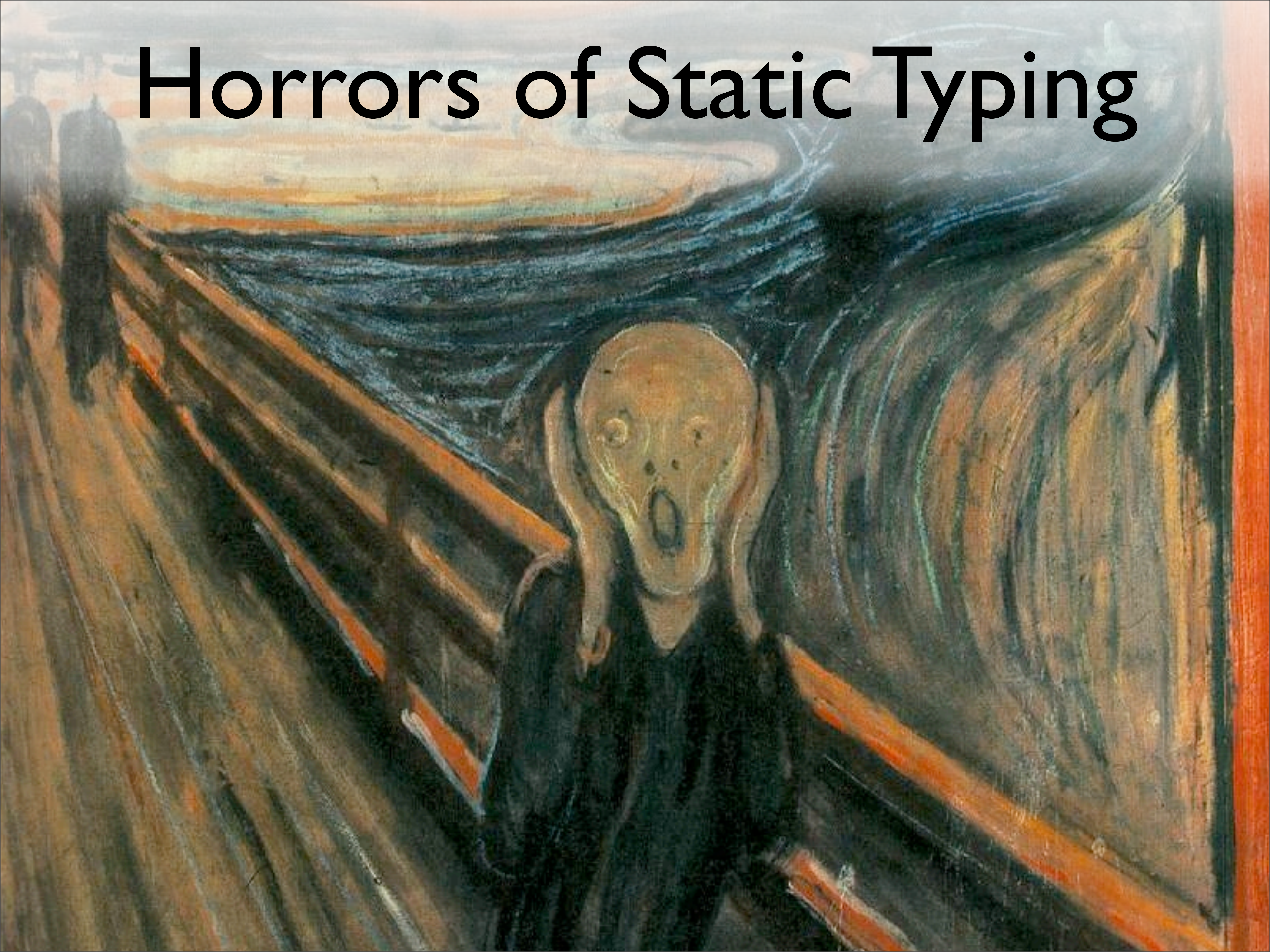
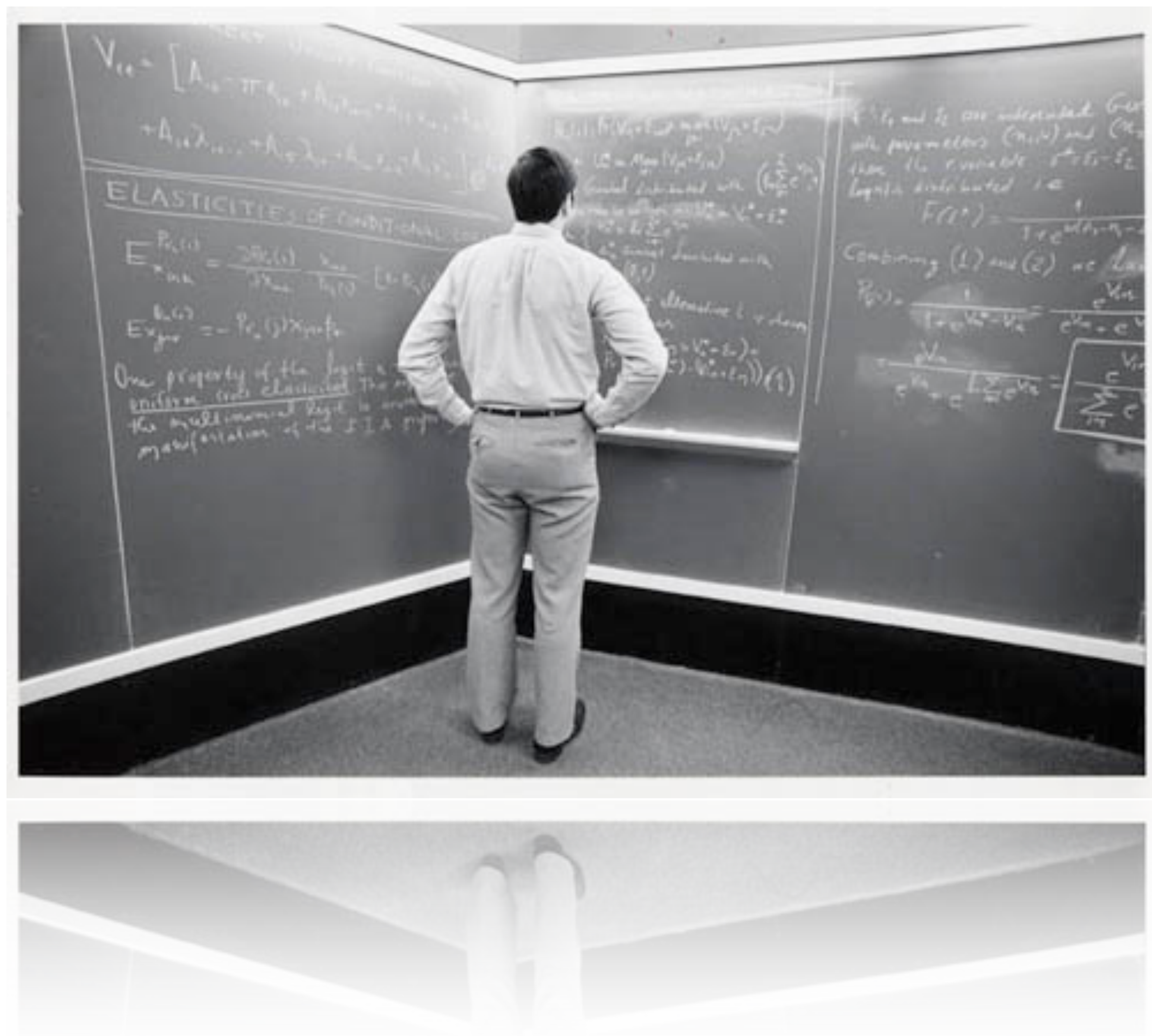
Horrors of Static Typing



Type Theory 101



A type system is a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute. – Benjamin Pierce

A type system is a tractable ***syntactic method*** *for* ***proving the absence*** *of certain program behaviors by classifying phrases according to the* ***kinds of values*** *they compute.* – Benjamin Pierce

Type Systems

- Obviate certain classes of errors
  - Progress and preservation
- Often used to encode constraints
- Serve as a form of syntactic documentation
- *Generally* require up-front assertions

Logic

- Curry-Howard isomorphism
- Types are propositions (assertions)
- Values are proofs of propositions
  - *Terms* are evidence, not proof
  - Non-termination is a problem

Types	Logic
$Unit$	$\textsc{True}$
$\forall_\tau \ \tau$	$\textsc{False}$
$T_1 \to T_2$	$P \to Q$
$T_1 \times T_2$	$P \wedge Q$
$T_1 + T_2$	$P \vee Q$

To oppose types is to oppose logic. – Paul Snively

Type systems are…

Type systems are…

- …confusing

Type systems are…

- …confusing
- …restrictive

Type systems are…

- …confusing
- …restrictive
- …*annoying*

Numbers

Numbers

- *Ideally*, not a problem

Numbers

- *Ideally*, not a problem
- Computers are finite (surprise!)

Numbers

- *Ideally*, not a problem
- Computers are finite (surprise!)
- Precision: a special case of dependent types

Numbers

- *Ideally*, not a problem
- Computers are finite (surprise!)
- Precision: a special case of dependent types
- Surprisingly tricky to get “right”

Numbers

- *Ideally*, not a problem
- Computers are finite (surprise!)
- Precision: a special case of dependent types
- Surprisingly tricky to get "right"
- Let's not even *think* about sign…

Numbers

- `1 + 2`

Numbers

- `1 + 2`
- `3.14 + 2.72`

Numbers

- `1 + 2`
- `3.14 + 2.72`
- `3.14 + 2`

Numbers

- `1 + 2`
- `3.14 + 2.72`
- `3.14 + 2`
- `1 + 2.72`

Numbers

- `1 + 2`
- `3.14 + 2.72`
- `3.14 + 2`
- `1 + 2.72`
- `2 / 3`

Numbers

- Subtyping is insufficiently expressive
  - `java.lang.Number`, anyone?
- Typeclasses are the most proven solution
- Haskell still has some surprising corners
  - `Real Int` is defined!

```haskell
class Num a where
    (+) :: a -> a -> a
    (-) :: a -> a -> a

    negate :: a -> a


class Num a => Fractional a where
    (/) :: a -> a -> a
```

```
Prelude> :type 42
42 :: (Num t) => t
```

```
(1000 :: Int16) * (1000 :: Int16)
```

```haskell
(1000 :: Int16) * (1000 :: Int16)

-- 16960
```

Numbers

- Can’t catch everything
  - (not even close)

Numbers

- Can’t catch everything
  - (not even close)
- Typeclasses have far-reaching consequences

Numbers

- Can’t catch everything
  - (not even close)
- Typeclasses have far-reaching consequences
- Practical liftings produce surprising results

Numbers

- Can’t catch everything
  - (not even close)
- Typeclasses have far-reaching consequences
- Practical liftings produce surprising results
- *Can we do better?*

Object Collections

Collections

- Not “functional” implementations
- Implementation inheritance
  - Allows much larger set of functions!
- Object-oriented idioms and host language

```scala
val xs: List[String] = List("foo", "bar")
val str: String = xs.head
```

```
def mkString(xs: List[AnyRef]): String =
  xs.fold("") { _.toString + _ }

val strs = List("foo", "bar", "baz")
mkString(strs)
```

```
def mkString(xs: List[AnyRef]): String =
  xs.fold("") { _.toString + _ }

val strs = List("foo", "bar", "baz")
mkString(strs)
```

variance

```
class List[+A] {
  def prepend(a: A): List[A]
}
```

```scala
class List[+A] {
  def prepend(a: A): List[A]
}
```

```
class List[+A] {
  def prepend[B >: A](b: B): List[B]
}
```

```scala
val bs = BitSet(2, 7, 1, 4)

val ss = bs map { _.toString }
```

```
val bs = BitSet(2, 7, 1, 4)

val ss = bs map { _.toString }
```

functional dependencies

Collections

- Variance
  - Use site
  - Declaration site
- Typeclasses
  - Functional dependencies
- Path dependent types

```scala
def map[B, That](f: A => B)
  (implicit bf: CanBuildFrom[Repr, B, That]): That
```

```scala
val ss = bs map { _.toString }


(bs: BitSet)
  .map({ i: Int => s.toString: String })
    (cbf: CanBuildFrom[BitSet, String, Set[String]])
```

```
val ss = bs map { _.toString }
```

```
(bs: BitSet)
  .map({ i: Int => s.toString: String })
    (cbf: CanBuildFrom[BitSet, String, Set[String]])
```

```
val ss = bs map { _.toString }


(bs: BitSet)
  .map({ i: Int => s.toString: String })
    (cbf: CanBuildFrom[BitSet, String, Set[String]])
```

```
val ss = bs map { _.toString }


(bs: BitSet)
  .map({ i: Int => s.toString: String })
    (cbf: CanBuildFrom[BitSet, String, Set[String]])
```

```scala
val ss = bs map { _.toString }


(bs: BitSet)
  .map({ i: Int => s.toString: String })
    (cbf: CanBuildFrom[BitSet, String, Set[String]])
```

```scala
val ss = bs map { _.toString }


(bs: BitSet)
  .map({ i: Int => s.toString: String })
    (cbf: CanBuildFrom[BitSet, String, Set[String]])
```

Collections

- Highlight some massive weirdness in OO
  - …but also some strengths

Collections

- Highlight some massive weirdness in OO
  - …but also some strengths
- *Very* complex to get "right"

Collections

- Highlight some massive weirdness in OO
  - …but also some strengths
- *Very* complex to get “right”
- Still present some unusual challenges
  - Severe bias toward strictness
  - Ugly hacks internally that bypass typing

Functions

λ

Challenge

Write a function that produces the identity function as a value in your language of choice

```scala
def makeId[A, B](a: A): B => B = { b => b }

val id = makeId(42)
id("test")          // error!
```

```
def makeId[A, B](a: A): B => B = { b => b }

val id = makeId(42)
id("test")                // error!
```

```
error: type mismatch;
 found   : java.lang.String("test")
 required: Nothing
              id("test")
                 ^
```

```scala
def makeId[A, B](a: A): B => B = { b => b }

val id: Nothing => Nothing = makeId(42)
id("test")              // error!
```

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


def makeId[A](a: A) =
  new Forall[({ type λ[B] = B => B })#λ] {
    def apply[B] = { b => b }
  }


val id = makeId(42)
id[String]("test")
```

```
makeId :: a -> (forall b . b -> b)
makeId _ b = b
```

SIP-18 (boo!)

```
{-# LANGUAGE ExistentialQuantification #-}

makeId :: a -> (forall b . b -> b)
makeId _ b = b
```

Functions

- Let-bound polymorphism

Functions

- Let-bound polymorphism
  - Scala considers classes to be a binding

Functions

- Let-bound polymorphism
  - Scala considers classes to be a binding
- Higher-rank polymorphism

Functions

- Let-bound polymorphism
  - Scala considers classes to be a binding
- Higher-rank polymorphism
- Cannot be type inferred in general!

Functions

- Let-bound polymorphism
  - Scala considers classes to be a binding
- Higher-rank polymorphism
- Cannot be type inferred in general!
- Combinatorial explosion in complexity

To oppose types is to oppose logic. – Paul Snively

well, not quite...

~~*To oppose types is to oppose logic.* – Paul Snively~~

program without types

~~program~~ without types

proof

~~program~~ without ~~types~~

proof propositions

Dynamic Typing

Dynamic Typing

- Dynamic typing is neither evil nor illogical

Dynamic Typing

- Dynamic typing is neither evil nor illogical
- Types confer very narrow benefits

Dynamic Typing

- Dynamic typing is neither evil nor illogical
- Types confer very narrow benefits
  - *It's not hard to stay on the sidewalk.* – Rich Hickey

Dynamic Typing

- Dynamic typing is neither evil nor illogical
- Types confer very narrow benefits
  - *It's not hard to stay on the sidewalk.* – Rich Hickey
- Types often require a lot of effort

Dynamic Typing

- Dynamic typing is neither evil nor illogical
- Types confer very narrow benefits
  - *It's not hard to stay on the sidewalk.* – Rich Hickey
- Types often require a lot of effort
- Always consider the tradeoff

Dynamic Typing

- Often harder to maintain

Dynamic Typing

- Often harder to maintain
- Potential for a larger class of mistakes

Dynamic Typing

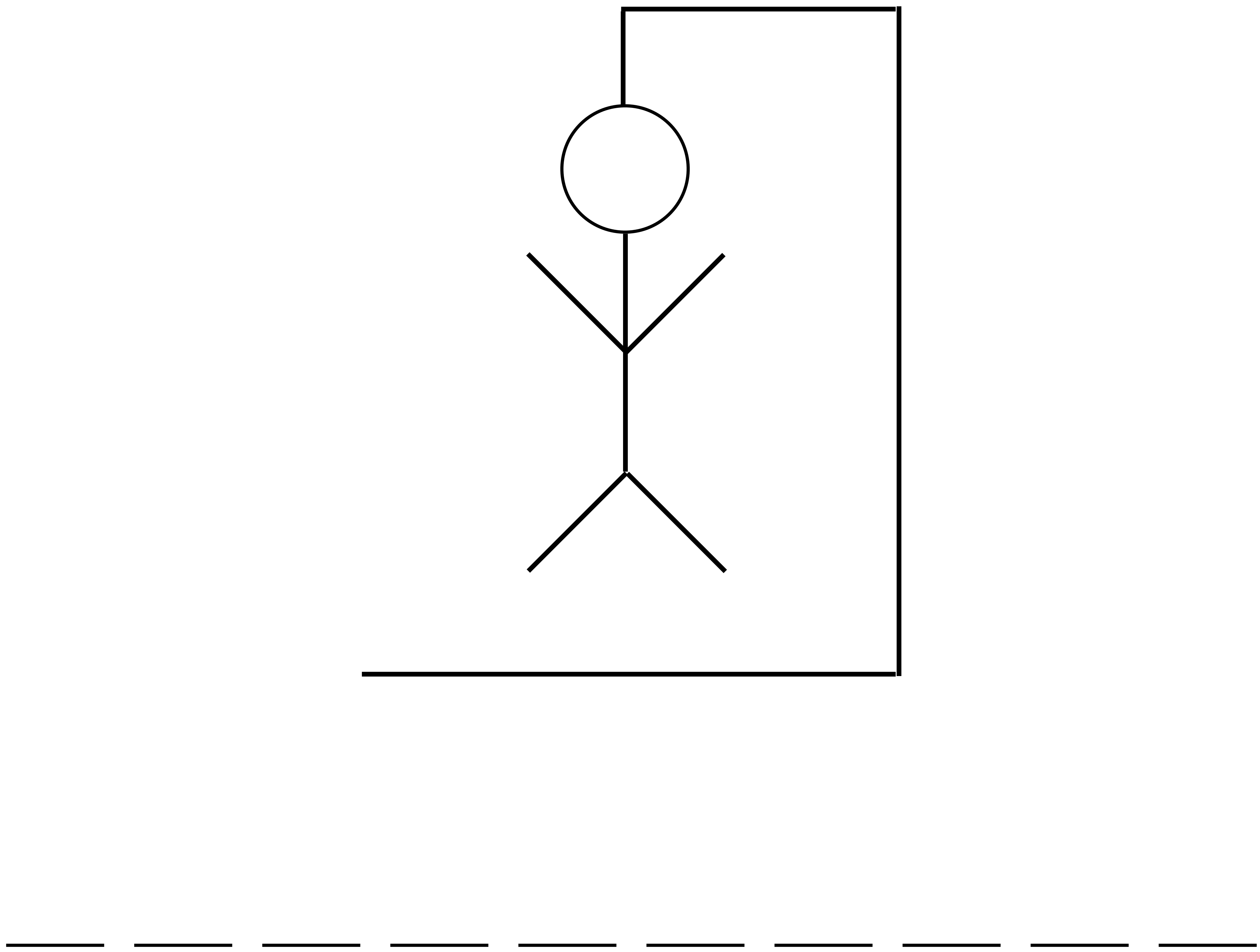
- Often harder to maintain
- Potential for a larger class of mistakes
- Focus on the runtime behavior

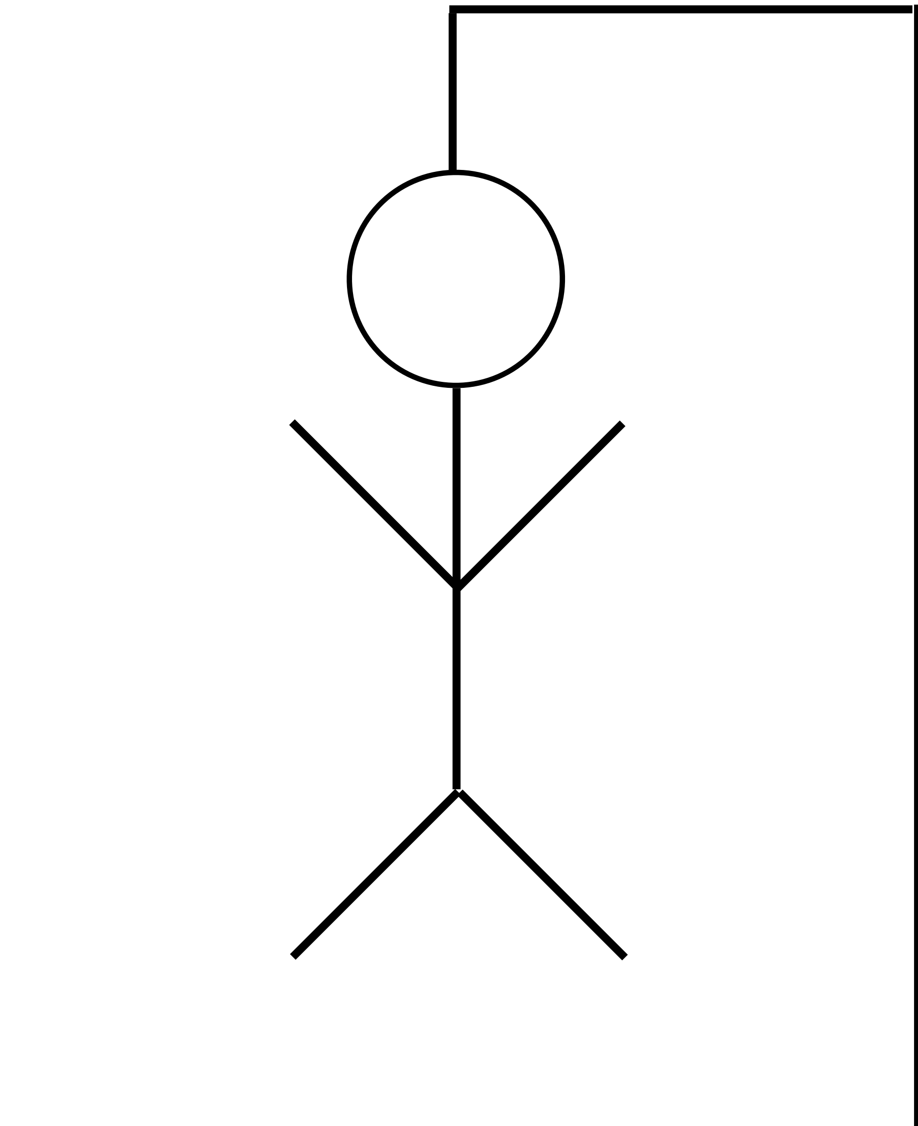
Dynamic Typing

- Often harder to maintain
- Potential for a larger class of mistakes
- Focus on the runtime behavior
  - It’s what we’re paid to do!

Conclusion

- Always weigh the cost/benefit ratio
- Some concepts are not amenable to types
- Type systems are (very) complex
  - …but useful when properly applied





Q U E S T I O N S ?