

Session 3: Types and classes II

COMP2221: Functional programming

Lawrence Mitchell*

*`lawrence.mitchell@durham.ac.uk`

Recap

- Idea that variables, *and functions* have *types*
- Saw some basic Haskell types
 - **Bool**
 - **Int, Integer, Float**
 - **Char**
 - *tuples* (a, b, c) and *lists* [a]
- Discussed *currying* of functions.

```
-- "uncurried"
```

```
add' :: (Int, Int) -> Int
```

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

```
-- "curried"
```

```
add'' :: Int -> (Int -> Int)
```

```
add'' x y = x + y
```

Typically preferred.

Currying conventions (reminder)

- (Almost) all functions in Haskell are written in *curried* form
- ⇒ To avoid messy syntax, this leads to associativity rules for `->` and function application.

first argument.

`->` associates to the *right*

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

```
-- Means
```

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

final return value

Function *application* associates to the *left*

```
mult x y z
```

```
-- Means
```

```
((mult x) y) z
```

```
:: Int
```

$mult\ x :: Int \rightarrow (Int \rightarrow Int)$

$(mult\ x)\ y :: Int \rightarrow Int$

Type inference

Generalization of "Hindley-Milner" type inference. "Algorithm-W".

- Any type declaration you write will be *checked* by the type inference engine. Error if incorrect

```
foo :: Int -> Bool
```

```
foo x = x + 3
```

error:

- Couldn't match expected type `Bool` with actual type `Int`
- In the expression: `x + 3`
- In an equation for `foo`: `foo x = x + 3`

$(+) :: \text{Num } a \Rightarrow a \rightarrow a \rightarrow a$

$(+) :: \text{Int} \rightarrow \text{Int} \rightarrow \text{Int}$

Python: `True + 3 = 4`

$\text{int}(\text{True}) = 1$.

$1 + 1.5$

$(\text{fromIntegral } 1) + 1.5$

hoogle.haskell.org -

Recommendation

Reasoning about types is a core part of understanding (and writing) Haskell code.

Char unicode?

⇒ always decorate function definitions with their type.

Syntax conventions

- Function application is so *important* that it is written as quietly as possible: with whitespace
- All functions can be called in *prefix* form:
“foo a b”, not “a foo b”
- ...but, special syntax for binary functions.

Binary functions: infix notation

Infix notation

All binary functions (which have type $a \rightarrow b \rightarrow c$) can be written as *infix* functions.

Symbol only names

Names consisting *only* of symbols (e.g. `+`, `*`)

```
1 + 2    -- infix notation
(+) 1 2  -- prefix notation
False && True  -- infix notation
(&&) False True  -- prefix notation
```

“Normal” names

Names with alpha-numeric characters (e.g. `div`, `mod`)

```
mod 3 2    -- prefix notation
3 `mod` 2  -- infix notation using backticks
```

Summary

- Functions defined by “equations” that match patterns:

```
head' [] = []
```

```
head' (x:xs) = x
```

“Where-ever you see `head' []` replace it with `[]`”

- No *side effects* \Rightarrow substitution is always safe/correct.
- Patterns are tried textually in order down the page.
- Guards can be used to constrain when equations can match

```
signum n | n > 0      = 1      if then else.  
         | n == 0     = 0  
         | otherwise = -1
```

Guard can be any expression that evaluates to a **Bool** value.

Compare

$$s(x) = \begin{cases} 1 & x > 0 \\ 0 & x = 0 \\ -1 & \text{otherwise} \end{cases}$$

Building block summary

- Prerequisites: none
- Content
 - Defining functions as “equations”
 - Pattern matching in equations
 - Guards and conditional expressions
 - Special syntax for infix notation (binary functions)
- Expected learning outcomes
 - student can *write* functions using conditional expressions and guard expressions
 - student *understands* order in which patterns are tried in matching
- Self-study
 - None

Polymorphism

"Many forms"

Polymorphism

- Recall, Haskell is *strictly typed*.
- What does this mean for (say) `length`?

Different types?

```
length [True, False, True] -- :: [Bool] -> Int ?  
length [1, 2, 3]           -- :: [Int] -> Int ?
```

These functions must have *different* types, no?

(After compilation).
every thing must be
monomorphic → "single form".

Polymorphism

- Recall, Haskell is *strictly typed*.
- What does this mean for (say) `length`?

Different types?

```
length [True, False, True] -- :: [Bool] -> Int ?  
length [1, 2, 3]           -- :: [Int] -> Int ?
```

These functions must have *different* types, no?

Polymorphic types

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

“`length` eats a list of values of any type `a` and returns an `Int`”

`a` is called a *type variable*.

This is called *parametric polymorphism*.

Contrast with OO languages: definitions

60s Chris Strachey.

Definition (Parametric polymorphism)

$\text{length} :: [a] \rightarrow \text{Int}$.

Write a *single* implementation of a function that applies generically and *identically* to values of any type.

Definition (“ad-hoc” polymorphism)

Write *multiple* implementations of a function, one for each type you wish to support.

Definition (Subtype polymorphism) \rightarrow classic OO.

Relate datatypes by some “substitutability”. Write a function for a supertype instance. Now all subtypes can use it.

“Duck typing” or “Liskov substitution principle”.

Contrast with OO languages: examples

Subtype polymorphism

```
class Foo(object):
    def length(self, ...):
        pass
class Bar(Foo):
    pass
a = Foo().length()
# Every Bar is-a Foo, so we can
# call the length method.
b = Bar().length()
```

FOO
↓
BAR

Parametric polymorphism

```
-- length doesn't care what type the entries
-- in the list are
length :: [a] -> Int
length [] = 0
length (_,xs) = 1 + length xs
```

Ad-hoc polymorphism

```
class Foo(object):
    pass
class Bar(object):
    pass
def length(obj):
    if isinstance(obj, Foo):
        ...
    elif isinstance(obj, Bar):
        ...
# length knows how to handle things
# of type Foo and type Bar
a = length(Foo())
b = length(Bar())
```

Contrast with OO languages

- Parametric polymorphism also called *generic programming*
- Introduced in ML in 1975.
- Has been adopted by a number of languages, including traditional OO ones.
- For example, Java or C# have “generics” for this purpose

```
// Implementation of HashSet is generic  
// Specialised on instantiation  
Set<int> intset = new HashSet<int>();  
Set<Object> objset = new HashSet<Object>();
```

Java Generics
Joshua Bloch
(90s)

- C++ templates also allow for similar style of programming

Constraining polymorphic functions

- Some polymorphic functions only apply to types that satisfy certain constraints
- For example (+) works on all types a , as long as that type is a number type.

Phil Wadler (1989).

Example

→ "type class".

$(+) :: \text{Num } a \Rightarrow a \rightarrow a \rightarrow a$

"For any type a that is an *instance* of the class **Num** of numeric types, (+) has type $a \rightarrow a \rightarrow a$ "

- This constraint is called a *class constraint*
- An expression or type with one or more such constraints is called *overloaded*.

type signature

$\Rightarrow \text{Num } a \Rightarrow a \rightarrow a \rightarrow a$ is an *overloaded type* and (+) is an *overloaded function*.

$(\text{Num } a, \text{Ord } a) \Rightarrow [a] \rightarrow [a]$
type constraints

Haskell classes

WARNING!

The *words* class and instance are the same as in object-oriented programming languages, but their *meaning* is very different.

Definition (Class)

A collection of *types* that support certain, specified, overloaded operations called *methods*.

Defines an interface

Definition (Instance)

A concrete type that belongs to a *class* and provides implementations of the required methods.

Implements the interface.

- Compare: type “a collection of related values”
- This is *not* like subclassing and inheritance in Java/C++
- Closest to a combination of Java *interfaces* and *generics*
- C++ “concepts” (in C++20) are also very similar.

Rust Traits are similar.

Defining classes I

- Let us say we want to encapsulate some new property of types **Foo**-ness

- We define the interface the type should support

```
class Foo a where  
  isfoo :: a -> Bool
```

- Now we say how types implement this

```
instance Foo Int where  
  isfoo _ = False
```

```
instance Foo Char where  
  isfoo c = c `elem` ['a'..'c']
```

- Can add new interfaces to old types, and new types to old interfaces.
- Contrast Java, where if I implement a new interface it is very difficult to make existing classes implement it.

↳ have to go and edit them all.

Defining classes II

- Classes (interfaces) can provide default implementation.
- Example, the **Eq** class representing equality requires both `(==)` and `(/=)`.
- Since $a == b \Leftrightarrow \text{not } (a /= b)$, we can provide *default* implementations and only require that an instance implements one.

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

```
-- instance for MyType only needs to provide one of (==) or (/=).
instance Eq MyType where
  x == y = ...
```

Building block summary

- Prerequisites: none
- Content
 - Looked at Haskell *classes* in the context of overloaded functions
 - Looked at generic programming (*polymorphism*) in Haskell
 - Defined *overloading* in terms of constrained polymorphism
 - Looked at constrained polymorphism and class constraints.
- Expected learning outcomes
 - student *knows* definition of generic programming and overloading as applied in Haskell
 - student can *write* simple polymorphic code in Haskell
 - student *understands* some differences between Haskell-style overloading, and Java-style subclassing
- Self-study
 - (Optional, but interesting). Wadler & Blott, *How to make ad-hoc polymorphism less ad hoc*, POPL (1989). <https://people.csail.mit.edu/dnj/teaching/6898/papers/wadler88.pdf>
 - (Optional, probably the first 45 minutes only?). Simon Peyton-Jones on type classes <https://www.youtube.com/watch?v=6COvD8oynmI>.