Type Classes as Objects and Implicits

Based on the paper by
Bruno C. d. S. Oliveira, Adriaan Moors, Martin Odersky

OOPSLA 2010

slides by Łukasz Karniewski

16.05.2011

Abstract

- Lightweight approach to type classes in OO languages with generics using the CONCEPT pattern
- The implicit parameter passing mechanism as the missing link for convenient type class programming in OO languages
- Scala's type system to be ideally suited for generic programming in the large

Introduction

What are type classes?

What are type classes?

- Introduced in Haskell
- In short: additional interface for a type
- Good for:
    - retroactive extension
    - concept-based C++ generic programming
    - type-level computation

Roles of type classes

- Defining concepts

```
def sort[T] (xs : List[T]) (ordT : Ord[T]) : List[T] = ...

trait Ord[T] {
	def compare(a : T, b : T) : Boolean
}

object intOrd extends Ord[Int] {
	def compare(a : Int, b : Int) : Boolean = a <= b
}
```

```
scala> sort (List(3, 2, 1)) (intOrd)
List(1, 2, 3)
```

- Automatic constraints propagation

```
def sort[T] (xs : List[T]) (implicit ordT : Ord[T]) : List[T] = ...

implicit object intOrd extends Ord[Int] { … }
```

```
scala> sort (List(3, 2, 1))
List(1, 2, 3)
```

Technicalities

- Examples in Haskell
  - associated types: *-XTypeFamilies* compiler flag
- Examples in Scala
  - version 2.8.0
  - advanced examples: *-Xexperimental* compiler flag

Language mechanisms

Haskell type classes and Scala implicits

Type classes in Haskell

- Classes represent interfaces (concepts)
- Class declaration:
  - class name
  - type parameter
  - methods declarations
- Method kinds:
  - consumer methods
  - *n*-ary methods
  - factory methods
- Multiple-parameters type classes

```
class Ord a where
        (<=) :: a → a → Boolean

class Show a where
        show :: a → String

class Read a where
        read :: String → a

class Coerce a b where
        coerce :: a → b
```

```
sort :: Ord a => [a] → [a]
sort xs = ...
```

Type classes in Haskell

- Instances represent implementations (models)
- Anonymous – one instance per type
- Overlapping instances

```
instance (Ord a, Ord b) => Ord (a, b) where
        (xa, xb) <= (ya, yb) = xa < ya || (xa == ya && xb <= yb)

instance (Ord a, Ord b) => Ord (a, b) where
        (xa, xb) <= (ya, yb) = xa <= ya && xb <= yb
```

```
instance Ord a => Ord [a] where …

instance Ord [Int] where ...
```

Scala implicits

- The **implicit** keyword
- Omitting final arguments
- The implicit scope – accessible without prefix

```
implicit val out = System.out

def log (msg : String) (implicit o : PrintStream) = o.println(msg)

log(”Does not compute!”)
log(”Does not compute!!”) (System.err)
```

```
def logPrefix (msg : String) (implicit o : PrintStream, prefix : String) = log(”[” + prefix + ”] ” + msg)

def ?[T] (implicit w : T) : T = w

logPrefix(”Message”) (?, ”prefix”)
```

The implicit scope

- The compiler searches for:
  - Definitions introduced with **implicit val**, **implicit object**, **implicit def**
  - Implicit arguments in local scope
- Not found – use default value (if given)
- Ambiguity – choose *most specific* one
  - A is more specific than B, if B is defined in superclass or in companion object of class defining A

The implicit scope

```
trait Monoid[A] {
    def binary_op(x : A, y : A) : A
    def identity                : A
}

def acc[A] (l : List[A]) (implicit m : Monoid[A]) : A =
    l.foldLeft (m.identity) ((x, y) => m.binary_op(x, y))

object A {
    implicit object sumMonoid extends Monoid[Int] {
        def binary_op(x : Int, y : Int) = x + y
        def identity                    = 0
    }
    def sum (l : List[Int]) : Int = acc(l)
}

object B {
    implicit object prodMonoid extends Monoid[Int] {
        def binary_op(x : Int, y : Int) = x * y
        def identity                    = 1
    }
    def product (l : List[Int]) : Int = acc(l)
}
```

```
val test : (Int, Int, Int) = {
    import A._
    import B._

    val l = List (1, 2, 3, 4, 5)

    (sum(l), product(l), acc(l) (prodMonoid))
}
```

The missing link

- Type-driven selection mechanism
- Convenience
- Manual ambiguity resolving
- Custom implicit conversions

The missing link

```
trait Ord[T] {
	def compare(x : T, y : T) : Boolean
}

implicit def ordPair[A, B] (implicit ordA : Ord[A], ordB : Ord[B]) = new Ord[(A, B)] { … }

def cmp[A] (x : A, y : A) (implicit ord : Ord[A]) : Boolean = ord.compare(x, y)
```

```
def cmp[A : Ord] (x : A, y : A) : Boolean = ?[Ord[A]].compare(x, y)

cmp(x, y)
```

```
trait Ordered[T] {
	def comp(o : T) : Boolean
}

implicit def mkOrdered[T : Ord] (x : T) : Ordered[T] = new Ordered[T] {
	def comp(o : T) = ?[Ord[T]].compare(x, o)
}

x.comp(y)
x comp y
```

The CONCEPT Pattern

Benefits of type classes in OO programs

The CONCEPT Pattern

- Type-class-style interfaces in OO with generics
- CONCEPT:
    - concept interface ↔ type class
    - modeled type ↔ type parameter
    - conceptual methods ↔ type class methods
    - model ↔ class instance
- Consumer, factory and *n*-ary methods
- Multi-type concepts

The CONCEPT Pattern

```
trait Ord[T] {
	def compare(x : T, y : T) : Boolean
}

class Apple (x : Int) {}

object ordApple extends Ord[Apple] {
	def compare (a : Apple, b : Apple) =
		a.x <= b.x
}

def pick[T] (a : T, a : T) (ord : Ord[T]) =
	if (ord.compare(a, b)) a else b
```

```
val a = new Apple(3)
val b = new Apple(5)
val c = pick(a, b) (ordApple)
```

```
object ordApple2 extends Ord[Apple] {
	def compare (a : Apple, b : Apple) =
		a.x > b.x
}

val d = pick(a, b) (ordApple2)
```

```
implicit object ordApple extends Ord[Apple]…

def cmp[T] (a : T, b : T) (implicit ord : Ord[T]) =
	ord.compare(a, b)

def pick[T : Ord] (a : T, b : T) =
	if (cmp(a, b)) a else b
```

```
val c = pick(a, b)
```

```
trait Show[T] {
	def show (x : T) : String
}

trait Read[T] {
	def read (x : String) : T
}

trait Coerce[A, B] {
	def coerce (x : A) : B
}
```

Benefits and limitations

- Benefits:
  - retroactive modeling
  - multiple method implementations
  - type-safe statically dispatched *n*-ary methods
  - factory methods
- Limitations: static dispatch
- Alternative: interfaces

```
trait Ord[T] {
	def compare(x : T) : Boolean
}

class Apple (x : Int) extends Ord[Apple] { … }
```

Language support

- Java, C#:
  - Explicit model passing
  - Syntactic noise
  - C#: retroactive implementations via *extension methods*
- Haskell, JavaGI, C++0x:
  - Direct language support for concept-style interfaces
- Scala approach:
  - The CONCEPT pattern and implicits

Examples

Haskell type classes vs. CONCEPT pattern in Scala

Ordering concept

```
trait Eq[T] {
        def equal (a : T, b : T) : Boolean
}

trait Ord[T] extends Eq[T] {
        def compare(a : T, b : T) : Boolean
        def equal (a : T, b : T) : Boolean =
                compare(a, b) && compare(b, a)
}

class IntOrd extends Ord[Int] {
        def compare (a : Int, b : Int) = a <= b
}
```

- Default definitions
- Retroactive modeling
- Multiple implementations

```
class ListOrd[T] (ord : Ord[T])
extends Ord[List[T]] {
   def compare (l1 : List[T], l2 : List[T]) =
       (l1, l2) match {
          case (x::xs, y::ys) =>
             if (ord.compare(x, y)) compare(xs, ys)
             else ord.compare(x, y)
          case (_, Nil) => false
          case (Nil, _) => true
       }
}

class ListOrd2[T] (ord : Ord[T])
extends Ord[List[T]] {
   private val listOrd = new ListOrd[T] (ord)
   def compare (l1 : List[T], l2 : List[T]) =
      l1.length < l2.length && listOrd.compare(l1, l2)
}
```

Ordering concept

```
def sort[T] (xs : List[T]) (ord : Ord[T]) : List[T] = …

val l1 = List(7, 2, 6, 4, 5, 9)
val l1 = List(2, 3)

val test1 = new ListOrd(new IntOrd()).compare(l1, l2)

val test2 = new ListOrd2(new IntOrd()).compare(l1, l2)

val test3 = sort (l1) (new IntOrd())
```

```
class Ord a where
        compare :: a → a → Boolean

sort :: Ord a => [a] → [a]
sort xs = ...

l1 = [7, 2, 6, 4, 5, 9]
l2 = [2, 3]

test1 = compare l1 l2
test3 = sort l1
```

```
implicit val intOrd = new Ord[Int] { … }

implicit def listOrd[T] (implicit ord : Ord[T]) = new Ord[List[T]] { … }

def listOrd2[T] (implicit ord : Ord[T]) = new Ord[List[T]] { … }

def cmp[T] (x : T, y : T) (implicit ord : Ord[T]) = ord.compare(x, y)
```

```
val test1 = cmp (l1, l2)
val test2 = cmp (l1, l2) (listOrd2)
val test3 = sort (l1)
```

Set concept

```
trait Set[S] {
        val empty : S
        def insert (s : S, x : Int) : S
        def contains (s : S, x : Int) : Boolean
        def union (s : S, z : S) : S
}

class ListSet extends Set[List[Int]] {
        val empty = List ()
        def insert (s : List[Int], x : Int) = x :: s
        def contains (s : List[Int], x : Int) = s.contaions(x)
        def union (s : List[Int], z : List[Int]) = x.union(y)
}

class FunctionalSet extends Set[Int => Boolean] {
        val empty = (x : Int) => false
        def insert (f : Int => Boolean, x : Int) =
                z => x.equals(z) || f(z)
        def contains (f : Int => Boolean, x : Int) = f(x)
        def union (f : Int => Boolean, g : Int => Boolean) =
                x => f(x) || g(x)
}
```

```
def test[S] (s : Set[S]) : Boolean =
        s.contains (s.insert (s.empty, 42), 42)
```

```
class Set s where
        empty     :: s
        insert    :: s → Int → s
        contains  :: s → Int → s
        union     :: s → s → s

instance Set [Int] where
        empty     = []
        insert    = \s x → x : s
        contains  = \s x → elem x s
        union     = \s z → List.union s z

instance Set (Int → Bool) where
        empty     = \x → False
        insert    = \f x z → x == z || f z
        contains  = \f x → f x
        union     = \f g x → f x || g x
```

```
test :: Set s => Bool
test = contains (insert empty 42) 42
```

Statically-typed *printf*

```
trait Format[A] {
  def format (s : String) : A
}

def printf[A] (format : Format[A]) : A = format.format("")

class I[A] (formatD : Format[A])
extends Format[Int => A] {
        def format(s : String) =
                i => formatD.format(s + i.toString)
}

class C[A] (formatD : Format[A])
extends Format[Char => A] {
        def format(s : String) =
                c => formatD.format(s + c.toString)
}

class E extends Format[String] {
  def format(s : String) = s
}

class S[A](l : String, formatD : Format[A])
extends Format[A] {
        def format(s : String) = formatD.format(s + l)
}
```

```
val fmt:Format[Int => Char => String] =
  new S("Int: ", new I(new S(", Char: ",
new C(new S(".", new E)))))

def test = printf (fmt) (3) ('c')
```

```
test :: String
test = printf (3 :: Int) 'c'
```

- More flexible
- Less compact
- Possible to infer formats with implicits

Type class programs are OO programs

```
data Ord a = Ord {
        eq          :: a → a → Bool,
        compare  :: a → a → Bool
}

intOrd :: Ord Int
intOrd = Ord {
        eq          = \a b → compare intOrd a b &&
                                    compare intOrd b a,

        compare  = \a b → a <= b
}

listOrd :: Ord a → Ord [a]
listOrd ord = Ord {
        eq          = \a b → compare (listOrd ord) a b &&
                                    compare (listOrd ord) b a,

        compare  = \l1 l2 → case (l1, l2) of
                (x:xs, y:ys) →
                        if (eq ord x y)
                                then compare (listOrd ord) xs ys
                                else compare ord x y
                (_, []) → False
                (_, _) → True
}
```

- Conversion:
  - type classes → records
  - instances → values
- Named instances and arguments
- Explicit instances passing
- Explicit recursive calls

Advanced uses of type classes

Associated types and advanced Scala features

Associated types and type members

- Haskell – associated types
  - associated to a type class
  - concrete in the instance
- Scala – type members
  - selected on a type
  - p.T ↔ p.**type**#T – path-dependent type
  - dependent method types

```
class Collects c where
        type Elem c
        empty :: c
        insert :: c → Elem c → c
        toList :: c → [Elem c]

instance Collects BitSet where
        type Elem BitSet = Char
        …
```

```
def identity (x : AnyRef) : x.type = x

def ?[T <: AnyRef] (implicit w : T) : w.type = w
```

Session types

```
sealed case class Stop ()
sealed case class In[-A, +B] (recv : A => B)
sealed case class Out[+A, +B] (data : A, cont : B)

trait Session[S] {
        type Self = S
        type Dual
        type DualOf[D] = Session[Self] { type Dual = D }
        def run (self : Self, dual : Dual) : Unit
}

implicit object StopDual extends Session[Stop] {
        type Dual = Stop
        def run(self : Self, dual : Dual) : Unit = {}
}

implicit def InDual[D, C] (implicit cont : Session[C]) =
        new Session[In[D, C]] {
                type Dual = Out[D, cont.Dual]
                def run(self : Self, dual : Dual) : Unit =
                        cont.run(self.recv(dual.data), dual.cont)
        }

implicit def OutDual[D, C] (implicit cont : Session[C]) =
        new Session[Out[D, C]] {
                type Dual = In[D, cont.Dual]
                def run(self : Self, dual : Dual) : Unit =
                        cont.run(self.cont, dual.recv(self.data))
        }
```

```
def add_server =
        In { x : Int =>
        In { y : Int => System.out.println("Thinking")
        Out (x + y, Stop())
        }}

def add_client =
        Out (3,
        Out (4, { System.out.println("Waiting")
        In { z : Int => System.out.println(z); Stop() }
        }))
```

```
def runSession[S, D : Session[S]#DualOf]
(session : S, dual : D) =
        ?[Session[S]#DualOf[D]].run(session, dual)

def myRun = runSession(add_server, add_client)
```

```
InDual:
   [D, C] (implicit cont : Session[C]) Session[In[D, C]]
      { type Dual = Out[D, cont.Dual] }
```

```
scala> myRun
Waiting
Thinking
7
```

Arity-polymorphic *zipWith*

```
case class Zero
case class Succ[N] (x : N)

trait ZipWith[N, S] {
   type ZipWithType
   def manyApp : N => Stream[S] => ZipWithType
   def zipWith : N => S => ZipWithType =
       n => f => manyApp (n) (repeat (f))
}

def zipWith[N, S] (n : N, s : S)
   (implicit zw : ZipWith[N , S]) : zw.ZipWithType =
       zw.zipWith (n) (s)

implicit def ZeroZW[S] = new ZipWith[Zero, S] {
   type ZipWithType = Stream[S]
   def manyApp = n => xs => xs
}

implicit def SuccZW[N, S, R] (implicit zw : ZipWith[N, R]) =
   new ZipWith[Succ[N], S => R] {
       type ZipWithType = Stream[S] => zw.ZipWithType
       def manyApp = n => xs => ss => n match {
           case Succ (i) => zw.manyApp (i) (zapp (xs, ss))
       }
   }
```

```
def zipWith0 : Stream[Int] = zipWith(Zero(), 0)

def map[A, B] (f : A => B) : Stream[A] => Stream[B] =
   zipWith(Succ (Zero ()), f)

def zipWIth3[A, B, C, D] (f : A => B => C => D)
   : Stream[A] => Stream[B] => Stream[C] => Stream[D] =
       zipWith(Succ ( Succ ( Succ (Zero ()))), f)
```

```
zipWith :: (a → b → c) → [a] → [b] → [c]

zipWIthN :: (a1 → … → aN) → [a1] → … → [aN]
```

- Type-level computation
- *ZipWithType* – return type computed from argument type

Arity-polymorphic *zipWith* – prioritised implicits

```
trait ZipWith[S] {
   type ZipWithType
   def manyApp : Stream[S] => ZipWithType
   def zipWith : S => ZipWithType =
      f => manyApp (repeat (f))
}

class ZipWithDefault {
   implicit def ZeroZW[S] = new ZipWith[S] {
        type ZipWithType = Stream[S]
        def manyApp = xs => xs
   }
}

object ZipWith extends ZipWithDefault {
   def apply[S] (s : S) (implicit zw : ZipWith[S])
     : zw.ZipWithType = zw.zipWith (s)

   implicit def SuccZW[S, R] (implicit zw : ZipWith[R]) =
      new ZipWith[S => R] {
         type ZipWithType = Stream[S] => zw.ZipWithType
         def manyApp =
            xs => ss => zw.manyApp (zapp (xs, ss))
      }
}
```

```
import ZipWith._

def zipWith0 : Stream[Int] = ZipWith(0)

def map[A, B] (f : A => B) : Stream[A] => Stream[B] =
   ZipWith(f)
```

```
def repeat[A] (x : A) : Stream[A] = cons (x, repeat(x))

def zapp[A, B] (xs : Stream[A => B], ys : Stream[A])
 : Stream[B] =
    (xs, ys) match {
       case (cons(f, fs), cons(s, ss)) =>
          cons (f (s), zapp(fs, ss))
       case (_, _) => Stream.empty
    }
```

- Overlapping implicits
- *SuccZW* preferred over *ZeroZW*
- Not applicable to Haskell

Generalized type constraints

- A mechanism for methods to constrain the type parameters of the class
- Equal (=:=), subtype (<:<), convertible (<%<)
- Implicit search for *evidence* that the relation holds

```
sealed abstract class =:=[F, T] extends (F => T)

implicit def typeEq[A] : A =:= A = new (A =:= A) {
        def apply(x : A) = x
}

case class Foo[A] (a : A) {
        def strLen (implicit evidence : A =:= String) =
                a.length
}
```

```
sealed abstract class <:<[-F, +T] extends (F => T)

implicit def conforms[A] : A <:< A = new (A <:< A) {
        def apply(x : A) = x
}

trait Traversable[T] {
        type Coll[X]
        def flatten[U] (implicit w : T <:< Traversable[U]) : Coll[U]
}
```

```
scala> Foo("bar").strLen
res1: Int = 3

scala> Foo(123).strLen
<console>:9: error: could not find implicit value for parameter evidence: =:=[Int,String]
```

The common pattern

- Relations on types:
  - type constructor of the same arity
  - implicits with corresponding type
- Lightweight type-level computation
- Real-world application: Scala 2.8 collections library
- Examples:
  - Session types: *DualOf*
  - *n*-ary *zipWith*: argument and return type
  - Generalized constraint relations:
    =:=, <:<, <%<

Real-world applications

Support for generic programming in different languages

Real-world example

- Scala 2.8 collections library (Odersky and Moors 2009)
- Crucial role in design:
  - implicits
  - type members
  - dependent method types
- The *relations-on-types* pattern:
  - How transformations on collections affect the types of the elements and their containers
- ”STL/Boost of Scala”

JavaGI

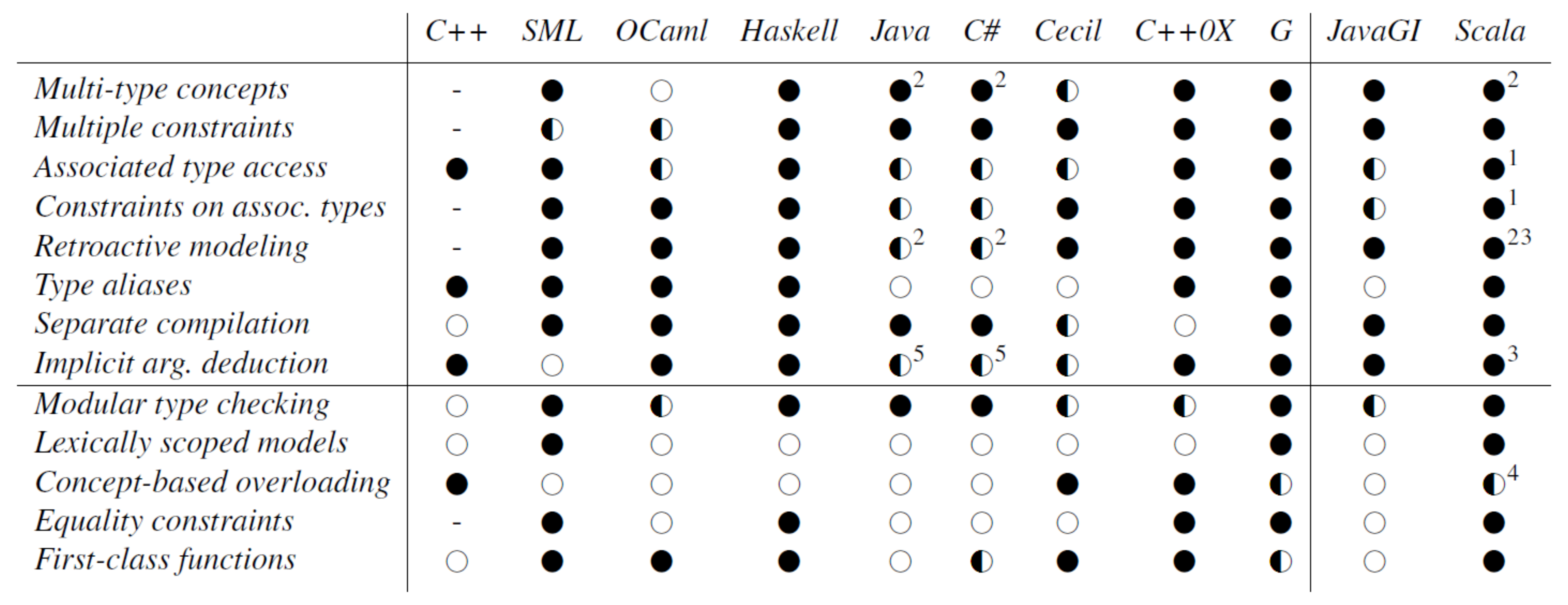
- Generalized Interfaces for Java
- Extension of Java: generalized interfaces and implementations
- Dynamic multiple dispatch
- Retroactive extensions

```
interface Ord {
        int compareTo (This that);
}

implementation Ord [Number] {
        int compareTo (Number that) { … }
}

<T> T max (T a, T b) where T implements Ord {
        if (a.compareTo(b) > 0) return a;
        else return b;
}
```

C++ concepts vs. type classes and the CONCEPT pattern

- C++ concepts:
  - Used to document generic algorithms
  - No representation within the C++ language
- Different purposes and motivations:
  - C++ – performance
  - Type classes and implicits – convenience and abstraction
- The CONCEPT pattern:
  - Expressing concepts with standard OO class systems without the performance constraints of C++

Level of support for generic programming in several languages



	C++	SML	OCaml	Haskell	Java	C#	Cecil	C++0X	G	JavaGI	Scala
Multi-type concepts	-	●	○	●	●[2]	●[2]	◐	●	●	●	●[2]
Multiple constraints	-	◐	◐	●	●	●	●	●	●	●	●
Associated type access	●	●	◐	●	◐	◐	◐	●	●	◐	●[1]
Constraints on assoc. types	-	●	●	●	◐	◐	●	●	●	◐	●[1]
Retroactive modeling	-	●	●	●	◐[2]	◐[2]	●	●	●	●	●[2,3]
Type aliases	●	●	●	●	○	○	○	●	●	○	●
Separate compilation	○	●	●	●	●	●	◐	○	●	●	●
Implicit arg. deduction	●	○	●	●	◐[5]	◐[5]	◐	●	●	●	●[3]
Modular type checking	○	●	◐	●	●	●	◐	◐	●	◐	●
Lexically scoped models	○	●	○	○	○	○	○	○	●	○	●
Concept-based overloading	●	○	○	○	○	○	●	●	◐	○	◐[4]
Equality constraints	-	●	○	●	○	○	○	●	●	○	●
First-class functions	○	●	●	●	○	◐	●	●	◐	○	●

● – good, ◐ – sufficient, ○ – poor

1) Supported via type members and dependent method types
2) Supported via the CONCEPT pattern
3) Supported via implicits
4) Partially supported by prioritized overlapping implicits
5) Decreased score due to the use of the CONCEPT pattern

Summary

General conclusions

Summary

- The CONCEPT pattern
  - Benefits of type classes in a standard OO language with generics
  - Slight convenience loss
- Implicit parameter passing
  - Convenience of use of type classes
  - Wider applicability and usefulness in other domains
- Scala – excellent support for generic programming in the large

Literature

Type Classes as Objects and Implicits
Bruno C. d. S. Oliveira (*Seoul National University*),
Adriaan Moors, Martin Odersky (*Ecole Polytechnique Fédérale de Lausanne*),

OOPSLA 2010