Table of Contents

Types

Benson Joeris
benson@bjoeris.com
http://bjoeris.com

pluralsight
hardcore developer training

Overview

- Haskell's Type System
- Exploring types in GHCi
- Explicit Types
- Type Inference
- Polymorphic Functions
- Type Class Constraints

Table of Contents

Type Systems

Static Types

- Variables have a fixed type

```
int x = "hello";
```

- C, C++, C#, Java

Dynamic Types

- Variables can store any type

```
var x = "hello";
```

- JavaScript, Python, Ruby, Perl

Type Systems

Static Types

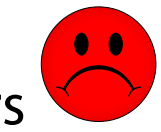
- More errors caught at compile-time

```
int x = "hello"+3;
```

- Write lots of types

Dynamic Types

- More run-time errors



```
var x = "hello"+3;
```

- Never write a type

```
typename _Alloc_traits<_Tp, _Allocator>::allocator_type get_allocator();
```

Type Systems

Static Types

```
int foo_int(int x)
{
  return 2 * x + 1;
}
float foo_float(float x)
{
  return 2 * x + 1;
}
```

Dynamic Types

```
function foo(x)
{
 return 2 * x + 1;
}
```

Type Systems

Static Types

- More errors caught at compile-time 
- Write lots of types 
- Repeated code for different types 

Dynamic Types

- More run-time errors 
- Never write a type 
- Same code works for many types 

Haskell's Type System

- Statically typed
  - *LOTS* of compile-time errors
  - Few run-time errors

- Types are inferred
  - Don't have to write out explicit types
  - Explicit types communicate with *PEOPLE*, checked by compiler

- Same code can work for many different types

Table of Contents

Exploring Types in GHCi

- `:t` –Print the type of an expression
- `a -> b` –Function taking type a as a parameter, and returning type b
- `a -> b -> c` –Function taking two parameters, of type a and b, and returning type c
  - `a -> b -> c  =  a -> (b -> c)`

Table of Contents

Explicit Type

```
str :: [Char]
str = "hello"
```

```
str = "hello"
```

Explicit Function Type

```
foo :: Int -> Int
foo x = 2 * x + 1
```

Explicit Function Type

```
add3Int :: Int -> Int -> Int -> Int
add3Int x y z = x + y + z
```

Type Annotation

```
x = 3 :: Int
```

```
y = (3 :: Int) + (2.1 :: Double)
```

```
y' = 3 + 2.1
```

Table of Contents

Type Inference

```
square x = x * x
```

```
nonsense = square "hello"
```

```
squareTwice x = square (square x)
```

```
brokenShowSquare x =
  "The square is: " ++ square x
```

Table of Contents

When to use Explicit Types

- Communicating with people

```
mystery :: [Char] -> Int
```

When to use Explicit Types

- Tracking down compiler errors

```
whats_wrong = x + y
  where x = length "hello"
        y = 6/2
```

```
whats_wrong' = x + y
  where x :: Int
        x = length "hello"
          y :: Int
          y = 6/2
```

When to use Explicit Types

- Help the compiler

```
x = show (read "123")
```

```
x' = show (read "123" :: Int)
```

When to use Explicit Types

- Optimizing Performance

```
foo = x * y + z
  where x = 32
        y = 42
        z = -5
```

```
-- slightly faster
foo :: Int
foo = x * y + z
  where x = 32
        y = 42
        z = -5
```

  - Not until you have to

When to use Explicit Types

- Communicating with people
- Tracking down compiler errors
- Helping the compiler (occasionally)
- Optimizing (only when necessary)

Table of Contents

Polymorphic Functions

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

length_chars :: [Char] -> Int
length_chars [] = 0
length_chars (x:xs) = length_chars xs + 1
```

```
length [] = 0
length (x:xs) = length xs + 1
```

Polymorphic Functions

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

- Polymorphic Function –A function with a type variable
  - Not object-oriented polymorphism
  - Similar to C#/Java generics or C++ templates

Polymorphic Functions

```
length :: [hello_123] -> Int
length [] = 0
length (x:xs) = length xs + 1
```

- Type variables –start lower case
  - `a, b, x, foo, hello_123`
- Concrete types –start Upper case
  - `Int, Integer, Char, Double`

Polymorphic Functions

```
empty_list :: [a]
empty_list = []
```

```
list_double = 3.2 : empty_list
list_char = 'a' : empty_list
```

Polymorphic Functions

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

- Repeated type variables always represent the same type

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

- Different type variables can represent different types

```
bad = (badHead [1.3,2,4]) : "foo"
```

Table of Contents

Type Class Constraints

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

```
badSum :: [a] -> a
badSum [] = 0
badSum (x : xs) = x + sum xs
```

```
sum [] = 0
sum (x : xs) = x + sum xs
```

Type Class Constraints

```
sum :: Num a => [a] -> a
sum [] = 0
sum (x : xs) = x + sum xs
```

Type Class Constraints

```
show :: Show a => a -> [Char]
```

Type Class Constraints

```
showSum :: (Num a, Show a) => [a] -> [Char]
showSum xs = show (sum xs)
```

Type Class Constraints

Table of Contents

Summary

- Static vs Dynamic Typing
- GHCi Type Exploration
- Explicit Types
- Type Inference
- Polymorphic Functions
- Type Class Constraints