中国科学技术大学
University of Science and Technology of China

Overloading and Type Classes

(Adhoc Polymorphism)

《程序语言设计和程序分析》

张昱

yuzhang@ustc.edu.cn

中国科学技术大学

计算机科学与技术学院

References

- D. Rémy(Cambium project-team): Type systems for PLs
  - Chapter 7 Overloading
- [Concepts in PLs] Revised Chapter 7 Type Classes
- PFPL
  - Chapter 44 Type Abstractions and Type Classes
- Papers
  - [ESOP 1988] Parametric Overloading in Polymorphic PLs
  - [POPL 2007] Modular Type Classes
- Implementation
  - Implementing, and Understanding Type Classes
  - Implementing type classes as OCaml modules
- Types and Propositions:
  - [TPHOLs 1997] Type classes and overloading in higher-order logic

Outline

- Parametric Polymorphism vs. Overloading
- Why Overloading
- Overloading Mechanisms
  - Static / dynamic resolution of overloading
- Parametric Overloading and Type Classes

  also known as bounded polymorphism, or type classes

  - Dictionary passing
  - Macro
  - Intentionally type analysis

Parametric Polymorphism vs. Overloading

Parametric polymorphism

- Single algorithm for *any* type

  If $f: t \to t$, **then** $f: \mathrm{int} \to \mathrm{int}$, $f: \mathrm{bool} \to \mathrm{bool}$, **…**

Overloading

- Single symbol may refer to different algorithms/operations.
- Each algorithm may have different unrelated type.
- Choice of algorithm determined by type context.

Parametric overloading

- The types being instances of a single type expression over some extended set of type variables

  $+$ has types $\mathbf{int} \to \mathbf{int} \to \mathbf{int}$, $\mathbf{float} \to \mathbf{float} \to \mathbf{float}$,

  but not $X \to X \to X$ for any X.

Why Overloading ?

- Many useful functions are not parametric
- Can list membership work for any type?

$$\text{member} : \forall X. X \text{ list} \rightarrow X \rightarrow \text{bool}$$

- Can list sorting work for any type?

$$\text{sort} : \forall X. X \text{ list} \rightarrow X \text{ list}$$

Why Overloading ?

- Many useful functions are not parametric
- Can list membership work for any type?

$$\text{member} : \forall X. X \text{ list} \rightarrow X \rightarrow \text{bool}$$

  - No! Only for types X that support equality.

- Can list sorting work for any type?

$$\text{sort} : \forall X. X \text{ list} \rightarrow X \text{ list}$$

  - No! Only for types X that support ordering.

Variants of Overloading

- **Static overloading: *static* resolution strategy**
  - Simple semantics: meaning determined statically
  - Does not increase expressiveness
  - Reduce verbosity, increase modularity and abstraction
- **Dynamic overloading**
  - meaning determined dynamically
  - Increase expressiveness
  - Extra mechanism to support the dynamic resolution
    - Require full or partial type info., or some type-related info.

中国科学技术大学
University of Science and Technology of China

Overloading Mechanisms

Static Overloading

- **Approach 1:**
  A function containing overloaded symbols => multiple functions
  - e.g. double x = x + x
    defines two versions: Int -> Int and Float -> Float

But, how to resolve

doubles (x, y, z) = (double x, double y, double z)

- 8 possible versions!

=> *Exponential growth in number of versions*

Static Overloading

- **Approach 2 (used in SML-MLton):**

  restrict the definition, i.e., specify one of the possible versions as the meaning

  - e.g. double x = x + x => double: Int -> Int

    double 3 √ double 3.2 ✖

  If you want double: Float -> Float, you need define the function explicitly specifying type.

- **In Java**
  - Overloading a method in a class => static resolution
  - But if an argument has a runtime type that is subtype of the compile-time time => dynamic resolution

Dynamic Overloading

Resolution with a type passing semantics

Runtime type dispatch using a general *typecase* construct

- High runtime cost of *typecase* unless type patterns are significantly restricted

Resolution with a type erasing semantics

To avoid the expensive cost of *typecase*,

restrict the overloaded functions by using tags.

$$\text{let } f = \lambda x.x + x \text{ in } [\,]$$

e.g. Dictionary passing

can be elaborated into

$$\text{let } f = \lambda(+).\lambda x.x + x \text{ in } [\,]$$

f 1.0 is then elaborated to f (+.) 1.0

Parametric Overloading

□ Overloading Equality

1. **Equality was overloaded as an operator.**

   But *member* using '==' does not work in general

$$\text{member } [\,]\ y = \text{False}$$
$$\text{member } (x : xs)\ \ y = (x == y)\ ||\ \text{member } xs\ \ y$$
$$\text{member } [\,1,\,2,\,3]\ \ 32\ \ \surd$$
$$\text{member "Haskell" 'k'}\ \ ✖$$

Parametric Overloading

Overloading Equality

1. **Equality was overloaded as an operator.**
   But *member* using '==' does not work in general
2. **Make type of equality fully polymorphic (Miranda)**

   (==) :: t -> t-> Bool

thus *member* is polymorphic, member:: [t] -> t-> Bool

If t does not provide a definition of equality, then there is a runtime error when equality applied to a value of type t.

=> Violate principle of abstraction

Parametric Overloading

University of Science and Technology of China

Overloading Equality

1. **Equality was overloaded as an operator.**
   But *member* using '==' does not work in general
2. **Make type of equality fully polymorphic (Miranda)**
3. **Make equality polymorphic in a limited way (used in current SML)**

(==) :: ''t -> ''t-> Bool ''t indicate t is an eqtype variable

member has precise type, i.e. [''t] -> ''t -> Bool

if t does not support equality, there will be a **static** error

Parametric Overloading

Overloading Equality

1. **Equality was overloaded as an operator.**
   But *member* using '==' does not work in general
2. **Make type of equality fully polymorphic (Miranda)**
3. **Make equality polymorphic in a limited way (used in current SML)**

(==) :: ''t -> ''t-> Bool ''t indicate t is an eqtype variable

member has precise type, i.e. [''t] -> ''t -> Bool

if t does not support equality, there will be a **static** error

Equality is a special case,
how can we generalize overloading?

Type Classes

- **Type classes are a mechanism in Haskell**
  - **Generalize eqtype to user-defined collections of types (called *type classes*)**

```
member:: (a-> a-> Bool) -> [a] -> a-> Bool
member cmp [] y = False
member cmp (x : xs) y = (cmp x y) || member cmp xs y
```

- **Dictionary-passing style implementation [ESOP1988]**
  - Type-class declaration – dictionary
  - Name of a type class method – label in the dictionary
  - Parametric overloading
    - pass the dictionary to the function

https://okmij.org/ftp/Computation/typeclass.html

Examples: Dictionary Passing

Haskell

```
class Show a where
  show :: a -> String

instance Show Bool where
  show True = "True "
  show False = "False "

instance Show Int where
  show x = Prelude.show x --
internal
```

In Haskell

- Show a is type class
- Show Bool and Show Int are instances of Show.

OCaml

```
type 'a show = {show: 'a ->
string}

let show_bool : bool show =
   {show = function
      | true -> "True "
      | false -> "False"}

let show_int : int show =
   {show = string_of_int}
```

In OCaml

- 'a show is dictionary
- show_bool and show_int are labels in the dictionary.

Examples: Dictionary Passing

University of Science and Technology of China

Haskell

```
class Show a where
  show :: a -> String

instance Show Bool where
  show True = "True  “
  show False = "False  “

instance Show Int where
  show x = Prelude.show x -- internal
```

Define an overloaded function print:

```
print :: Show a => a -> IO ()
print x = putStrLn $ show x

test_print :: IO ()
test_print = print True
```

OCaml

```
type 'a show = {show: 'a -> string}

let show_bool : bool show =
   {show = function
      | true -> "True  “
      | false -> "False"}

let show_int : int show =
   {show = string_of_int}
```

```
let print : 'a show -> 'a -> unit =
   fun {show=show} x -> print_endline (show x)

let test_print : unit =
   print show_bool true
```

- print is a **restricted** polymorphic function, and it applies to values *whose types are showable*
- In Haskell: Show Bool and Show Int are members of Show class.
- In OCaml: the evidence of being showable, the dictionary, is the explicit argument.

More Examples

- **Type class whose methods have a different of overloading: e.g. Num**
- **An instance with a constraint:**

  e.g. a Show instance for all list types [a] where the element type a is also restricted to be a member of Show.

  show_list: ‘a show -> ‘a list show (OCaml)
- **A class of comparable types**

  e.g. class Eq a (Haskell) or type 'a eq (OCaml)
- **Polymorphic recursion**

See http://okmij.org/ftp/Computation/typeclass.html#dict

Other Implementations

- **Type classes as macros**
  - **Static monomorphization (compile-time)**
    - Take the **type-checked** code with type classes
    - Generate code with no type classes and no bounded polymorphism

    vs. C++ templates ? Template instantiation may produce ill-typed code
- **Intentional type analysis (run-time)**

  Choose the appropriate overloading operation at run-time

  See http://okmij.org/ftp/Computation/typeclass.html#dict

THANKS

- Rust支持trait，这是具有一致性的有限形式的类型类
- 在Scala中，类型类是编程惯例，可以用现存语言特征比如隐式参数来实现，本身不是独立的语言特征