Polymorphism

COMP 524: Programming Language Concepts
Björn B. Brandenburg

The University of North Carolina at Chapel Hill

Based in part on slides and notes by S. Olivier, A. Block, N. Fisher, F. Hernandez-Campos, and D. Stotts.

Static Type Checking & Redundancy

Assumptions so far.

- ➡ Each name is bound to exactly one entity (e.g., a subroutine).
- ➡ **Static** typing: every entity has a **specific** type.

Suppose we wanted to extract the first element of a 2-tuple.

- ➡ Easy in Prolog or Python.
  - **Dynamic** type checking: no type violation at runtime.
- ➡ Hard to do in (basic) Haskell or Java (if it had tuples).
  - What is the **type** of the **first** element?
  - What is the **type** of the **second** element?
  - What is the **type** of `getFirst`?

Idea: Type Variables

Problem with specific types.

➡ **Unnecessarily** constrained.

‣ E.g., tuple de-structuring does not depend on type, so why have restrictions?

What if we could write it for "any" type?

➡ Analogy: arithmetic with numbers vs. arithmetic with variables.

➡ Raises level of abstraction.

‣ Often called **generic programming**.

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

Idea: Type Variables

Problem with specific types.

- ➡ **Unnecessarily** constrained.
  - E.g., tuple de-structuring does not depend on type, so why have

Haskell: lower-case letters are type variables.
`getFirst` is defined for all types a and b without specific restrictions, i.e. **any** type.

Wh

- ➡ A
- ➡ Raises level of abstraction.
  - Often called **generic programming**.

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

Parametric Polymorphism

Parametrized subroutines.

- Defined in terms of one or more **type parameters**.
- "Subroutine recipe:" how to define a **specific instance** of the family of subroutines **given specific types**.

Implementation.

- Compiler can **generate type-specific versions**.
  - Or, if possible, code that works with any type (e.g., getFirst).
- **Type checking** becomes more complicated.
  - In fact, with certain kinds of polymorphism, type system can be come **undecidable** (for details see grad school).

Widespread in modern imperative languages.

- Often called **generic programming**.

Type Classes

What is the type of multiplication?

➡Can take any two **numbers**.

‣There are many number types: Int, Float, …

➡But not just **any type**.

‣E.g., addition of tuples not (uniquely) defined.

Idea: type restrictions.

➡Multiplication defined for all types **such that** the type **is a number**.

```
> :t (*)
(*) :: (Num a) => a -> a -> a
```

Type Classes

Haskell: if **a** is a member of the **type class Num**…

➡But not just **any type**.

‣E.g., addition (u

…then…

…multiplication is defined as function that maps 2 **a**s to one **a**.

Idea: type restrictions

➡Multiplication defined for all ty type **is a number**.

```
> :t (*)
(*) :: (Num a) => a -> a -> a
```

Polymorphic Types

Composite types with type variables.

- ➡ Some data structures are **defined for any type**.
  - List, Tree, Map, Stack, etc.
    - "a X of Y", e.g., "a List of Int"
- ➡ **Generic** or **parametrized** types.
- ➡ Heavily used in collection libraries.

```
data Tree a = Nil
            | Node { left  :: Tree a
                   , value :: a
                   , right :: Tree a
                   }
```

Polymorphic Types

Haskell: Tree type is parametrized.

‣ "a X of Y

➡ **Generic** o

Type parameter used for components.

➡ Heavily used in collection libraries.

```
data Tree a = Nil
            | Node { left  :: Tree a
                   , value :: a
                   , right :: Tree a
                   }
```

Ad-Hoc Polymorphism / Overloading

What about multiplication in Java?

- ➡Defined for a **few specific types**.
- ➡Uses same symbol '*'.

Overloading.

- ➡Same name is used for multiple bindings.
- ➡Disambiguated based on types.
- ➡**Context-independent**: only parameter types used for disambiguation.
- ➡**Context-dependent**: parameter types may be ambiguous if return type is unambiguous.

Ad-Hoc Polymorphism / Overloading

What about multiplication in Java?

- ➡Defined for a **few specific types**.
- ➡Uses same symbol ‘*’.

> **Haskell**: ad-hoc polymorphism is not supported; polymorphic code is required to use type classes.

for disambiguation.

- ➡**Context-dependent**: parameter types may be ambiguous if return type is unambiguous.

Type Classes in Haskell

Definition of a type.

- ➡A set of values.
- ➡A **set of operations** that can be applied to values of the types.

Definition of a type class.

- ➡A **set of types** that for which a number of **standard operations** is declared.
  - ‣e.g., "every **Numeric** type must support addition"
- ➡Haskell's way of **controlling overloading**.
  - ‣A function can only be overloaded if it is defined by a type class.

Type Classes in Haskell

Common Type Classes

`Eq` — values can be tested for equality (`==`, `/=`)
`Ord` — values are ordered (`<`, `<=`, `>`, `>=`, `max`, `min`)
`Show` — can be converted to string (`show`)
`Read` — can be parsed from a string (`read`)
`Num` — a numeric type (`+`, `-`, `*`, `negate`, `abs`, `signum`)
`Integral` — integers (`mod`, `div`)
`Fractional` — divisible numbers (`/`, `recip`)

a type class.

Defining a Type Class

```
-- Minimal complete definition: either '==' or '/='.
--
class Eq a  where
    (==), (/=)                :: a -> a -> Bool

    x /= y                    = not (x == y)
    x == y                    = not (x /= y)
```

http://www.haskell.org/ghc/docs/latest/html/libraries/base-4.2.0.0/Prelude.html#t%3AEq

Type Class Definition.

- ➡Specifies a **name**.
- ➡**Required operations** (+ types!)
- ➡**Default implementations**.

Defining a Typ

Define name.

```
-- Minimal complete defin   tion: either '==' or '/='.
--
class Eq a  where
    (==), (/=)           :: a -> a -> Bool

    x /= y               = not (x == y)
    x == y               = not (x /= y)
```

http://www.haskell.org/ghc/docs/latest/html/libraries/base-4.2.0.0/Prelude.html#t%3AEq

Type Class Definition.

➡Specifies a **name**.

➡**Required operations** (+ types!)

➡**Default implementations**.

Defining a Type Class

```
-- Minimal complete definition: either '==' or '/='.
--
class Eq a  where
    (==), (/=)           :: a -> a -> Bool

    x /= y               = not (x == y)
    x == y               = not (x /= y)
```

http://www.haskell.org/ghc/docs/latest/html/libraries/base-4.2.0.0/Prelude.html#t%3AEq

Type Class Definition.

➡Specifies a **name**.

➡**Required operations** (+ types

➡**Default implementations**.

Required operations and associated types.

Default Implementations:
User can specify either function, the missing one uses the default implementation. If user provides both, then default is overruled.

```
class Eq a  where
    (==), (/=)              :: a -> a -> Bool

    x /= y                  = not (x == y)
    x == y                  = not (x /= y)
```

http://www.haskell.org/ghc/docs/latest/html/libraries/base-4.2.0.0/Prelude.html#t%3AEq

Type Class Definition.

- ➡Specifies a **name**.
- ➡**Required operations** (+ types!)
- ➡**Default implementations**.

Declaring a Type Class Instance

adding a type to a type class

```
data Reply = Yes | No | Maybe

repl_equal :: Reply -> Reply -> Bool
repl_equal Yes Yes     = True
repl_equal No No       = True
repl_equal Maybe Maybe = True
repl_equal _  _        = False

instance Eq Reply where
    (==) = repl_equal
```

Define functions + instance.

- ➡**Define appropriate functions** like any other function.
- ➡Add an **instance declaration** to overload type class symbols.

Declaring a Type Class Instance

adding a type to a type class

```
data Reply = Yes | No | Maybe

repl_equal :: Reply -> Reply -> Bool
repl_equal Yes Yes     = True
repl_equal No No       = True
repl_equal Maybe Maybe = True
repl_equal _  _        = False

instance Eq Reply where
    (==) = repl_equal
```

Simple Algebraic Type
(works for any type)

Define functions + instance.

➡**Define appropriate functions** like any other function.

➡Add an **instance declaration** to overload type class symbols.

Simple Equality Function
can be arbitrarily complicated

ss Instance

class

```
data Reply = Yes | No | Maybe

repl_equal :: Reply -> Reply -> Bool
repl_equal Yes Yes     = True
repl_equal No No       = True
repl_equal Maybe Maybe = True
repl_equal _  _        = False

instance Eq Reply where
    (==) = repl_equal
```

Define functions + instance.

- ➡**Define appropriate functions** like any other function.
- ➡Add an **instance declaration** to overload type class symbols.

instance declaration
add equations to standard operations
missing symbols will use default impl.

```
data Reply = Yes | No | Maybe

repl_equal :: Reply -> Reply -> Bool
repl_equal Yes Yes     = True
repl_equal No No       = True
repl_equal Maybe Maybe = True
repl_equal _  _        = False

instance Eq Reply where
    (==) = repl_equal
```

Define functions + instance.

- ➡**Define appropriate functions** like any other function.
- ➡Add an **instance declaration** to overload type class symbols.

Deriving Standard Classes

compiler-generated instances

Repetition.

➡Some type class instances almost always look the same.

➡E.g., **Eq**, **Show**, **Read**, …

➡Defining such instances over and over is tedious.

Derived instances.

➡Built-in support for some **special** type classes.

➡Tell compiler to generate appropriate code.

```
data Reply = Yes | No | Maybe
             deriving (Eq)
```

Type Class Hierarchy

Generalizations.

- ➡Some type classes have a hierarchical relationship.
- ➡E.g., an **`Integral`** type should also a **`Num`** type.
- ➡This can be required in the type class definition.
  - ‣Enforced by compiler.

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

Type Class Hierarchy

Hierarchy:
Every ordered type must also have a concept of equality.

Generalizations.

- ➡Some type classes
- ➡E.g., an **`Integral`** type should also a **`Num`** type.
- ➡This can be required in the type class definition.
  - ‣Enforced by compiler.

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

Polymorphic Instances

How to declare instances for polymorphic types?

```
data Tree a = Nil
            | Node { val :: a, left :: Tree a, right :: Tree a}
```

Tree node equality.

- ➡Nil equals nil.
- ➡Node equals node if **values are equal** and subtrees are equal.
  - ‣What if **a** is not actually in **Eq**?

```
instance (Eq a) => Eq (Tree a) where
    Nil           == Nil           = True
    Node v1 l1 r1 == Node v2 l2 r2 = v1 == v2 && l1 == l2 && r1 == r2
    _             == _             = False
```

Polymorphic Instances

How to declare instances for polymorphic types?

```
data Tree a = Nil
            | Node { val
```

Polymorphic Instance:
Instance only defined for types with equality; undefined otherwise.

Tree node equality.

- ➡Nil equals nil.
- ➡Node equals node if **values are equal** and subtrees are equal.
  - ‣What if **a** is not actually in **Eq**?

```
instance (Eq a) => Eq (Tree a) where
    Nil             == Nil             = True
    Node v1 l1 r1 == Node v2 l2 r2 = v1 == v2 && l1 == l2 && r1 == r2
    _               == _               = False
```