5.– 8. September 2011
in Nürnberg

Herbstcampus

Wissenstransfer
par excellence

Schwarze Magie

Scalas Typsystem ausgenutzt

Lars Hupel

Schwarze Magie

Scalas Typsytem ausgenutzt
Herbstcampus 2011 – S32

Lars Hupel
hupel@in.tum.de

8. September 2011

Über den Autor

- Informatik-Student, TU München
- Java-Programmierer seit 5 Jahren
- Scala-Programmierer seit 2 Jahren
- Interesse an formalen Methoden, Typsystemen, ...

Werbung

Scala Enthusiasten Metropolregion Nürnberg

Scalas Typsystem

- Grundlage bleibt objektorientiert
- erweiterte Typinferenz
- Singletons
- Existential types
- Definition-site variance
- Implizite Konversionen und Parameter
- ...

Innere Klassen

Java vs. Scala

Java

```
class Outer {
    class Inner {
    }
}

Outer o = new Outer();
Outer.Inner i = o.new Inner();
```

Innere Klassen

Java vs. Scala

Java

- nicht-statische innere Klassen haben Referenz auf äußere Klasse
- $\rightsquigarrow$ Instanziierung benötigt äußeres Objekt
- aber: alle inneren Objekte haben selben Typ

Innere Klassen

Java vs. Scala

Java

- nicht-statische innere Klassen haben Referenz auf äußere Klasse
- $\rightsquigarrow$ Instanziierung benötigt äußeres Objekt
- aber: alle inneren Objekte haben selben Typ

Innere Klassen

Java vs. Scala

Scala

```
class Outer {
    class Inner
}

val outer = new Outer()
val i1: outer.Inner = new outer.Inner()
val i2: Outer#Inner = i1
```

Innere Klassen

Java vs. Scala

Scala

- Referenz auf äußeres Objekt wird Teil des Typs
- $\rightsquigarrow$ innere Objekte zweier verschiedener äußerer Objekte nicht miteinander kompatibel
- $\rightsquigarrow$ Probleme bei Java-Interoperabilität
- Lösung: Projektion `Outer#Inner`

Bereichsdatentyp

- eine Objekt ist mit einem Zahlenbereich assoziiert
- Methoden dieses Objekts sollen nun nur auf Zahlen aus diesem Bereich operieren
- Zahlen außerhalb des Bereichs sollen zu Compiler-Fehler führen

Bereichsdatentyp

- eine Objekt ist mit einem Zahlenbereich assoziiert
- Methoden dieses Objekts sollen nun nur auf Zahlen aus diesem Bereich operieren
- Zahlen außerhalb des Bereichs sollen zu Compiler-Fehler führen

Bereichsdatentyp

Implementation

```
class Range(begin: Int, end: Int) {
    class RangeInt private[Range](val n: Int) {
        def next: Option[Range.this.RangeInt] =
        if (n < end) Some(new RangeInt(n+1))
        else         None
    }

    def first = new RangeInt(begin)
    def last = new RangeInt(end)
}

class HasRange {
    val myRange: Range = new Range(1, 10)
    def func(i: myRange.RangeInt) = println(i.n)
}
```

Bereichsdatentyp

Verwendung

```
> new HasRange()
res0: HasRange = HasRange@5b927504
> res0.func(res0.myRange.first)
1

> new Range(11, 20)
res1: Range = Range@790f2f3c
> res0.func(res1.first)
error: type mismatch;
 found    : res1.RangeInt
 required: res0.myRange.RangeInt
```

Singletons

Scala ordnet einigen Werten einen eindeutigen Typen zu, der aber nicht immer inferiert wird

```
> class Foo
> val foo = new Foo {}
foo: Foo = Foo@608a6351

> checkType(foo)
Foo

> checkType[foo.type](foo)
Foo@608a6351.type
```

Singletons und Implicits

apply/update

```
class Sugar {
    def apply(i: Int) = i
    def update(i: Int, value: Int) = ()
}

val sugar = new Sugar()

sugar(3) ⇝ sugar.apply(3)
sugar(3) = 5 ⇝ sugar.update(3, 5)
```

Singletons und Implicits

Symbol

```
'a ⇝ Symbol.apply("a")
'a = 0 ⇝ Symbol.update("a", 0)
```

Singletons und Implicits

Symbol

```
'a ⇝ Symbol.apply("a")
'a = 0 ⇝ Symbol.update("a", 0)
```

aber: `Symbol` hat keine `update`-Methode

Singletons und Implicits

Symbol

```
implicit def pimpSymbol(s: Symbol.type) = new {
    def update(str: String, newVal: Any) =
        println(str + " := " + newVal)
}

> 'a = "foo"
a := foo
```

Collections in 2.8

Scalas Collections wurden mit 2.8 komplett überarbeitet:
Transformationen liefern immer den bestmöglichen Typ

möglich durch `CanBuildFrom`

Collections in 2.8

Scalas Collections wurden mit 2.8 komplett überarbeitet:
Transformationen liefern immer den bestmöglichen Typ

möglich durch `CanBuildFrom`

CanBuildFrom

```
trait CanBuildFrom[-From, -Elem, +To] {

    def apply(from: From): Builder[Elem, To]

    def apply(): Builder[Elem, To]

}
```

CanBuildFrom

`CanBuildFrom` ermöglicht es, über Collections zu abstrahieren

`CanBuildFrom[From, Elem, To]` bedeutet: es ist möglich...

- auf Grundlage einer Collection des Typs `From`
- mit Elementen des Zieltyps `Elem`
- eine Collection des Typs `To`

... zu erstellen

CanBuildFrom

`CanBuildFrom` ermöglicht es, über Collections zu abstrahieren

`CanBuildFrom[From, Elem, To]` bedeutet: es ist möglich...

- auf Grundlage einer Collection des Typs `From`
- mit Elementen des Zieltyps `Elem`
- eine Collection des Typs `To`

... zu erstellen

CanBuildFrom

`CanBuildFrom` ermöglicht es, über Collections zu abstrahieren

`CanBuildFrom[From, Elem, To]` bedeutet: es ist möglich...

- auf Grundlage einer Collection des Typs `From`
- mit Elementen des Zieltyps `Elem`
- eine Collection des Typs `To`

... zu erstellen

CanBuildFrom

`CanBuildFrom` ermöglicht es, über Collections zu abstrahieren

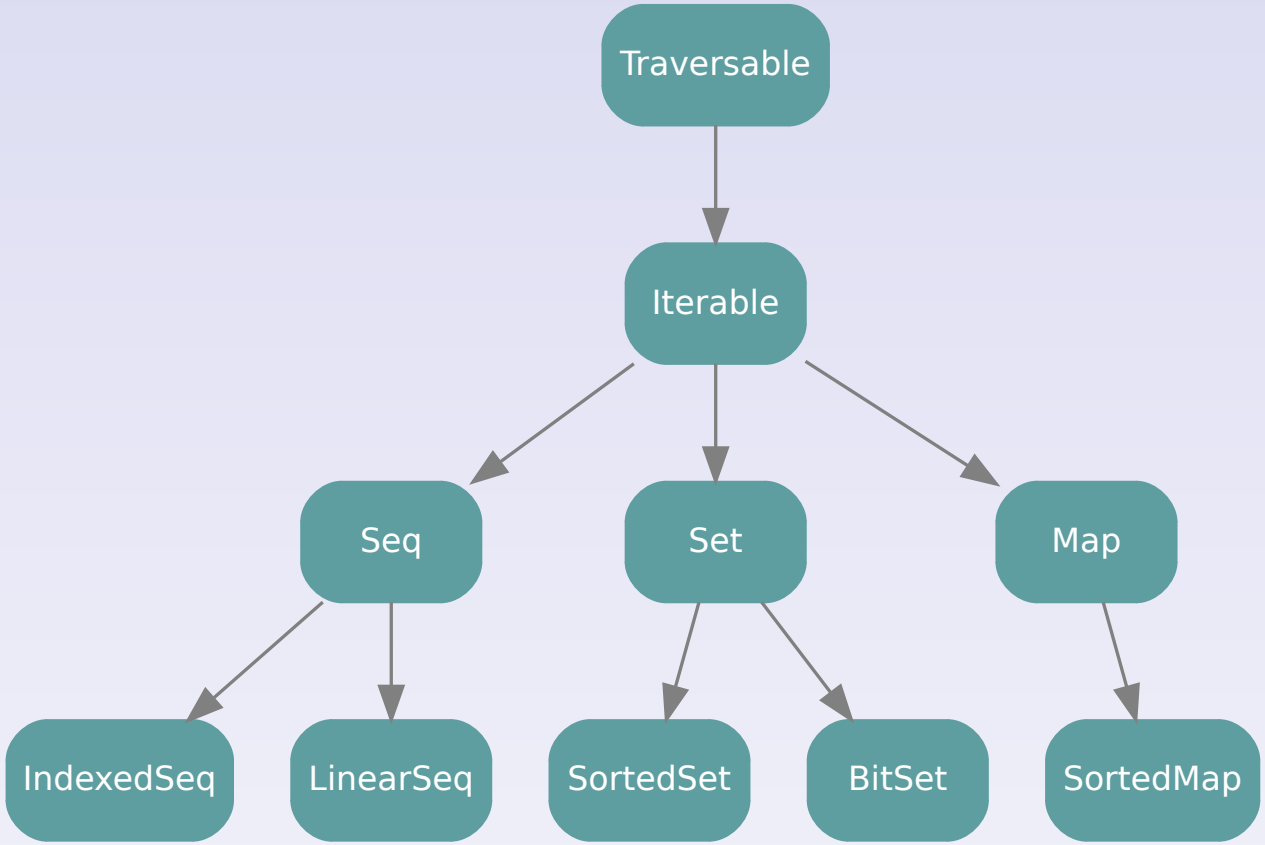
`CanBuildFrom[From, Elem, To]` bedeutet: es ist möglich...

- auf Grundlage einer Collection des Typs `From`
- mit Elementen des Zieltyps `Elem`
- eine Collection des Typs `To`

... zu erstellen

Quicksort

Schritt 1: Signatur definieren

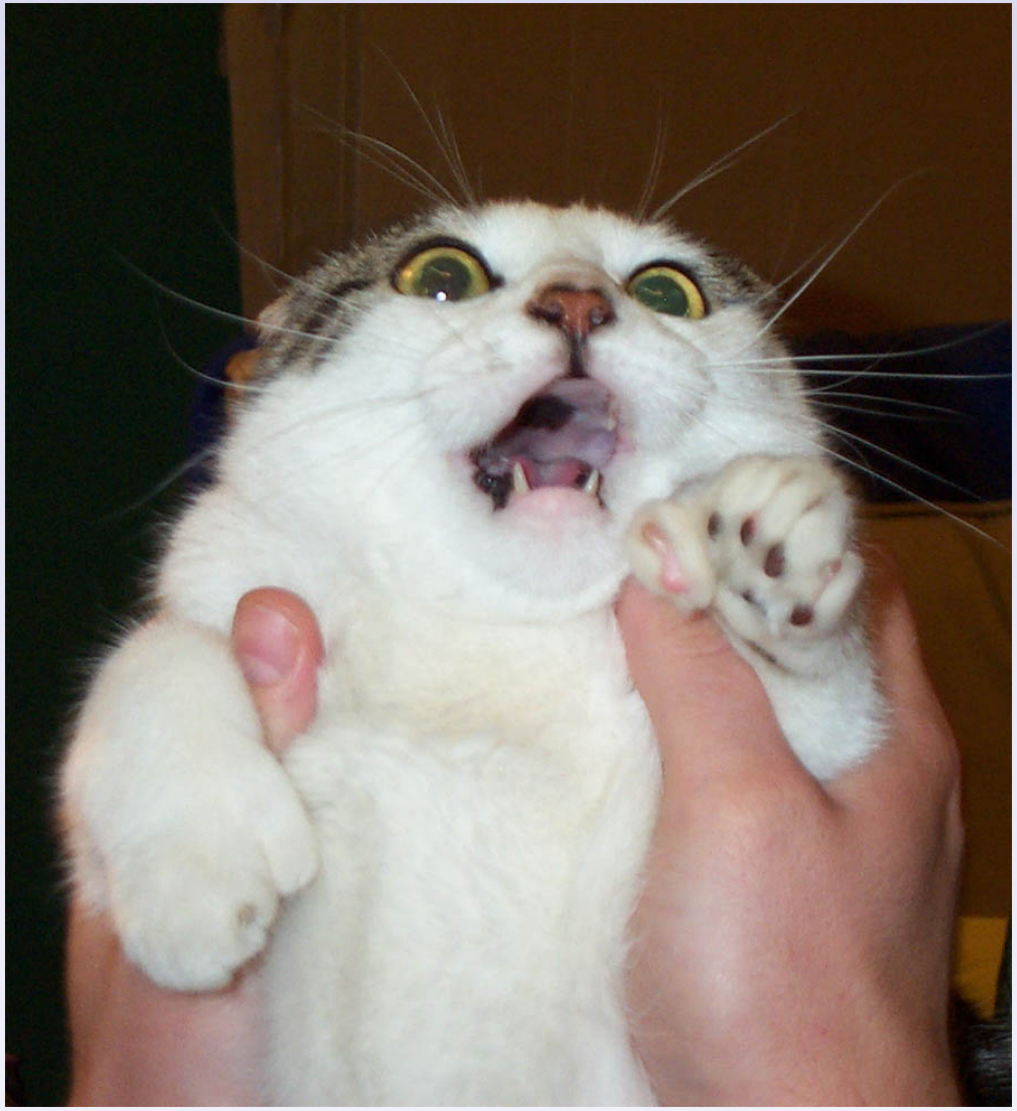


`http://www.decodified.com/scala/collections-api.xml`, Mathias, CC-by 3.0

Quicksort

Schritt 1: Signatur definieren

```
def sort[T, Coll](a: Coll)
  (implicit ev: Coll <:< SeqLike[T, Coll],
            cbf: CanBuildFrom[Coll, T, Coll],
            n: Ordering[T]): Coll
```



Quicksort

Schritt 1: Signatur definieren

```
def sort[T, Coll](a: Coll)
  (implicit ev: Coll <:< SeqLike[T, Coll],
            cbf: CanBuildFrom[Coll, T, Coll],
            n: Ordering[T]): Coll
```

- `ev: Coll` Subtyp von `SeqLike[T, Coll]`?
- `SeqLike[T, Coll]`: Transformationen auf `Coll` erhalten den Typ `Coll`, wenn ...
- ein entsprechendes `CanBuildFrom` vorhanden ist

Quicksort

Schritt 1: Signatur definieren

```
def sort[T, Coll](a: Coll)
  (implicit ev: Coll <:< SeqLike[T, Coll],
            cbf: CanBuildFrom[Coll, T, Coll],
            n: Ordering[T]): Coll
```

- `ev: Coll` Subtyp von `SeqLike[T, Coll]`?
- `SeqLike[T, Coll]`: Transformationen auf `Coll` erhalten den Typ `Coll`, wenn ...
- ein entsprechendes `CanBuildFrom` vorhanden ist

Quicksort

Schritt 1: Signatur definieren

```
def sort[T, Coll](a: Coll)
  (implicit ev: Coll <:< SeqLike[T, Coll],
            cbf: CanBuildFrom[Coll, T, Coll],
            n: Ordering[T]): Coll
```

- `ev`: `Coll` Subtyp von `SeqLike[T, Coll]`?
- `SeqLike[T, Coll]`: Transformationen auf `Coll` erhalten den Typ `Coll`, wenn ...
- ein entsprechendes `CanBuildFrom` vorhanden ist

Quicksort

Schritt 1: Signatur definieren

```
def sort[T, Coll](a: Coll)
  (implicit ev: Coll <:< SeqLike[T, Coll],
            cbf: CanBuildFrom[Coll, T, Coll],
            n: Ordering[T]): Coll
```

- `ev`: `Coll` Subtyp von `SeqLike[T, Coll]`?
- `SeqLike[T, Coll]`: Transformationen auf `Coll` erhalten den Typ `Coll`, wenn ...
- ein entsprechendes `CanBuildFrom` vorhanden ist

Quicksort

Schritt 2: Implementation

Demo

Werte und Typen

Terminologie

- Datenkonstruktoren erzeugen Daten
- *in Java: „Konstruktoren“*
- Typkonstruktoren erzeugen Typen
- *in Java: „generische Klasse“*
- auch bekannt als „parametrische Polymorphie“

Werte und Typen

Terminologie

- Datenkonstruktoren erzeugen Daten
- *in Java: „Konstruktoren“*
- Typkonstruktoren erzeugen Typen
- *in Java: „generische Klasse“*
- auch bekannt als „parametrische Polymorphie“

Werte und Typen

Terminologie

- Datenkonstruktoren erzeugen Daten
- *in Java: „Konstruktoren“*
- Typkonstruktoren erzeugen Typen
- *in Java: „generische Klasse“*
- auch bekannt als „parametrische Polymorphie“

Werte und Typen

Terminologie

- Datenkonstruktoren erzeugen Daten
- *in Java: „Konstruktoren“*
- Typkonstruktoren erzeugen Typen
- *in Java: „generische Klasse“*
- auch bekannt als „parametrische Polymorphie“

Werte und Typen

Kinds

„Kind“ bezeichnet die Struktur eines Typkonstruktors

Notation

- $*$ steht für beliebigen Typ
- -> steht für Abbildung

Werte und Typen

Kinds

„Kind“ bezeichnet die Struktur eines Typkonstruktors

Notation

- $*$ steht für beliebigen Typ
- -> steht für Abbildung

Werte und Typen

Kinds

„Kind“ bezeichnet die Struktur eines Typkonstruktors

Notation

- $*$ steht für beliebigen Typ
- -> steht für Abbildung

Typen und Kinds

Beispiele

Typ	Deklaration	Kind
Liste	`class List[A]`	`* -> *`
Paar	`class Tuple2[A, B]`	`(* x *) -> *`
Applikation	`type Ap[T[_], S] = T[S]`	`((* -> *) x *) -> *`

Methoden und Funktionen

- Methoden sind *keine* first class citizen
- Funktionen sind „gewöhnliche“ Objekte mit `apply`-Methode
- Methode $\longrightarrow$ Funktion per `f _`

aber: Was ist mit generischen Methoden?

Methoden und Funktionen

- Methoden sind *keine* first class citizen
- Funktionen sind „gewöhnliche“ Objekte mit `apply`-Methode
- Methode $\longrightarrow$ Funktion per `f _`

aber: Was ist mit generischen Methoden?

Methoden und Funktionen

Notwendigkeit von Kinds

```
def transform[A](opt: Option[A]): List[A] =
    opt.toList

> transform _
res0: Option[Nothing] => List[Nothing] = <function1>
```

Methoden und Funktionen

Notwendigkeit von Kinds

```
trait ~>[-A[_], +B[_]] {
    def apply[T](a: A[T]): B[T]
}

new (Option ~> List) {
    def apply[T](opt: Option[T]): List[T] =
        opt.toList
}
```

Methoden und Funktionen

Notwendigkeit von Kinds

```
trait ~>[-A[_], +B[_]] {
    def apply[T](a: A[T]): B[T]
}

new (Option ~> List) {
    def apply[T](opt: Option[T]): List[T] =
        opt.toList
}
```

Erkenntnis: Kinds erhöhen die Ähnlichkeit von *Methoden* und *Funktionen*

Kindinferenz

- ... ist so gut wie nicht vorhanden
- explizite Deklaration erforderlich
- Inferenz der Parameter beim Aufruf einer solchen Methode schlägt oft fehl

Typäquivalenz

Problem

generische Klasse, bei der einige Methoden nur bei speziellen Typparametern angeboten werden

Typäquivalenz

Problem

generische Klasse, bei der einige Methoden nur bei speziellen Typparametern angeboten werden

Lösung: impliziter „Beweis“

Naive Implementation

```
class =:=[From, To] extends (From => To)

implicit def tpEquals[A] = new =:=[A, A]
```

Disclaimer

- Zugriffsschutz fehlt
- Casten kann nicht verhindert werden

Naive Implementation

Verwendung

```
trait Generic[A] {

    val a: A

    def onlyForString(implicit ev: A =:= String) =
        a.charAt(0)

}
```

Naive Implementation

Nachteile

- aus `A =:= B` kann nicht `B =:= A` konstruiert werden
- aus `A =:= B` und `B =:= C` kann nicht `A =:= C` konstruiert werden
- aus `A =:= B` kann nicht `F[A] =:= F[B]` konstruiert werden
- $\rightsquigarrow$ keine vollständige Äquivalenz auf Typebene

Naive Implementation

Nachteile

- aus `A =:= B` kann nicht `B =:= A` konstruiert werden
- aus `A =:= B` und `B =:= C` kann nicht `A =:= C` konstruiert werden
- aus `A =:= B` kann nicht `F[A] =:= F[B]` konstruiert werden
- $\rightsquigarrow$ keine vollständige Äquivalenz auf Typebene

Naive Implementation

Nachteile

- aus `A =:= B` kann nicht `B =:= A` konstruiert werden
- aus `A =:= B` und `B =:= C` kann nicht `A =:= C` konstruiert werden
- aus `A =:= B` kann nicht `F[A] =:= F[B]` konstruiert werden
- $\rightsquigarrow$ keine vollständige Äquivalenz auf Typebene

Naive Implementation

Nachteile

- aus `A =:= B` kann nicht `B =:= A` konstruiert werden
- aus `A =:= B` und `B =:= C` kann nicht `A =:= C` konstruiert werden
- aus `A =:= B` kann nicht `F[A] =:= F[B]` konstruiert werden
- $\rightsquigarrow$ keine vollständige Äquivalenz auf Typebene

Naive Implementation

Nachteile

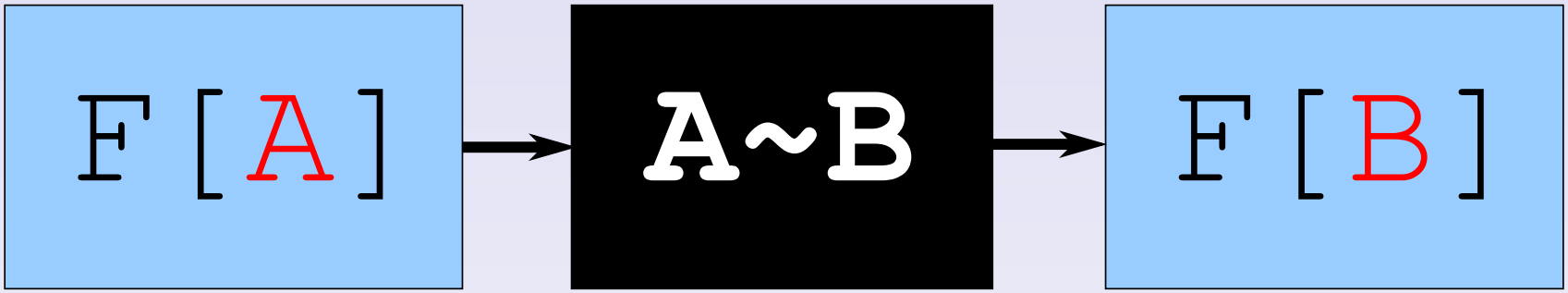
- aus `A =:= B` kann nicht `B =:= A` konstruiert werden
- aus `A =:= B` und `B =:= C` kann nicht `A =:= C` konstruiert werden
- aus `A =:= B` kann nicht `F[A] =:= F[B]` konstruiert werden
- $\rightsquigarrow$ keine vollständige Äquivalenz auf Typebene

Implementation mit Kinds

```
trait Leibniz[A,B] {
    def subst[F[_]](p: F[A]) : F[B]
}
```

abgekürzt als `A ~ B`

Implementation mit Kinds



Implementation mit Kinds

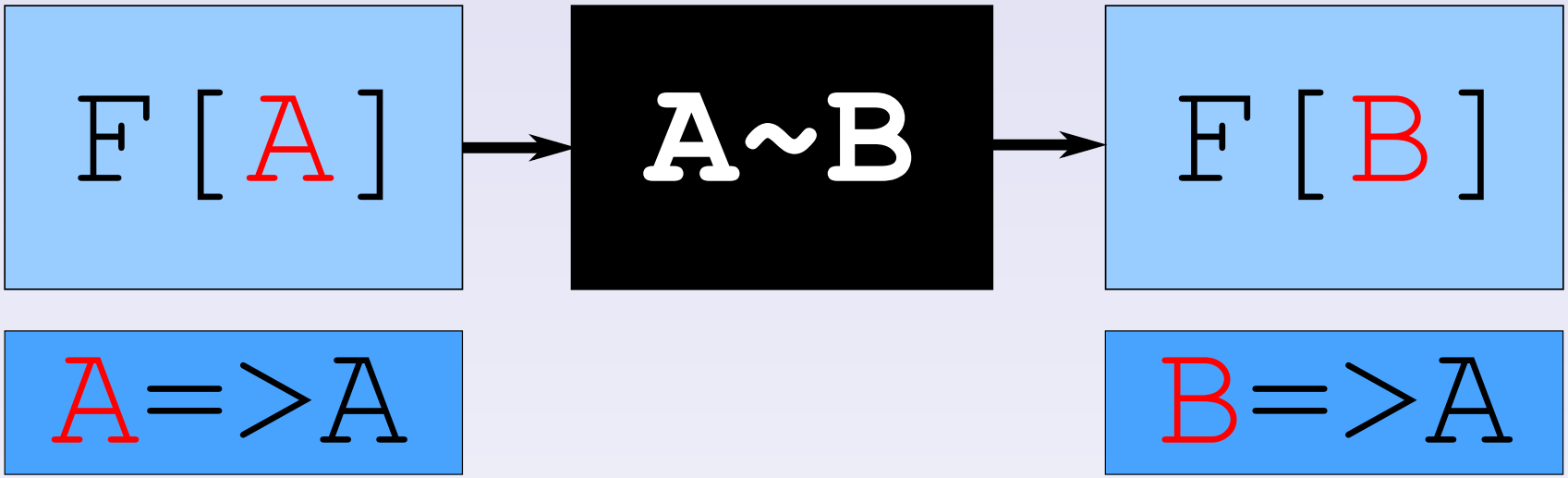
Konvertierungsfunktion

Wie bekommt man aus `A ~ B` ein `A => B`?

Implementation mit Kinds

Konvertierungsfunktion

Wie bekommt man aus `A ~ B` ein `A => B`?



Implementation mit Kinds

Konvertierungsfunktion

```
def witness[A,B](f: A ~ B): A => B =
    f.subst(identity[A] _)
```

Implementation mit Kinds

Konvertierungsfunktion

```
def witness[A,B](f: A ~ B): A => B =
    f.subst(identity[A] _)
```

```
argument expression's type is not compatible
with formal parameter type;
 found   : A => A
 required: ?F[A]
```

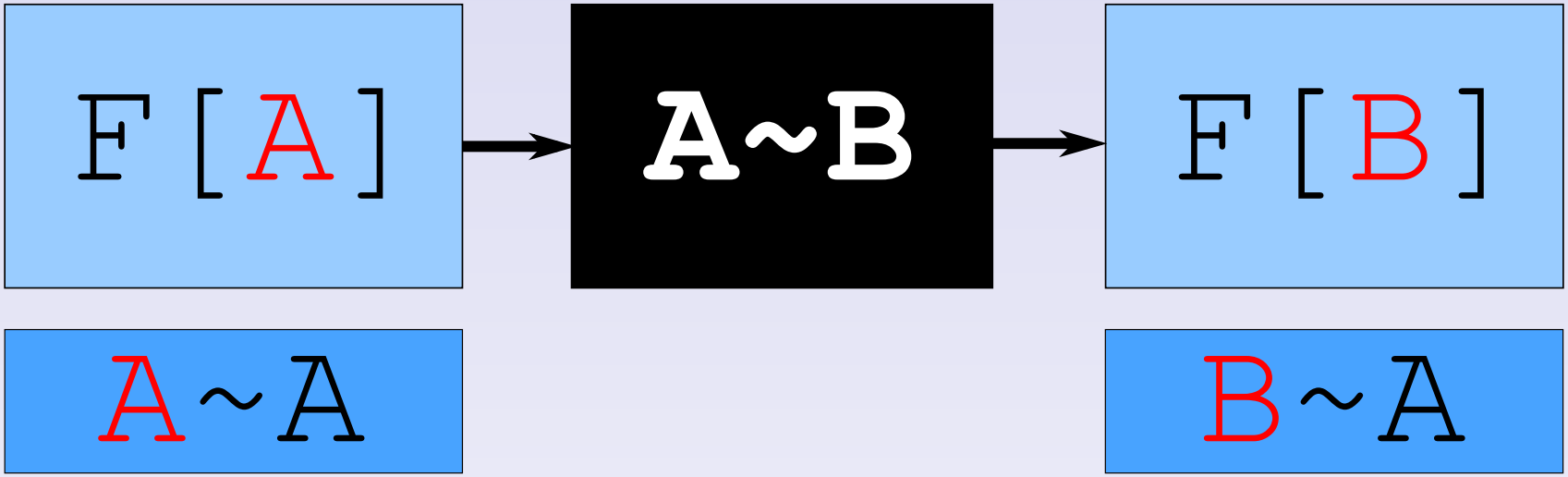
Implementation mit Kinds

Konvertierungsfunktion

```
def witness[A,B](f: A ~ B): A => B =
    f.subst[({type L[X]=A => X})#L](identity)
```

Implementation mit Kinds

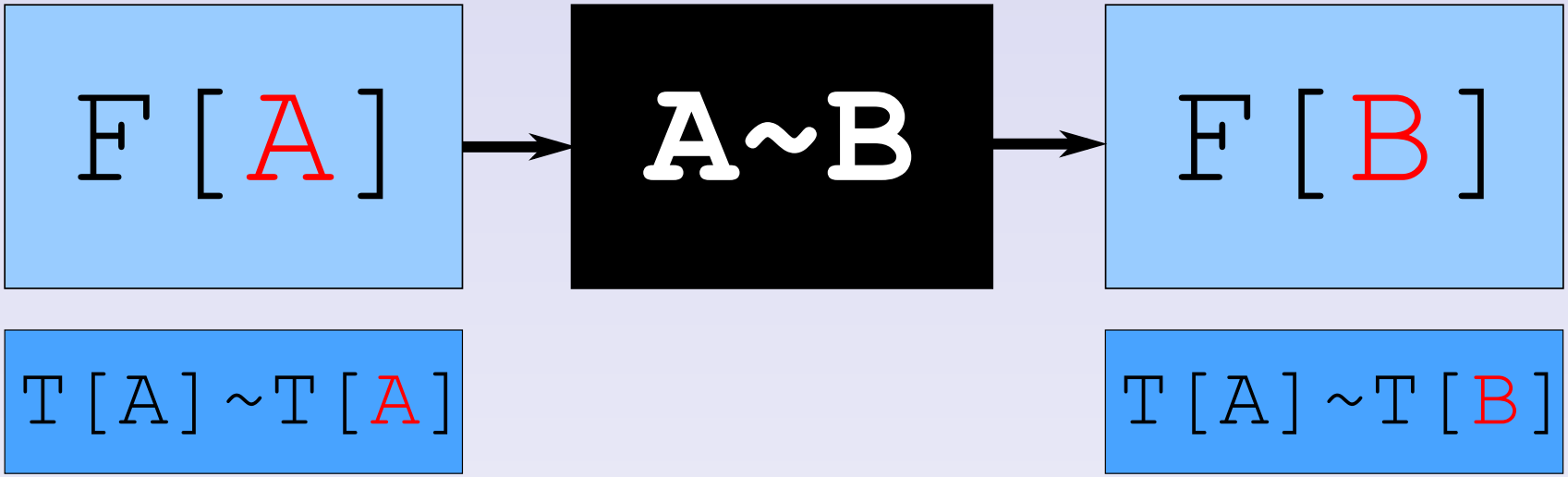
Inversion




```
def symm[A,B](f: A ~ B) : B ~ A =
    f.subst[({type L[X]=X ~ A})#L](refl)
```

Implementation mit Kinds

Lifting




```
def lift[T[_], A, B](a: A ~ B): (T[A] ~ T[B]) = {
    type L[X] = T[A] ~ T[X]
    a.subst[L](refl)
}
```

Heterogeneous Lists

Problem

Funktionen mit mehreren Rückgabewerten

Standardlösung: Verwendung von Tupeln

Heterogeneous Lists

Nachteile von Tupeln

- in Scala: jede Stelligkeit eigene Klasse
- $\rightsquigarrow$ begrenzte Stelligkeit

Heterogeneous Lists

Nachteile von Tupeln

- in Scala: jede Stelligkeit eigene Klasse
- $\rightsquigarrow$ begrenzte Stelligkeit

Lösung: Heterogeneous Lists
(statisch typisierte) Listen mit uneinheitlichen Elementtypen

Heterogeneous Lists

Nachteile von Tupeln

- in Scala: jede Stelligkeit eigene Klasse
- $\rightsquigarrow$ begrenzte Stelligkeit

Lösung: Heterogeneous Lists
(statisch typisierte) Listen mit uneinheitlichen Elementtypen

Implementation

```
sealed trait HList

final case class HCons[H, T <: HList](
  head: H, tail: T
) extends HList {
  def ::[T](v: T) = HCons(v, this)
}

case object HNil extends HList {
  def ::[T](v: T) = HCons(v, this)
}
```

Verwendung

Erzeugung

```
> def getValues = 3 :: "foo" :: HNil
getValues: HCons[Int,HCons[String,HNil.type]]

> getValues
res0: HCons[...] = HCons(3,HCons(foo,HNil))
```

Verwendung

Pattern Matching

```
> getValues match { case HCons(x, _) => x }
res0: Int = 3

> getValues match {
    case HCons(_, HCons(y, _)) => y
  }
res1: String = foo
```

Verwendung

Indexbasierter Zugriff

- eine Funktion `HList[H, T] => Int => Any` möglich, aber nicht hilfreich
- Idee: `index`-Funktion muss einen Typparameter haben, der die Position in der Liste repräsentiert

Lösung: Peano-Numerale

Verwendung

Indexbasierter Zugriff

- eine Funktion `HList[H, T] => Int => Any` möglich, aber nicht hilfreich
- Idee: `index`-Funktion muss einen Typparameter haben, der die Position in der Liste repräsentiert

Lösung: Peano-Numerale

Peano-Numerale

Wertbasiert

```
trait Nat
case object Zero
case class Succ(n: Nat)

val two = Succ(Succ(Zero))
```

Peano-Numerale

Typbasiert

```
trait Nat
trait Zero extends Nat
trait Succ[N <: Nat] extends Nat

type Two = Succ[Succ[Zero]]
```

Peano-Numerale und HLists

Zugriffstrait

```
trait AccessN[L <: HList, N <: Nat, R] {
    def apply(l: L): R
}
```

Peano-Numerale und HLists

HCons revisited

```
case class HCons[H, T <: HList](head: H, tail: T)
  extends HList {

    def apply[N <: Nat, R](n: N)
      (implicit access: AccessN[HCons[H, T], N, R]) =
        access(list)

}
```

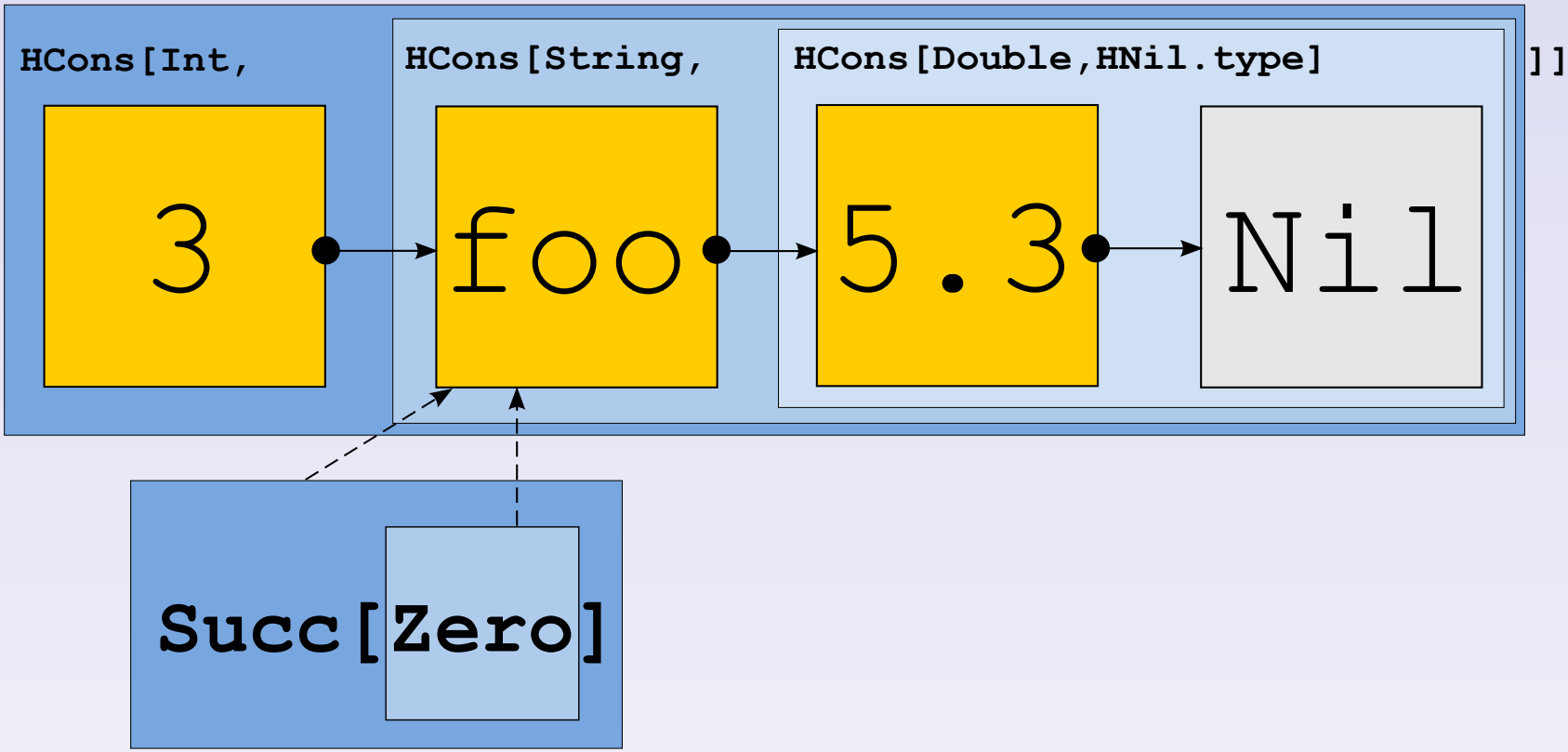
Peano-Numerale und HLists

Implementation

```
implicit def accessZero[H, T <: HList] =
    new AccessN[HCons[H, T], Zero, H] {
        def apply(l: HCons[H, T]) = l.head
    }

implicit def accessN[H, T <: HList, N <: Nat, R]
  (implicit accessTail: AccessN[T, N, R]) =
    new AccessN[HCons[H, T], Succ[N], R] {
        def apply(l: HCons[H, T]) = accessTail(l.tail)
    }
```

Peano-Numerale und HLists



Peano-Numerale und HLists

Verwendung

```
val _0 = new Zero {}
val _1 = new Succ[Zero] {}
val _2 = ...
```

Peano-Numerale und HLists

Verwendung

```
> val list = 3 :: "foo" :: 5.3 :: HNil

> list(_0)
res0: Int = 3

> list(_1)
res1: String = foo

> list(_3)
error: could not find implicit value for parameter ...
```

Heterogeneous Lists

Vorteile

- Verwalten von verschiedenen Typen, ohne eine Klasse anlegen zu müssen
- Tupel beliebiger Stelligkeit
- kein Verlust von Typinformationen verglichen mit `List[Any]`
- $\rightsquigarrow$ Compiler kann falsche Verwendung erkennen
- dank Scalas Syntaxzucker „fühlen“ sich HLists wie normale Listen an
- sind auch in Java möglich, aber viel Overhead nötig

Anwendung

Problem: Signatur von `String.format` stellt nicht sicher, dass die Argumente zum Formatstring passen
$\rightsquigarrow$ Compiler kann Fehler nicht aufspüren

Idee: Funktion konstruieren, die eine `HList` von den Argumenten nimmt und diese formatiert

Anwendung

Problem: Signatur von `String.format` stellt nicht sicher, dass die Argumente zum Formatstring passen
$\rightsquigarrow$ Compiler kann Fehler nicht aufspüren

Idee: Funktion konstruieren, die eine `HList` von den Argumenten nimmt und diese formatiert

Typsicheres printf

Implementation

Demo

Typsicheres printf

Vorteile

- Compiler kann Formatierung prüfen
- $\rightsquigarrow$ weniger Exceptions
- selbst definierte Operatoren erhöhen (richtig eingesetzt) die Lesbarkeit

Typsicheres printf

Nachteile

- gezeigte Lösung vereinfacht, benötigt noch Tricks zum Laufen
- Fehlermeldungen können sehr unübersichtlich werden
- „Denken in Typen“ erfordert in der Regel einige Eingewöhnung

Fragen?

http://www.lars-hupel.de
http://gplus.to/larsrh