>
> All the basic types, `Bool`, `Char`, `String`, `Int`, `Integer`, `Float`, and `Double`, are instances of the `Ord` class, as are list and tuple types, provided that their element and component types are instances.
>
> Excerpt From: Graham Hutton. “Programming in Haskell” (2nd ed.).

`String`s, lists, and tuples are ordered lexicographically.

#### Show

These class is made of types that can be converted to a `String`. The basic list, tuple, and types we know are all instances of `Show`.

#### Read

This class complements `Show` in that it contains types which chan be converted from a `String` to some other type. The basic list, tuple, and types we know are all instances of `Show`.

`Read` can usually infer the type that the `String` should be converted to, but we can make the type inference explicit by using `:: T`. As an example:

```
read "[1,2,3]":: [Int]
```

tells GHC to turn the string into a list of `Int` instead of keeping it as a `String`.

**Num**

> This class contains types whose values are numeric, and as such can be processed using the following six methods:
>
> `(+):: a -> a -> a`
>
> `(-):: a -> a -> a`
>
> `(*):: a -> a -> a`
>
> `negate :: a -> a -> Bool`
>
> `abs :: a -> a`
>
> `signum :: a -> a`
>
> The basic types `Int`, `Integer`, `Float`, and `Double` are instances of the `Num` class.
>
> Excerpt From: Graham Hutton. “Programming in Haskell” (2nd ed.).

**Integral**

> This class contains types whose values are instances of the numeric class `Num`, but in addition whose values are integers, and as such support the methods of integer division and integer remainder:
>
> `div :: a -> a -> a`
>
> `mod :: a -> a -> a`
>
> The basic types `Int` and `Integer`, are instances of the `Integral` class.
>
> Excerpt From: Graham Hutton. “Programming in Haskell” (2nd ed.).

**Fractional**

> This class contains types whose values are instances of the numeric class Num, but in addition whose values are non-integeral, and as such support the methods of fractional division and fractional reciprocation:
>
> `(/):: a -> a -> a`
>
> `recip :: a -> a -> a`
>
> The basic types `Float` and `Double` are instances.
>
> Excerpt From: Graham Hutton. “Programming in Haskell” (2nd ed.).