

Programming Languages, CPSC 604—Slides 4  
About Haskell

Jaakko Järvi

January 28, 2009

# Haskell basics

- Modern (pure) lazy functional language
- Statically typed, supports type inference
- Compilers and interpreters:  
<http://www.haskell.org/implementations.html>
  - Hugs interpreter
  - GHC Compiler
- A peculiar language feature: indentation matters
- Also: capitalization matters

# Type inference

- `x = 1 + 2`  
1 has type `Integer`, 2 has type `Integer`, adding two `Integers` results in another `Integer`  $\Rightarrow x : Integer$ .<sup>1</sup>
- `inc x = x + 1`  
With similar reasoning, `inc : Integer  $\rightarrow$  Integer`
- Explicit type annotations are possible:  
`inc :: Integer  $\rightarrow$  Integer`  
`inc x = x + 1`

---

<sup>1</sup>Actually, from above, a type class `Num` is inferred, not the exact `Integer` type

# Cons lists

- The data-structure for almost everything is `List`
- Constructing lists:

```
[]           -- empty list  
[1]          -- list with one element  
[1, 2, 3]    -- a longer list
```

- List patterns:
  - `x:xs` matches to any list with one or more elements
  - `x:y:z:xs` matches to any list with three or more elements
  - `[x]` matches to any list with one element
  - `[]` matches to empty list

```
let x:xs = [1, 2, 3]  
    -- x is 1  
    -- xs is [2, 3]
```

## Defining functions

- Defined as equations (with pattern matching)

```
len1 :: [a] → Integer
len1 []           = 0
len1 (x:xs)      = 1 + len1 xs
```

- With lambda abstraction

```
len2 :: [a] → Integer
len2 = \x → if (null x) then 0 else 1 + (len2 (tail x))
```

- Note the function invocation syntax:

```
(len1 [1, 2, 3])
```

# Currying

- Haskell functions are *curried*

```
add :: Int → Int → Int  
add x y = x + y
```

```
add3 :: Int → Int  
add3 = add 3
```

```
z :: Int  
z = add3 4
```

# Pure

- No side effects (e.g. no assignment). For example, in

```
x = add 1 2
```

a fresh variable `x` is bound to the (uncomputed) value of `add 1 2` and stays bound to that until it goes out of scope.

⇒ Haskell functions are *pure* mathematical functions

- Makes reasoning about programs more feasible
- Note that side-effects are necessary for any realistic programming (for IO, efficiency, ...). Haskell type system encapsulates all effects inside *monads*

# Lazy

- Lazy evaluation (call-by-need):
  - Never evaluate an expression, unless it's value is needed
- Example: The following program is not erroneous.

```
omit x = 0
```

```
v = omit (1/0)
```

```
main = putStr (show v)
```

# Parametrically polymorphic functions

- Examples:

```
id :: a → a
```

```
id x = x
```

```
length :: [a] → Int
```

```
length [] = 0
```

```
length (x:xs) = 1 + length xs
```

```
tail :: [a] → [a]
```

```
tail [] = []
```

```
tail (x:xs) = xs
```

```
eval :: (a → b) → a → b
```

```
eval f x = f x
```

- Note syntax for type parameters

## Type classes

- Consider now a non-parameterically polymorphic function.

```
not_equal :: a → a → Bool ???
```

```
not_equal x y = if (x == y) then False else True
```

- There are requirements for `a`
- Not all `a`'s will be acceptable. The type bound to `a` must be equality comparable
- `a` must be an instance of the type class `Eq`

```
not_equal :: Eq a ⇒ a → a → Bool
```

```
not_equal x y = if (x == y) then False else True
```

# Type classes

- Primary motivation: Function overloading mechanism for Haskell (ad-hoc polymorphism)<sup>2</sup>
  - Parametric polymorphism: one implementation covers all types
  - Ad-hoc polymorphism: same syntax for different implementations

---

<sup>2</sup>Wadler, Blott: “How to Make Ad-Hoc Polymorphism Less Ad Hoc”, 1988

## Type classes

- Type classes represent a set of requirements
- Requirements are expressed as function signatures
- Default implementations for each signature can be provided
- Example:

```
class Eq a where
  (==), (/=) :: a → a → Bool
  x /= y      = not (x == y)
  x == y      = not (x /= y)
```

- The class definition can be read as:
  - “A class of types that conforms to the specified interface”
- Note how the declaration of conformance is separate from the definition of a type (unlike, say, `implements` in Java)

## Instance declarations

- Members of type classes are called *instances*.
- A type is not an instance of a type class unless explicitly stated — nominal conformance.

```
instance Eq Bool where
  True  == True    = True
  False == False   = True
  _     == _       = False
```

- This would be painful without parameterized instance declarations, referred to as *conditional instance declarations*. Example:

```
instance Eq a => Eq [a] where
  []      == []      = True
  (x:xs) == (y:ys)  = x==y && xs==ys
  _       == _       = False
```

- `Eq a =>` is the *context* (constraint).

## Constraining polymorphic functions

- If a function is not explicitly annotated with its type, constraints will be deduced with type inference

```
not_equal x y = if (x == y) then False else True
```

- From `x == y` it can be inferred that the types of `x` and `y` must be instances of `Eq`, and they must be of the same type.
- The type of `not_equal` is thus deduced to:

```
not_equal :: Eq a => a -> a -> Bool
```

- Note that inference leads to the least constrained function type (*principal type*).
- Apart from small locally defined functions, type annotations should be used (and sometimes must to help the type inference process)
- Consequence of type inference: **a particular function name, such as `==` can only be required by one type class.**

## Type class inheritance

- “Inheritance” in type classes is comparable to extending interfaces in Java
- Accomplished with conditional class definitions.
- The same syntax `Eq a` for expressing the context is used.

```
class Eq a ⇒ Ord a where
  (<), (<=), (>), (>=) :: a → a → Bool
  max, min           :: a → a → a
  compare            :: a → a → Ordering
```

- To be an instance of `Ord`, type must meet the signature requirements listed in `Ord`, and in addition be an instance of `Eq`.
  - An instance declaration that makes a type an instance of `Ord` does not establish that the type is an instance of `Eq`!

## Multiple type class constraints

- A single type parameter can be constrained with several type classes.
- E.g. a function that needs to compare values, and also show them as strings:

```
class Show a where  
  show :: a → String
```

```
show_min :: (Ord a, Show a) ⇒ a → a → String  
show_min x y = show (min x y)
```

# Modules and Abstract Data Types

- `data` vs. `newtype` vs. `type`
- records, tuples, lists
- `import`