Utrecht University

Concepts of Programming Language Design

Overloading (via Type Classes)

Gabriele Keller
Tom Smeding

Overview

higher & first-order syntax

inference rules, induction

tools to talk about languages

abstract machines

big step and small step operational semantics

value & type environments

parametric polymorphism/ generics

control stacks

(algebraic) data types

type classes/overloading

partial application/function closures

semantic features

functional

OO

inheritance/subclassing
method overloading

static & dynamic scoping

static & dynamic typing

language concepts

explicit & implicit typing

procedural/imperative

Utrecht University

Ad hoc polymorphism vs parametric polymorphism

- Parametric polymorphism enables us to implement functions which work on any types:

```
map :: forall a. forall b. (a-> b) -> [a] -> [b]
```

- Subtyping to use operators/functions on arguments which can be coerced to the correct type:

$$1_{\mathtt{Int}} +_{\mathtt{Float}} 2_{\mathtt{Int}}$$

- Overloading to use operators/functions on arguments on different types:

```
show 1
show [1]
“as” == “bs”
1 == 2
```

Ad hoc polymorphism vs parametric polymorphism

- Adhoc polymorphism/overloading enables us to implement functions which work on a set of types
  - `+` operator in C#
    - works on integral and floating point numeric types, strings
  - `+` operator in Haskell, works on all types which are in type class `Num`

Utrecht University

Method overloading in C#

- Methods in C# can be overloaded with implementations of different parameter type (different result type not sufficient!)

- Method overloading is resolved at compile time using the type information

```
public int Add(int a, int b)
    {
        int sum = a + b;
        return sum;
    }

public float Add(float a, float b)
    {
        float sum = a + b;
        return sum;
    }


public int Add(int a, int b, int c)
    {
        int sum = a + b + c;
        return sum;
    }
```

- We already looked at how to resolve method overloading for class methods
  - search through the class table to find appropriate definition

Utrecht University

Interaction between overloading and generics in C#

- What happens if overloading overlaps due to generics?

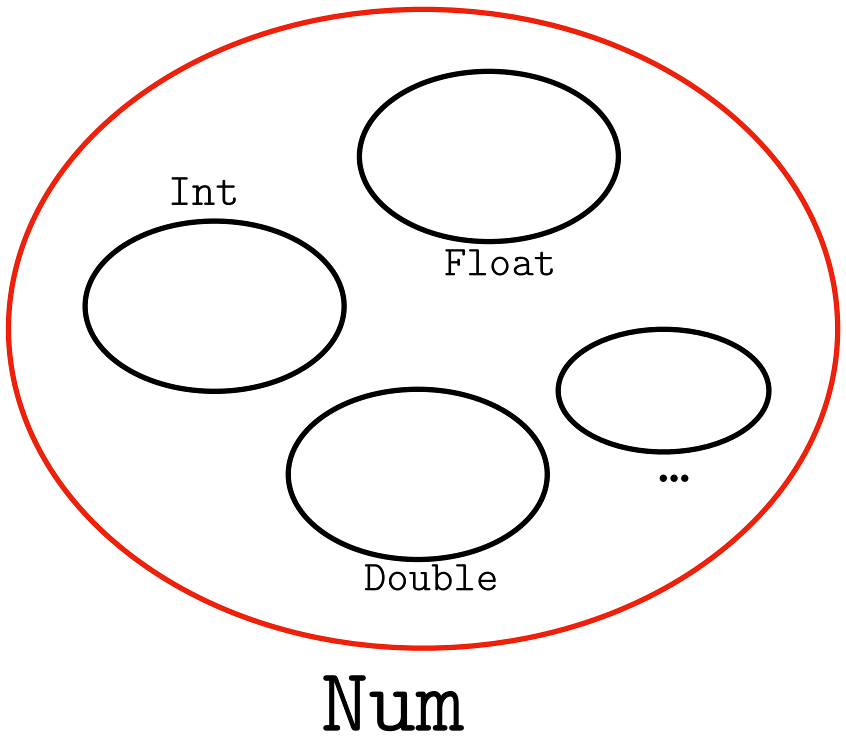
```
public static bool Check<T>(T t){
  return true;
}

public static bool Check(U t)
{
  return false;
}

public static bool Wrapper<T>(T t) {
  return Check(t);
}


Check (new U());
Wrapper <U>(new U());
```

Type Classes and Overloading

- Core Idea behind overloading via **type classes**:
  - group together types sharing a set of operations in a class of types
    - a class for arithmetic operations
    - a class for comparing values for equality
    - a class of types convertible to string representation
  - the operations defined by a type class are called class methods
  - related to the idea of abstract base classes/protocol oriented programming
  - first used in the language definition of Haskell
  - ConceptsC++
  - partially in Rust, Scala

Haskell: Overloading via Typeclasses

- Type classes are sets
  - type classes are sets of types
  - types are sets of values



- Example:
  - the type class `Num`
    - the type `Int`, `Float`, `Double` (and other numeric types) are in the type class `Num`
    - the class methods of `Num` are all arithmetic operations
  - the type class `Eq`
    - the types that can be compared for equality are in the type class of `Eq`
    - the class methods of `Eq` are `==` and `/=`

Utrecht University

Notation

- `Num` t means that the type t is in the type class `Num`
- For example
  - `Num Float`
  - `Num Int`
  - `Eq (Int, Float)`
- A signature f `::` $\forall$ t. `Num` t `=>` τ means
  - f has type τ under the condition that t is a member of type class `Num`
- For example
  - `(+) :: ∀ a.Num a => a → a → a`
  - `(==):: ∀ a.Eq a  => a → a → Bool`
  - `1    :: ∀ a.Num a => a`
- Schematic Types
  - Is `((1::Int) + (1.0::Float))` a type correct expression?
  - No, type scheme of `(+)` requires both arguments to be of the same type

Utrecht University

Typing Rules for Type Classes

- Polymorphic MinHs with type classes

Predicates	π	::=	$ClassName\ \tau$
Polytypes	σ	::=	$\tau \mid \forall\, Ident.\ \sigma \mid \pi \Rightarrow \sigma$
Monotypes	τ	::=	$\texttt{Bool} \mid \texttt{Int} \mid \tau \rightarrow \tau_2$

Overloading Resolved

- How does object-based overloading get resolved dynamically?
  - objects encode which method to apply
  - e.g., Java/C#/C++ objects have pointer to a vtable (virtual method table, dispatch table), which contains all the methods of the class
  - would not work with MinHs, as of an overloaded function need not be an object

Overloading Resolved

- Still, we can draw an inspiration
  - key idea:
    - type checker not only checks, but adjusts the code:
    - passes a table with methods of the type class as an extra argument to an overloaded function (we call such a table a dictionary)
    - overloaded function picks the appropriate function instance from the dictionary
  - dictionary (simplified) for `Eq Int` is a pair of the functions (`==`) on `Int` and (`/=`) on `Int`
    - `Eq Int` is $(==_{\texttt{Int}},\ /=_{\texttt{Int}})$
  - overloaded function as projection
    - function `(==)` projects the first component of the pair representing the `Eq` dictionary
    - function `(/=)` projects the second component of the pair representing the `Eq` dictionary

Overloading Resolved

$$\texttt{(==) 'a' 'b'} \leadsto \texttt{selectEqual ((==)}_{\texttt{Char}}\texttt{, (/=)}_{\texttt{Char}}\texttt{) 'a' 'b'}$$

```
selectEqual = fst
```

```
foo :: Eq p => Num p => p -> p -> p
foo a b =
  if (a == b)
    then a + 1
    else b
```

```
foo :: EqDict p -> NumDict p -> p -> p -> p
foo eqDict numDict a b =
  if (selectEqual eqDict a b)
    then (selectAdd numDict a 1)
    else b
```

Overloading Resolution

- Dictionary translation: the code generation of the type checker to resolve overloading is called dictionary translation
  - changes the type of the overloaded function
    - removes type classes, replaces by dictionary
  - adds code (dictionary passing and projection)
  - it's a type preserving translation
    - type of the full expression does not change
- Types after the dictionary translation: type of `(==)` after the dictionary translation:
  - `EqDict a → a → a → Bool`
  - where `EqDict a` stands for the type of the dictionary of `Eq a`
  - instead of source type: `Eq a => a → a → Bool`