Functional Programming

Types

Daniel Kröni
University of Applied Sciences Northwestern Switzerland

Learning Targets

You

- know what types are
- understand the importance of types in a programming language
- know the basic types in Haskell
- can tell the type of a function

Content

- **Basic concepts / Motivation for types**
  - Why types at all?
- **Basic types**
  - Bool, Char, String, Int, Integer, Double
- **Own Enumeration types**
  - data Color = Red | Yellow | Green
- **Tuple types**
  - (True, "Hallo", 5) :: (Bool, String, Int)
- **Own Records**
  - data Person = Person { name :: String, age :: Int }
- **Function types**
  - add :: Int -> Int -> Int
- **Type classes**

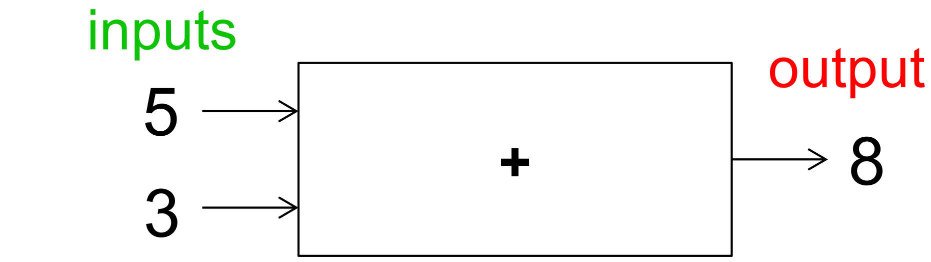
What is a Type?

- Values are represented by data – usually bits (0 | 1).
- Without proper interpretation, data is of no use. It is only a vast accumulation of bits.
- Abstraction from low level representation
- **Types add meaning to plain bits**
- Type systems prevent us from accidentally mixing up types
  E.g. **`not :: Bool -> Bool`**
  - `(not True)` is ok
  - `(not 2)` does not make sense: The function not is not applicable to Integers

A type is a set of related values

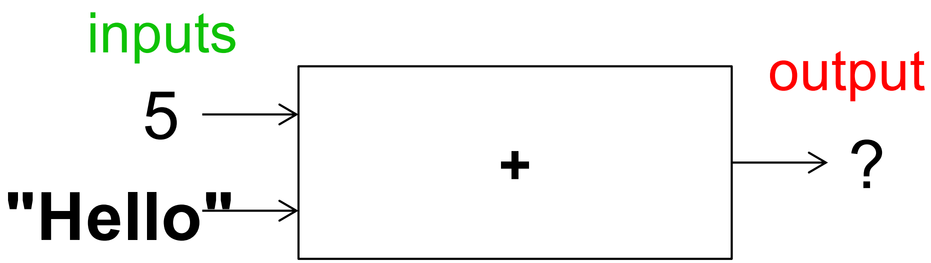
Why Types?

Last week:



- The function “+” can add two numbers and will provide the sum (a number again) as its result

What will happen if we try to



Worksheet: Types

WS_TypeBasics

Different Types of Types

Many programming languages distinguish between

- **Basic Types *(predefined Types that cannot be made up of other types)***
  - *Bool*
  - *Char*
  - *Int / Integer / Double*
- **Own Enumerations**
  - data Color = Red | Yellow | Green
- **Aggregated Types**
  - Pairs / Tuples
    - GPS Coordinates: an aggregation of (Int, Int)
  - Own Record types
    - data Person = Person { name :: String, age :: Int }
  - Lists
    - Path / Directions: the way from Brugg to Zurich as a list of GPS Coordinates / waypoints
- **Function Types (see later)**

Basic Types

- **Bool**
  - The two logical values `True` and `False`
  - The common infix operators and functions are provided
    - a && b (AND)
    - a || b (OR)
    - not a (NOT)

- **Char**
  - This type contains all single characters that are available from a normal keyboard such as 'a', 'A', '3', and '?'
  - Some characters have special meaning such as '\n' (newline) and '\t' (tab)
  - All characters are enclosed in single forward quotes `' '`

Basic Types

- **Int**
  - Fixed precision integers such as 100, -3, 0 etc.
  - Possible integers: from -2^{63} to $2^{63}-1$
  - Uses a fixed amount of memory (usually 64 bits), CPU supported, fast
- **Integer**
  - Contains **all** integers, with as much memory as necessary being used for their storage.
  - Slower computations as type is not supported by CPUs
- **Double**
  - Floating point numbers
  - Contains numbers with a decimal point such as 3.14159, -12.8
  - Uses a fixed amount of memory (usually 64 bits), CPU supported, fast

Type Ascription

```
File (~/.ghci)
:set +t
```

Expression :: Name_of_Type

Type names must start with capital letters!

Examples:

```
True :: Bool
not False :: Bool
'\n' :: Char
"c" :: String
```

Use the command **:type** or **:t** to find out the type of an expression

```
> :type True
True :: Bool
> :t not False
not False :: Bool
> :t 'c'
'c' :: Char
```

Defining your own Enumeration Types

- **The following defines two types: Color and Todo**

```
data Color = Red | Yellow | Green deriving (Show)
data ToDo = Stop | Wait | Go deriving (Show)
```

  – Color has three possible values: Red, Yellow, Green
  – ToDo has three possible values: Stop, Wait, Go

- **For example, Red is a value of type Color:**

```
*> :t Red
Red :: Color
```

- **Bool is simply a predefined enumeration type**

```
*> :info Bool
data Bool = False | True -- Defined in `GHC.Types'
```

Aggregated Types - Tuples

- A tuple is a **finite sequence** of components of **possibly different** type
- **Syntax:** component types are enclosed in round brackets and separated by commas

```
(False, 8, "Hallo")
(8, 'a', True, "Text")
```

- The type of a tuple is written as (T_1, T_2, ...,T_n) and thus defining the type at each position in the tuple
- The **number of components** in a tuple is called its **arity**

```
> :t ('a', True, 'b')
('a', True, 'b') :: (Char, Bool, Char)
> :t ( (True, '8'), ('e', False, 'x') )
( (True, '8'), ('e', False, 'x') )
  :: ((Bool, Char), (Char, Bool, Char))
```

Accessing Components – Pattern Matching

- **Pairs are tuples with only two components (of arity 2). Eg.**

```
(5, 2)
(False, 'c')
```

- **Here are two selector functions to get the first and the second component of Int pairs:**

```
fstInt :: (Int,Int) -> Int
fstInt (x, y) = x

sndInt :: (Int,Int) -> Int
sndInt (x, y) = y
```

Avoid Repetition - Polymorphic Functions

- **But fstInt does not work on pairs with different component types**

```
> fstInt (False, 'c')
```

- **There are two polymorphic selector functions in Haskell to get the first and the second component of any pairs:**

```
fst :: (a, b) -> a
fst (x, y) = x

snd :: (a, b) -> b
snd (x, y) = y
```

```
> fst (False, 'c')
False
> fst (("Hallo", 'c'), 1)
("Hallo", 'c')
```

Polymorphic Types

Does not start with capital letter, therefore no type!

```
> :t fst
fst :: (a, b) -> a
```

- **a is a type variable, it represents any type**
- **A type that contains at least one type variable is called polymorphic type ("of many forms").**
- **Thus fst is a polymorphic function.**
- **Using polymorphic types, functions can operate on many different types.**
  - Less code
  - Less error prone
  - Behavior always the same, regardless of types used

Type Synonyms

- **The keyword type can be used to introduce a new name (a synonym) for an existing type. E.g.**

```
type Coord = (Int, Int)
```

  - This does not create a new type, only a new name!
- **The synonym and the original type can be used interchangeably**

```
xCoord :: Coord -> Int
xCoord (x, y) = x
```

```
time :: (Int, Int)
time = (23, 59)
```

```
xCoord time -- compiles
```

  - Good for documentation but no help from the compiler

Worksheet: Tuples

WS_Tuples

Defining your own Record Types

- **Person is a new type describing a person**

```
data Person = Person { name :: String, age :: Int }
                                          deriving (Show)
```

Type Constructor

Value Constructor

Components

Deriving Clause

- **The Type Constructor defines the name of the new type**
- **The Value Constructor is used to create a value of this type**
- **The Components define the fields**
- **The Deriving Clause automatically derives instances for the named classes (here only Show)**

```
data Person = Person { name :: String, age :: Int } deriving (Show)
```

Defining your own Record Types

- **The Person value constructor is just a function which takes the components as arguments and creates a value of type Person**

```
*> :t Person
Person :: String -> Int -> Person
```

```
*> Person "Dani" 39
Person {name = "Dani", age = 39}
```

  – The nice string representation is due to "deriving (Show)"

- `name` **and** `age` **are accessor functions which are used to extract the fields' values**

```
*> :t age
Person -> Int
*> age (Person "Mike" 22)
22
```

Function Types

- Functions also have a type. It consists of
  - The type ot the input parameter
  - The type of the output parameter
- Functions with one input type and one output type are defined as:

```
functionname :: Inputtype -> Outputtype
```

- Examples

```
not         :: Bool -> Bool
isDigit     :: Char -> Bool
isUpperCase :: Char -> Bool
```

- Note that isDigit and isUpperCase have the same type!

Declaring our own function

- A **declaration** of a function:

```
atTrafficLight :: Color -> ToDo
atTrafficLight Red    = Stop
atTrafficLight Yellow = Wait
atTrafficLight Green  = Go
```

  - declaration by **pattern matching**
  - one equation for each possible value of the input parameter
- an **application** of a function:

```
> atTrafficLight Yellow
Wait :: ToDo
```

Function Application

- **Typing rule**

Function f with type A -> B

Name a with type A

$$\frac{\Gamma \vdash f :: A \rightarrow B \qquad \Gamma \vdash a :: A}{\Gamma \vdash f\ a :: B}\ \textbf{App}$$

Function f applied to a has Type B

- **Example**

```
g :: Int -> Char -> Bool

g 4 'a' :: Bool
g 4 :: Char -> Bool
```

https://en.wikipedia.org/wiki/Type_rule

Type Inference

```
abbreviate Red    = 'r'
abbreviate Yellow = 'y'
abbreviate Green  = 'g'
```

- The type of a function needs not to be specified
- In most cases it can be automatically inferred
- This automatism is called **type inference**

```
> :t abbreviate
abbreviate :: Color -> Char
```

Types are the most important piece of documentation!

Typeclasses

- **Polymorphic types are very handy but not all functions can be defined as generic as fst!**

  The type of the first component in a tuple is completely irrelevant in the context of fst.

- **Let's have a look at the function max that returns the larger of two values.**

```
> :t max
max :: Ord a => a -> a -> a
```

  max only makes sense for types which have a notion of order. Like Int or String.

  This can be marked by a class constraint. In this case we constrain a to be of the class `Ord` (which is a predefined Typeclass who's instances support ordering of their values)

Basic classes

- **Eq – equality types**
  - Contains types whose values can be compared for equality and inequality
  - methods: (==), (/=)
- **Ord – ordered types**
  - Contains types whose values are totally ordered
  - methods: (<), (<=), (>), (>=), min, max
- **Show – showable types**
  - Contains types whose values can be converted into strings of characters
  - method show :: a -> String
- **Num – numeric types**
  - Contains types whose values are numeric
  - methods: (+), (-), (*), negate, abs, signum

Basic classes

- **Integral –** integral types
  - Contains types that are numeric but of integral value
  - methods: div, mod
- **Fractional –** fractional types
  - Contains types that are numeric but of fractional value
  - methods: (/), recip

How to write your own classes: see later!

Computing with Numbers

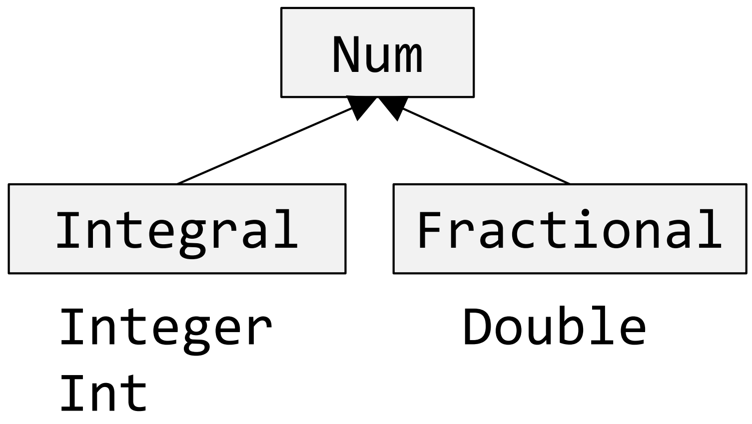
- **Haskell does not automatically convert numeric types!**
- **Numeric literals like 23 or 1.5 are polymorphic in Haskell**

```
23 :: Num a => a
```

```
1.5 :: Fractional a => a
```

```
> 3 / 1.5
2 :: Fractional a => a
```

```
> (3 :: Int) / 1.5
```

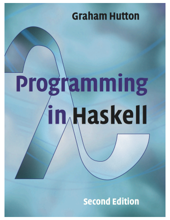


- Use fromIntegral to convert from Int / Integer back to Num a

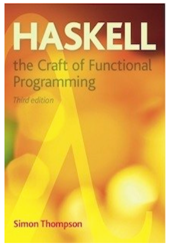
```
fromIntegral :: (Integral a, Num b) => a -> b
```

```
> fromIntegral (3 :: Int) / 1.5
2 :: Fractional a => a
```

Further Reading



Chapter 3



Chapter 3

Types and Typeclasses

http://learnyouahaskell.com/types-and-typeclasses