GETTING HIGHER

HIGHER KINDED AND HIGHER RANK TYPES IN SCALA

Mark Hopkins
@antiselfdual
www.antiselfdual.com

packetloop

ARBOR NETWORKS

- Packetloop was formed in May 2011 by three security consulting experts
- Startup acquired by Arbor Networks Sept 2013
- Security analytics as a service
- Realtime event processing, big data
- Development in Scala and Javascript
- pravail.com

Higher order functions

Higher kinded types

Higher rank types

TYPE THEORY

$$\dfrac{\dfrac{\dfrac{}{f : \forall\, \alpha_1\, (\alpha_1) \vdash f : \forall\, \alpha_1\, (\alpha_1)}\,(\text{var})}{f : \forall\, \alpha_1\, (\alpha_1) \vdash f(\alpha_2 \to \alpha_2) : \alpha_2 \to \alpha_2}\,(\text{spec}) \quad \dfrac{\dfrac{}{f : \forall\, \alpha_1\, (\alpha_1) \vdash f : \forall\, \alpha_1\, (\alpha_1)}\,(\text{var})}{f : \forall\, \alpha_1\, (\alpha_1) \vdash f\, \alpha_2 : \alpha_2}\,(\text{spec})}{\dfrac{f : \forall\, \alpha_1\, (\alpha_1) \vdash f(\alpha_2 \to \alpha_2)(f\, \alpha_2) : \alpha_2}{f : \forall\, \alpha_1\, (\alpha_1) \vdash \Lambda\, \alpha_2\, (f(\alpha_2 \to \alpha_2)(f\, \alpha_2)) : \forall\, \alpha_2\, (\alpha_2)}\,(\text{gen})}\,(\text{app})}{\{\,\} \vdash \lambda\, f : \forall\, \alpha_1\, (\alpha_1)\, (\Lambda\, \alpha_2\, (f(\alpha_2 \to \alpha_2)(f\, \alpha_2))) : (\forall\, \alpha_1\, (\alpha_1)) \to \forall\, \alpha_2\, (\alpha_2)}\,(\text{fn}).$$

TYPE THEORY

TYPE THEORY

A magical wonderland of possibilities……

Higher kinded and higher rank types are cool

Simon Peyton Jones calls them “sexy types”

Not just of academic interest:

they give programmers…

SUPER POWERS

SUPERPOWERS?

In languages like Java we rely on a lot of magic to Get Stuff Done.

- Either magic baked into the language
  e.g. serialisation, enums, iterators

- Or crazy hacks that step outside of the language

- Or unsafe features like down casting, which break the type system.

- In Scala these “higher” features let us build a lot of the magic ourselves
  …. safely and without going outside the language

OBJECTIVE

- Explain a couple of terms from type theory you may come across
- Hopefully explain why I'm excited about them
- Convince you they're not scary
- … perhaps already something you already know
- … they enable powerful ways of programming
- … mention a couple of potential pitfalls

TYPES ARE ABOUT

SAFETY

WHAT IS A TYPE SYSTEM?

A lightweight, ubiquitous, machine-checked formal system for proving the absence of certain bad program behaviours.

- lightweight, so programmers can use it
- machine-checked, with minimal programmer help
- ubiquitous, so programmers can’t avoid it

A mini proof checker or static verification tool

A type system is a (tractable) syntactic method for automatically checking the absence of certain erroneous behaviors by classifying program phrases according to the kinds of values they compute.

– BENJAMIN PIERCE

3 + 4 ✓

3 + true ✗

SAFE VS UNSAFE LANGUAGES

SAFE VS UNSAFE LANGUAGES

Safe languages make it impossible to shoot yourself in the foot while programming

SAFE VS UNSAFE LANGUAGES

Safe languages prevent
untrapped errors
at runtime

e.g. reading data beyond end of array,
SIGSEGV

SAFE VS UNSAFE LANGUAGES

A safe languages protects its own abstractions
Things behave themselves.

e.g. you can use an array just by knowing about operations on arrays, not having to worry about how things are laid out in memory.

Applies to both low level and to higher level features, e.g. scoping, privacy

	STATICALLY CHECKED	DYNAMICALLY CHECKED
SAFE	Scala, Java, Haskell, ML	Lisp, Scheme, Perl, Ruby, Python
UNSAFE	C, C++	

"The fundamental problem addressed by a type theory is to insure that programs have meaning.

The fundamental problem caused by a type theory is that meaningful programs may not have meanings ascribed to them.

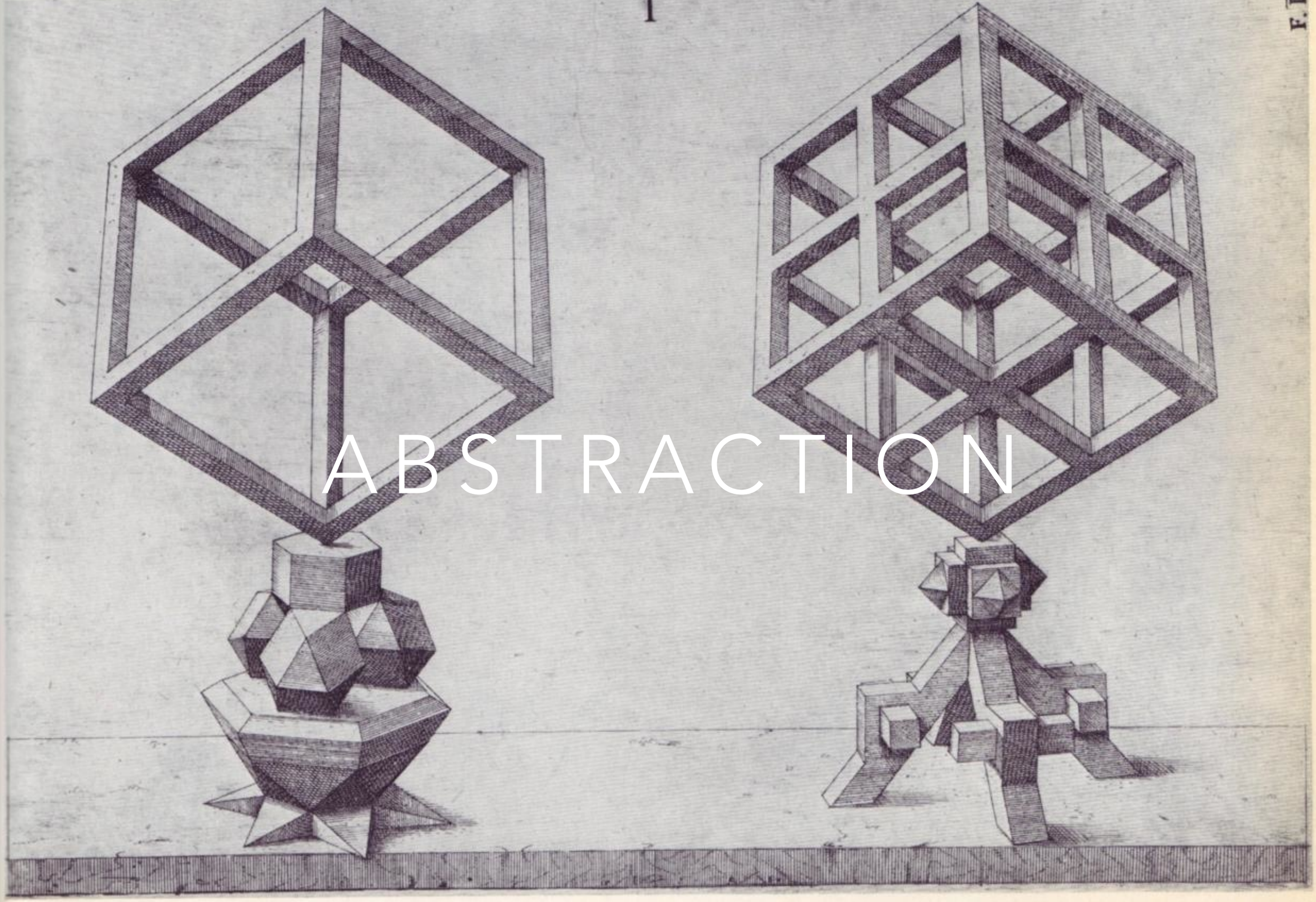
The quest for richer type systems results from this tension."

–MARK MANASSE

That is to say, the type checker will

- reject some legitimate programs
- allow some incorrect programs

Progress leads to…



ABSTRACTION

ABSTRACTION
EXPRESSIVENESS
POLYMORPHISM

With increasing abstraction and more sophisticated type systems, we can

- screen out more incorrect behaviour, automatically
- type check more valid programs

- Abstraction: reducing code duplication by spotting common patterns

- These common patterns are identified, often becoming first-class citizens of the language

- Write less code to do more, i.e. more expressive

POLYMORPHISM

- Comes in many flavours
- Instead of writing many similar pieces of code, write one, but parametrise it
- Specialise at different parameters to gain different functionality

POLYMORPHISM

Parametrise by…

- value, i.e. functions
- function, i.e. higher order functions
- type: parametric polymorphism (Java's generics),
  inclusion polymorphism (Java's "polymorphism"),
  ad hoc polymorphism (overloading)
- structure, eg. type classes
- effects i.e. monads
- shape: this is to parametrisation by type as 'by function' is to 'by value'
  datatype generic programming
- …..and many more

ABSTRACTION IS POWER

More than just screening out errors

A type system is a conversation between the programmer and the compiler

What kind of conversation?

The programmer and the compiler are in a relationship together.

A bit like a married couple

HASKELL

HASKELL

JAVA

JAVA

Lack of type inference

JAVA

JAVA

Low power, poor support for abstraction
forces hackery:

AspectJ
annotation processors
bytecode manipulation
unsafe casts
reflection

JAVA

JAVA

JAVA

JAVA

SCALA?

SCALA?

Somewhere in the middle

FIRST CLASS CITIZENS

An entity in a programming language is **first-class** if it supports all the operations generally available to other entities.

- it can be named
  val x = 2
- it can be supplied as a parameter
  f(2)
- it can be used anonymously
  val y = sin(**2 * x**)

- Machine language does not have first class variables
- C does not have first class arrays
- Java does not have first class functions
- Scala almost has first class types
  Coq, Agda, Isabelle, Idris have real first-class types

Scala has first-class functions

```
def f: Int => String = i => “hello” * i

f(3) // “hellohellohello”
```

higher order functions
  - functions that accept or return a function

are also first class:

```
def compose[A,B,C]: (B => C) => (A => B) => (A => C) =
f => g => a => f(g(a))

val h = comp[Int,Int,Int](_ * 7)(_ + 3)
h(3)      // 42
```

HIGHER ORDER FUNCTIONS

- Ubiquitous in Scala std lib, collection library
  map, fold, filter, collect, sortBy, ….
- combinators
- user-defined control flow abstractions
- Key to Scala’s expressivity

HIGHER KINDED TYPE

Same thing, but for types

HIGHER KINDED TYPE

Just as

a function that accepts or returns function(s) is a higher-order function

so

a type constructor that accepts or returns type constructor(s) is a higher-kinded type.

HIGHER KINDED TYPES

- trait Functor[F[_]] {
  def map[A, B](fa: F[A])(f: A => B): F[B]
  }

```
val ListFunctor = Functor[List]
val strings = List(“red”, “orange”, “yellow”, “green”)
ListFunctor.map(strings)(_.length)
// List(3, 6, 6, 5): List[Int]
```

HIGHER KINDED TYPES

Similarly

K[_, _, _]

K[_[_]]

K[_[_],_]

K[_[_[_[_[_[_[_[_,_]]]]]]]]

As values are classified by their type,
so types are classified by their kind.

K ::= * | K -> K

String, Int, Any: *

List: * -> *

List[Int]: *

Monad: (* -> *) -> *

Tuple2: * -> * -> *

APPLICATIONS

- Used heavily throughout Scala collection library, Scalaz
- Datatype generic programming / polytypic programming

DATATYPE GENERIC PROGRAMMING

List[Int], Tree[Symbol], BinaryTree[Double]

DATATYPE GENERIC PROGRAMMING

List[Int], Tree[Symbol], BinaryTree[Double]

DATATYPE GENERIC PROGRAMMING

List[Int], Tree[Symbol], BinaryTree[Double]

DATATYPE GENERIC PROGRAMMING

List[Int], Tree[Symbol], BinaryTree[Double]

Abstract over **shapes**, using higher kinded types

DATATYPE GENERIC PROGRAMMING

- Generalised map, fold, unfold, traversals that work for across multiple datatypes (Origami)

- Design patterns as language-level constructs

- Ad-hoc generic programming
  allows eg. rolling your own serialisation mechanism
  that works for any datatype (see Shapless)

PROBLEM 1

NO PARTIAL APPLICATION

```
case class State[S, A](f: S => (A, S))
```

No syntax for State[S, _]

NO PARTIAL APPLICATION

Introduce a new type:

```
type IntState[A] = State[Int, A]

type FromStringFunction[A] = String => A

type TupleWithInt[A] = (A, Int)

type R[A[_]] = List[A[Int]]
```

NO PARTIAL APPLICATION

case class State[S, A](f: S => (A, S))

No syntax for State[S, _]

though there is a compiler plugin

State[S, ?]
Tuple[Int, ?]
Lambda[A => (A,A)] // equivalent to type R[A] = (A, A)

See github.com/non/kind-projector (part of typelevel)

NO PARTIAL APPLICATION

or use a type lambda

```
({type f[a] = State[Int,a]})#f

({type f[a] = String => a})#f

({type f[a, b[_]] = b[a]})#f
```

PROBLEM 2

Inference for higher kinded types is limited

```
case class Base[A](a: A)

case class Recursive[F[_],A](fa: F[A])

val one = Base(1)
println(one)                    // Base(1)

val two = Recursive(one)
println(two)                    // Recursive(Base(1))

val three = Recursive(two)   // compilation error!
```

Have to help the compiler

```
case class Base[A](a: A)

case class Recursive[F[_],A](fa: F[A])

val one = Base(1)
println(one)          // Base(1)

val two = Recursive(one)
println(two)          // Recursive(Base(1))

val three = {
  type λ[α] = Recursive[Base, α]
  Recursive(two : λ[Int])
}
println(three)        // Recursive(Recursive(Base(1)))
```

Or…

```
case class Base[A](a: A)

case class Recursive[F[_],A](fa: F[A])
object Recursive {
 def apply[FA](fa: FA)(implicit u: Unapply[FA]) = new Recursive(u(fa))
}

val one = Base(1)
println(one)          // Base(1)

val two = Recursive(one)
println(two)          // Recursive(Base(1))

val three = Recursive(two)
println(three)        // Recursive(Recursive(Base(1)))
```

Unapply

```
trait Unapply[FA] {
  type F[_]
  type A
  def apply(fa: FA): F[A]
}
```

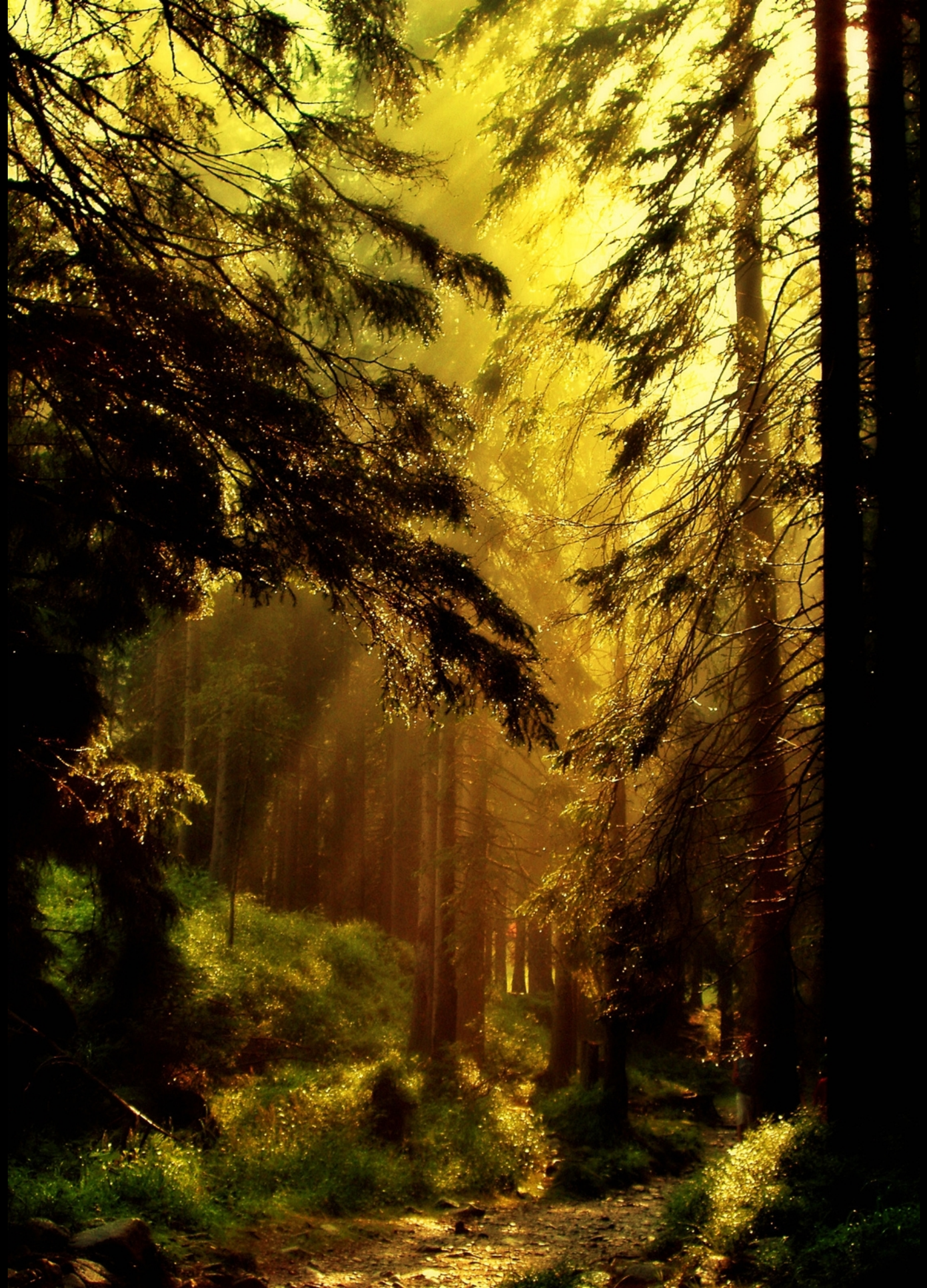
```
object Unapply {
  implicit def unapply[F0[_[_], _], G0[_], A0] = new Unapply[F0[G0, A0]] {
    type F[α] = F0[G0, α]
    type A = A0
    def apply(fa: F0[G0, A0]): F[A] = fa
  }
}
```

PROBLEM 3

Compiler bugs! :(

Variance annotations on higher-kinded type parameters aren’t dealt with correctly. Fixed in 2.11

Workaround: for now, avoid variance annotations on higher-kinded type parameters



HIGHER RANK TYPES

A type is of rank *k* if, when the type is written as a tree, no path from the root of the type to a universal quantifier passes to the left of *k* arrows.


What?

Say you have this
def singleton[A](a: A) = List(a)

Say you have this
def singleton[A](a: A) = List(a)

But now you want to abstract over which particular list-instantiation function you use.

Say you have this
def singleton[A](a: A) = List(a)

But now you want to abstract over which particular list-instantiation function you use.

def createList[A,B](f: A => List[A], b: B) = f(b)

Say you have this
def singleton[A](a: A) = List(a)

But now you want to abstract over which particular list-instantiation function you use.

def createList[A,B](f: A => List[A], b: B) = f(b)

Does not compile!

It’s “not polymorphic enough”
The type variables have to be fixed at the invocation site.
But then f is the wrong type to apply to b.

- In Scala, a function like
  `def f[A,B,C,D]: A => B => C => D`

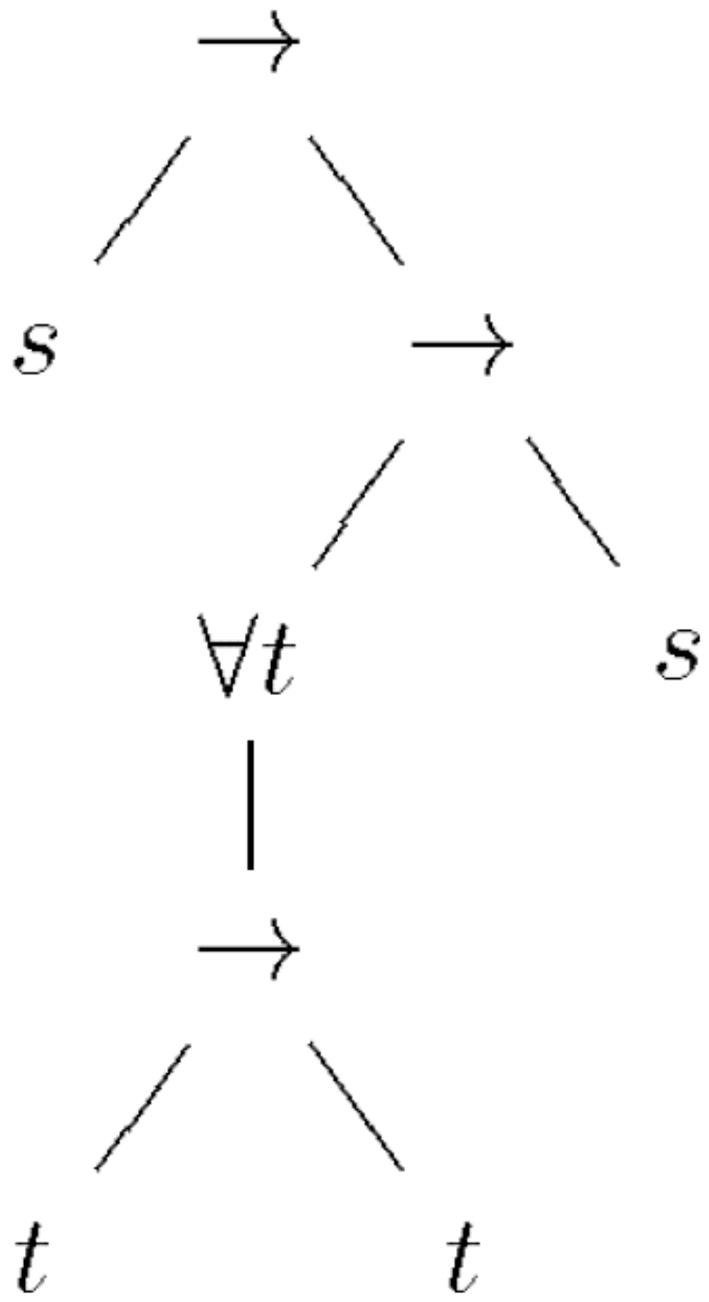
  actually has type
  `∀A.∀B.∀C.∀D A => B => C => D`

  Notice that all the “foralls” are on the outside of the expression

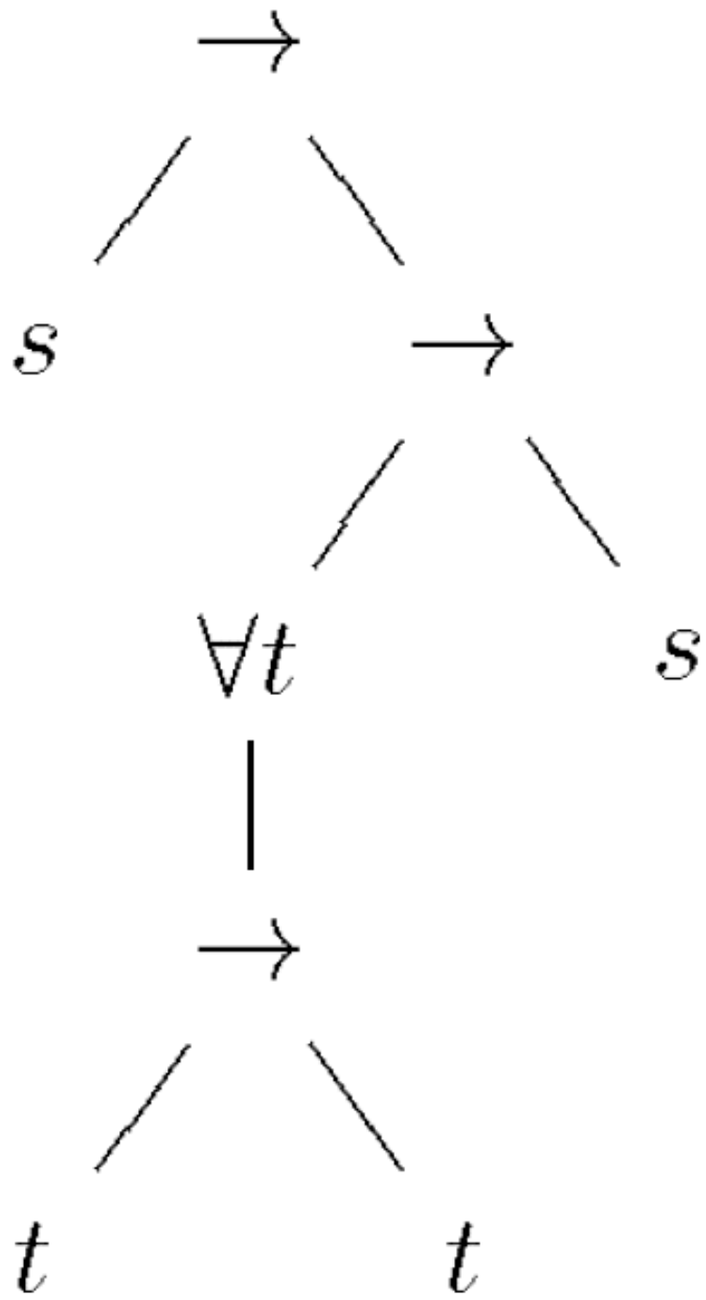
Higher rank types to the rescue!

HIGHER RANK TYPES

A type is of rank *k* if, when the type is written as a tree, no path from the root of the type to a universal quantifier passes to the left of *k* arrows.

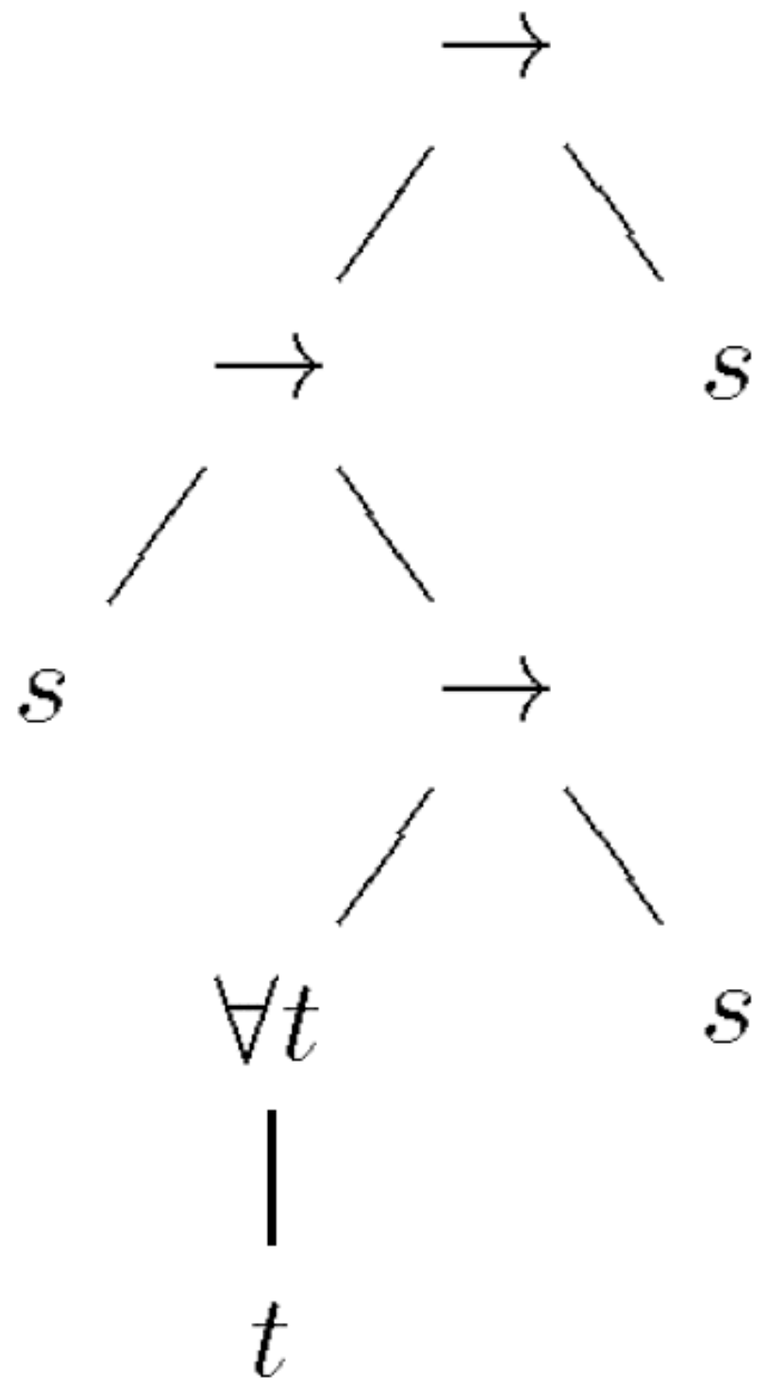


$$s \to (\forall t.t \to t) \to s$$

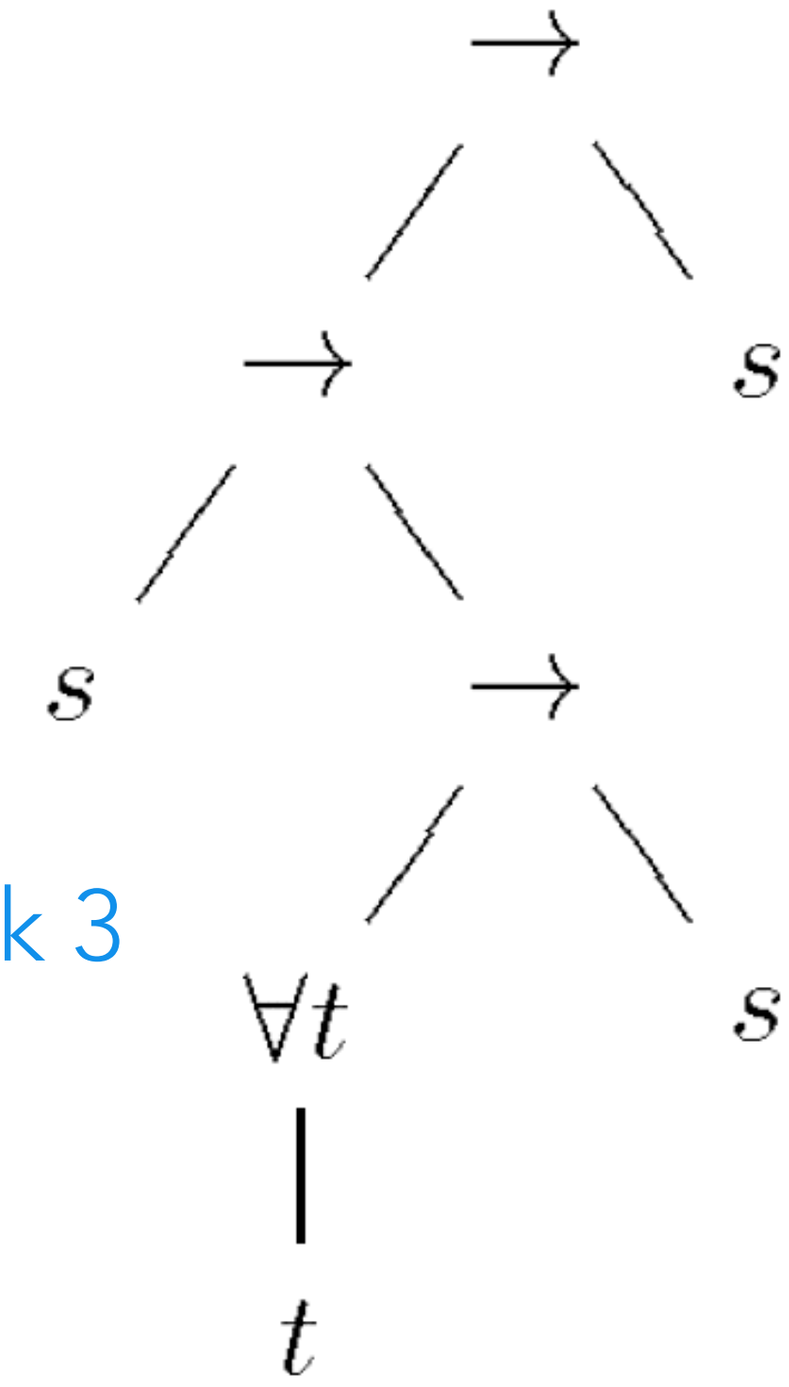


Rank 2

$$s \to (\forall t.t \to t) \to s$$



$$(s \to (\forall t.t) \to s) \to s$$



$$(s \to (\forall t.t) \to s) \to s$$

It’s “not polymorphic enough”
The type variables have to be fixed at the invocation site.
But then f is the wrong type to apply to b.

Higher rank types to the rescue!

```
def createList[B](f: A => List[A] forall A, b: B) = f(b)
```

Now we have enough wiggle room.

But `forall` doesn't actually exist in Scala.

Scala only has rank-1 polymorphism.

And only methods have it, not values.

```
def f[A] = (a: A) => (3, Set(a))
```

`f` has type ∀A. A => (Int, Set[A])

```
val g = (d: Double) => (3, Set(d))
```

g only has type Double => (Int, Set[Double])

But…. we can encode it with objects!

```
trait Forall[P[_] {
  def apply[A]: P[A]
}
```

```
type CreateList[A] = A => List[A]

def createList1[B](
  f: Forall[CreateList],
  b: B
) = f.apply(b)
```

or, with a type lambda,

```
def createList[B](
  f: Forall[({type λ[a] = a => List[a]})#λ],
  b: B
) = f.apply(b)
```

SUCCESS!

So it’s just OO then?

Yes and no

RANK N TYPES

What are they good for?

- Polymorphism for higher kinded types
- Encapsulation for types

RANK N TYPES

What are they good for?

- Enforcing invariants for datatypes
- Datatype generic programming
  as map needs rank-1 polymorphism
  so gmap needs rank-2 polymorphism
- Containment: ST monad, region types
- Abstracting over, converting between or combining monads
- Optimisation of recursive functions (deforestation)

RANK N TYPES

EXAMPLES

SQUARE MATRICES

```
trait Square[V[_], A](rows: V[V[A]])

def lookup[V[_],A]:

  Square[V, A] =>

  Forall[B],(V[B], Int) => B] =>

  (Int, Int) => A
```

STATE THREAD (ST) MONAD

Run code with mutable variables,
but with safety enforced by the compiler

When you create, read, write a variable, it is wrapped in a type labelled with a type parameter S.

```
newVar   :: a -> ST s (Ref s a)
readVar  :: Ref s a -> ST s a
writeVar :: Ref s a -> a -> ST s ()

return   :: a -> ST s a
bind     :: ST s a -> (a -> ST s b) -> ST s b

runST    :: (forall s. ST s a) -> a
```

rank 2!

S cannot leave the “forall”. It is locally scoped because of the universal quantifier. So no mutability can leak - the compiler guarantees this!

STATE THREAD (ST) MONAD

Run code with mutable variables,
but with safety enforced by the compiler

```
def runST[A](f: Forall[({type λ[S] = ST[S, A]})#λ]): A
```

SUMMARY

- Higher kinded types are higher-order type constructors
- Higher rank allows polymorphism for higher kinded types

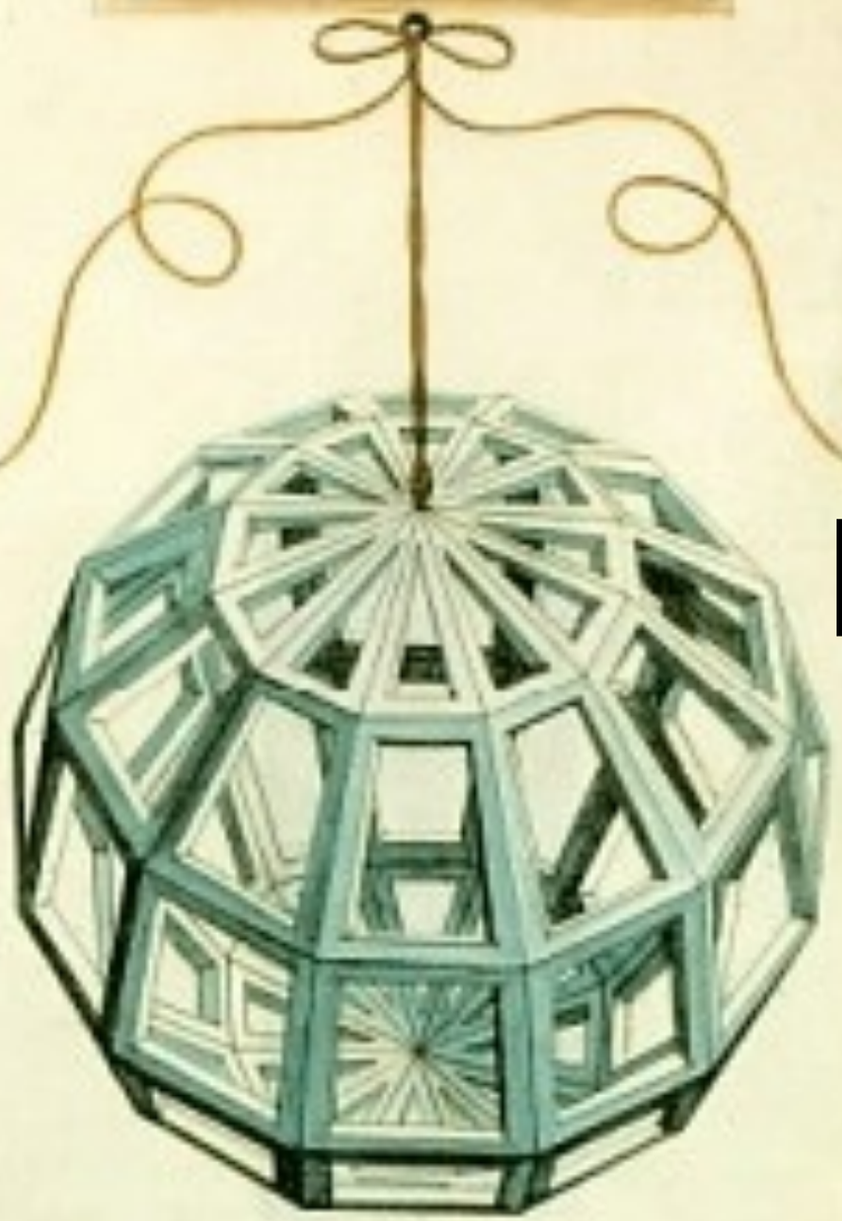
SUMMARY

- Higher kinded types allow us to abstract over type constructors
- Higher rank polymorphism allows us to abstract over polymorphic functions

SUMMARY

- Greater expressiveness and safety
- Locally scoped types
- Abstraction is power

SEPTVAGINTA DVARVM
BASIVM VACVVM.



FINIS

SEPTVAGINTA DVARVM
BASIVM SOLIDVM.

REFERENCES AND FURTHER READING

Types and programming languages, Benjamin C. Pierce

Type Systems, Luca Cardelli

http://stackoverflow.com/questions/15303437/what-are-the-limitations-on-inference-of-higher-kinded-types-in-scala

Scala: Types of a higher kind, Jed Wesley-Smith, 2003, http://blogs.atlassian.com/2013/09/scala-types-of-a-higher-kind/

Generics of a higher kind, Adriaan Moors, Frank Piessens, Martin Odersky, 2008

Wearing the hair shirt: A retrospective on Haskell: Simon Peyton Jones, 2003

Sexy types in action, Chung-chieh Shan

Scala for Generic Programmers, Bruno C. d. S. Oliveira and Jeremy Gibbons, 2008

Rank 2 type systems and recursive definitions, Trevor Ji, 1995

Lazy Functional State Threads, John Launchbury and Simon L Peyton Jones, 1994

Origami programming, Jeremy Gibbons, 2003

“When can Liskov be lifted?”, blog post by Stephen Compall, 2014, http://typelevel.org/blog/2014/03/09/liskov_lifting.html