Fun with kinds and GADTs

Simon Peyton Jones, (Microsoft Research)

with lots of contributions from Dimitrios Vytiniotis, Stephanie Weirich, Brent Yorgey, Julien Cretin, Pedro Magalhaes

August 2013

Focus

Programmers

Programming language implementors

Theorists

Focus

- Type inference

YES

Progr ners

NO

Pr a

la

implementors

NO

Theor

Types are wildly successful

Static typing is by far the most widely-used program verification technology in use today: particularly good cost/benefit ratio

- Lightweight (so programmers use them)
- Machine checked (fully automated, every compilation)
- Ubiquitous (so programmers can't avoid them)

The joy of types

- Types guarantee the absence of certain classes of errors: "well typed programs don't go wrong"
  - True + 'c'
  - Seg-faults
- The static type of a function is a **partial, machine-checked specification**: its says something (but not too much), **to a person**, about what the function does

  reverse :: [a] -> [a]

- Types are a **design language**; types are the UML of Haskell
- Types massively support **interactive program development** (Intellisense, F# type providers)
- The BIGGEST MERIT (though seldom mentioned) of types is their support for **software maintenance**

The pain of types

Sometimes the type system gets in the way

```
data IntList = Nil | Cons Int IntList

lengthI :: IntList -> Int
lengthI Nil            = 0
lengthI (Cons _ xs) = 1 + lengthI xs
```

Now I want a list of Char, but I do not want to duplicate all that code.

Choices

- Dynamically typed language

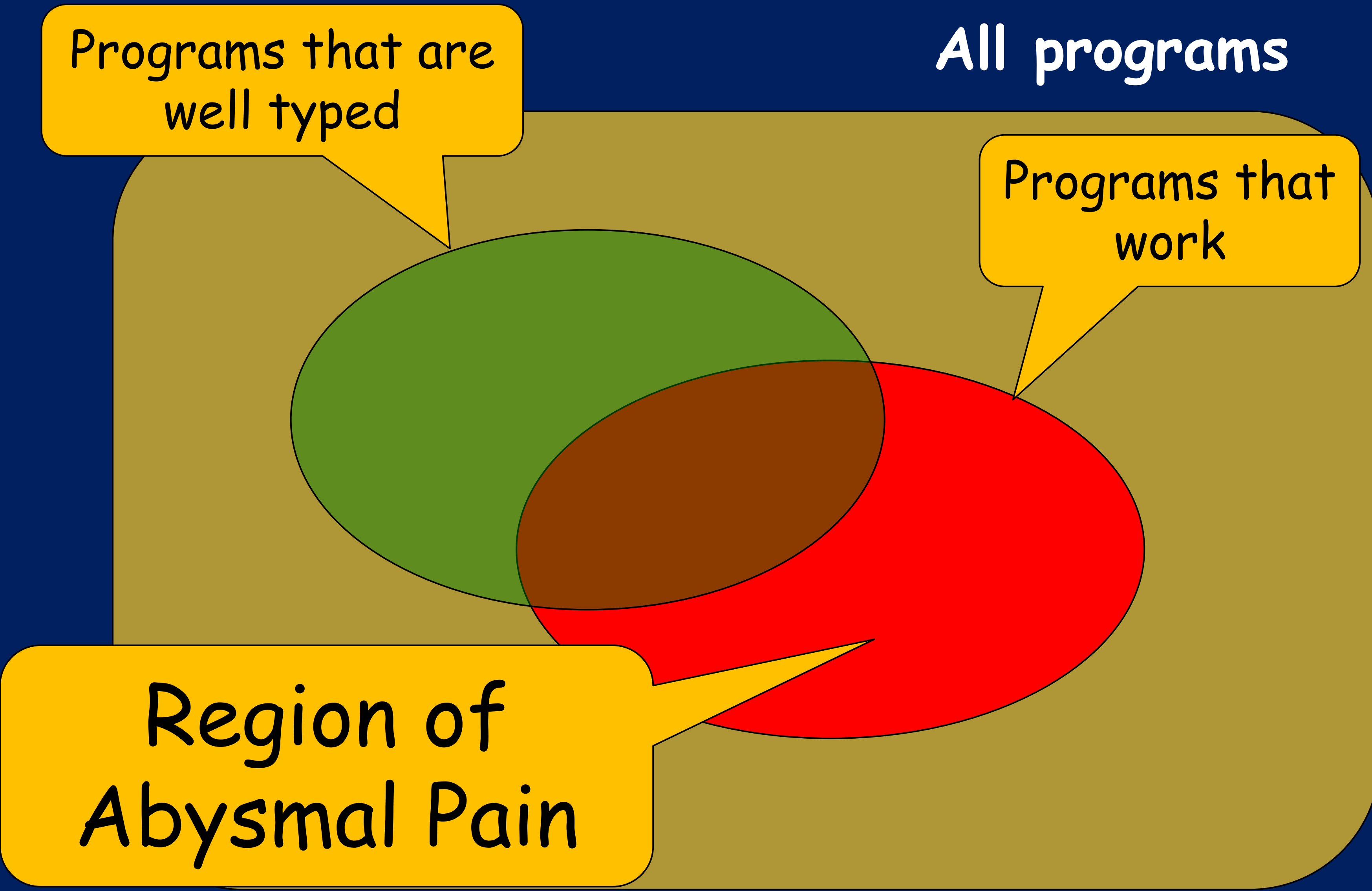
```
lengthI :: Value -> Value
lengthI Nil           = 0
lengthI (Cons _ xs) = 1 + lengthI xs
```

- More sophisticated type system

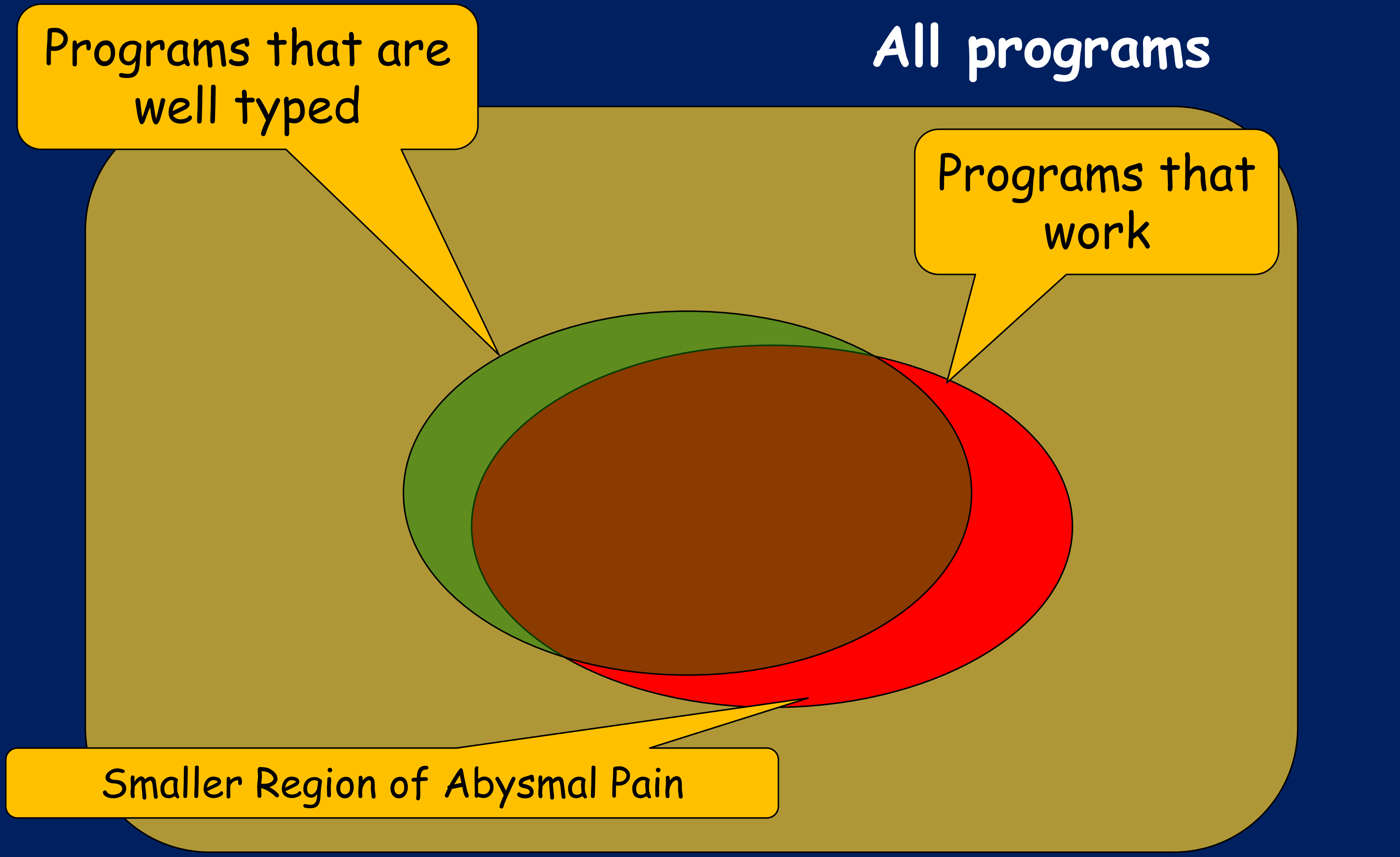
```
data List a = Nil | Cons a (List a)

length :: List a -> Int
length Nil           = 0
length (Cons _ xs) = 1 + length xs
```

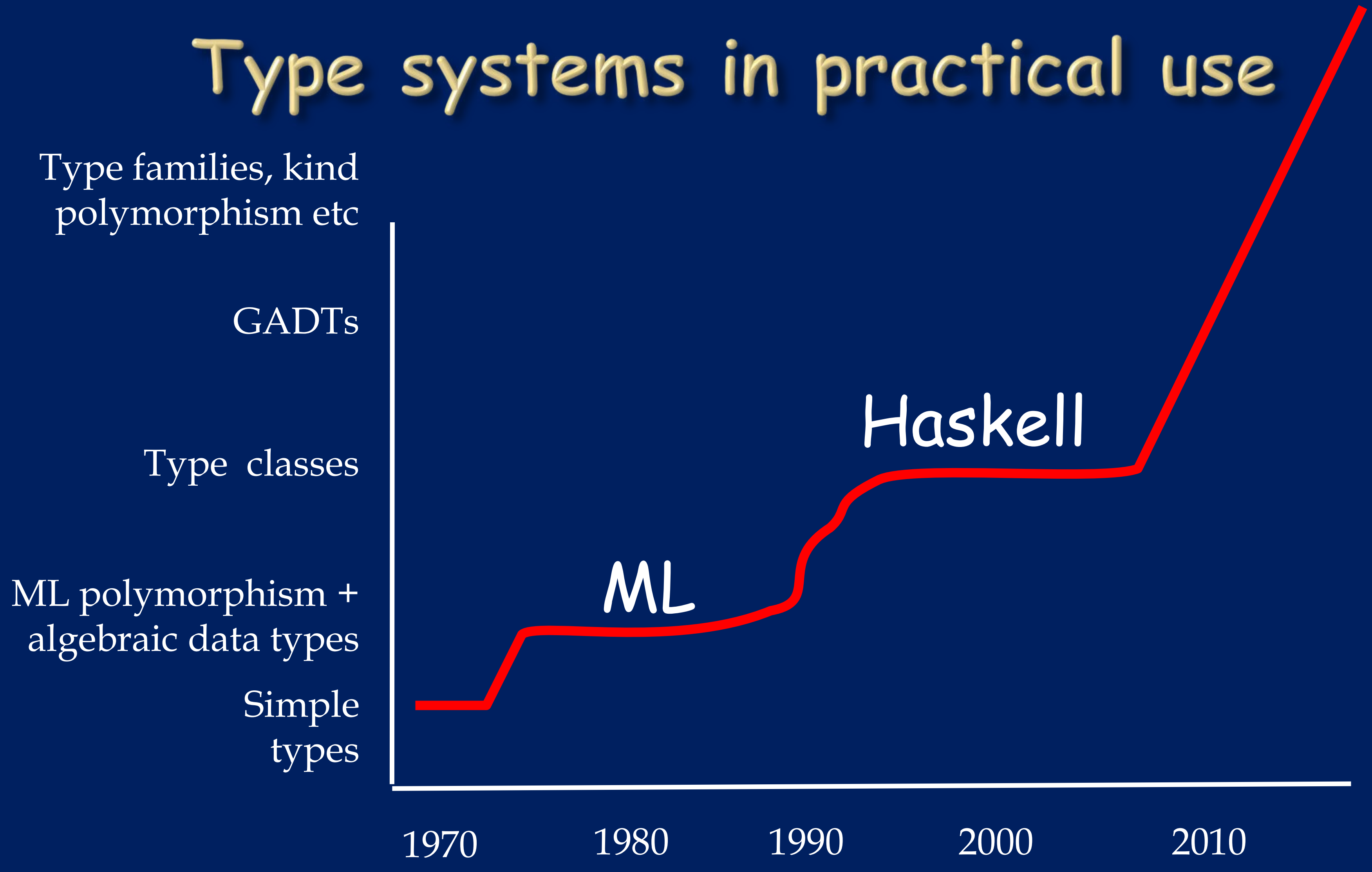
Bad type systems



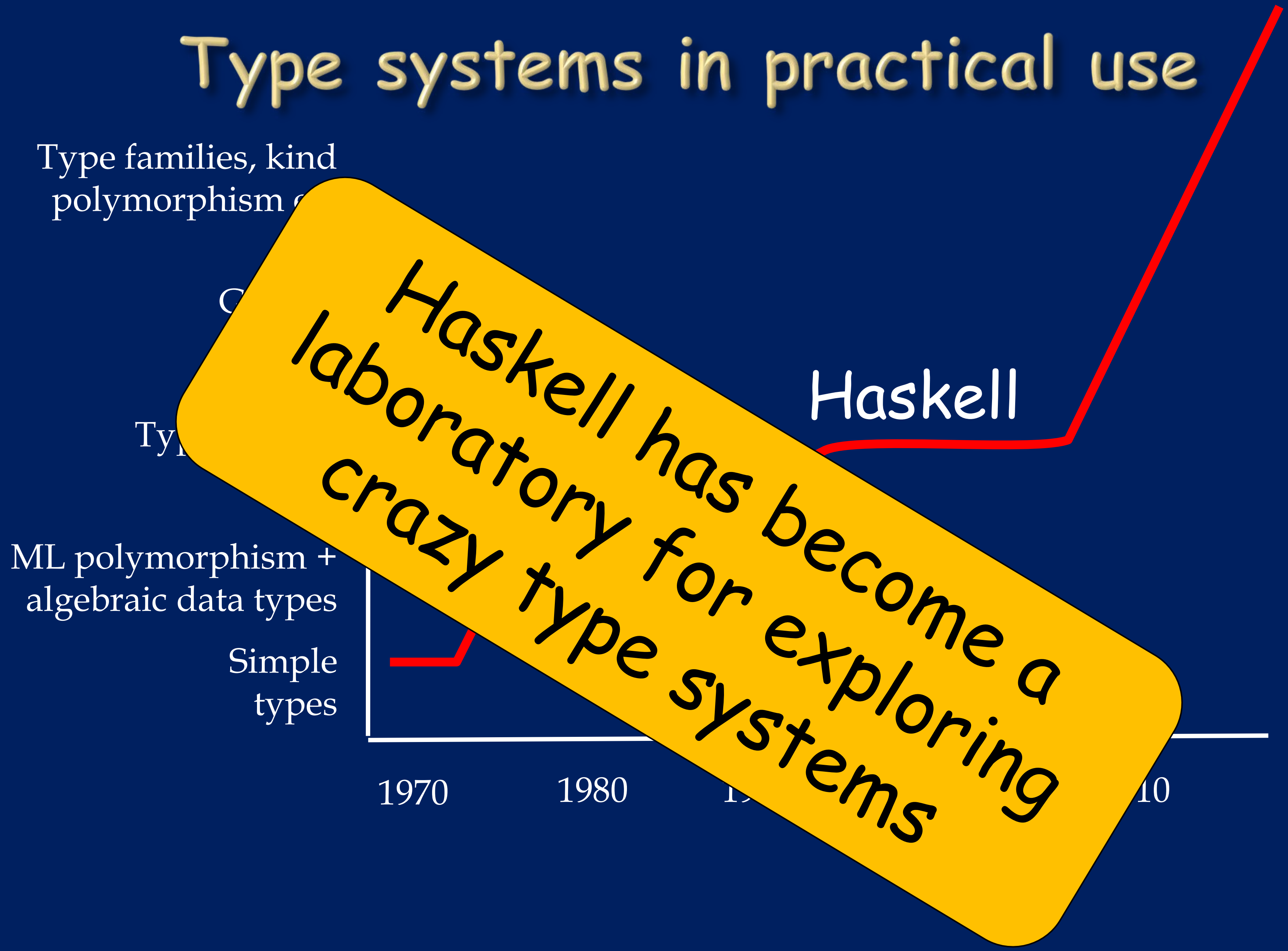
Sexy type systems



Type systems in practical use

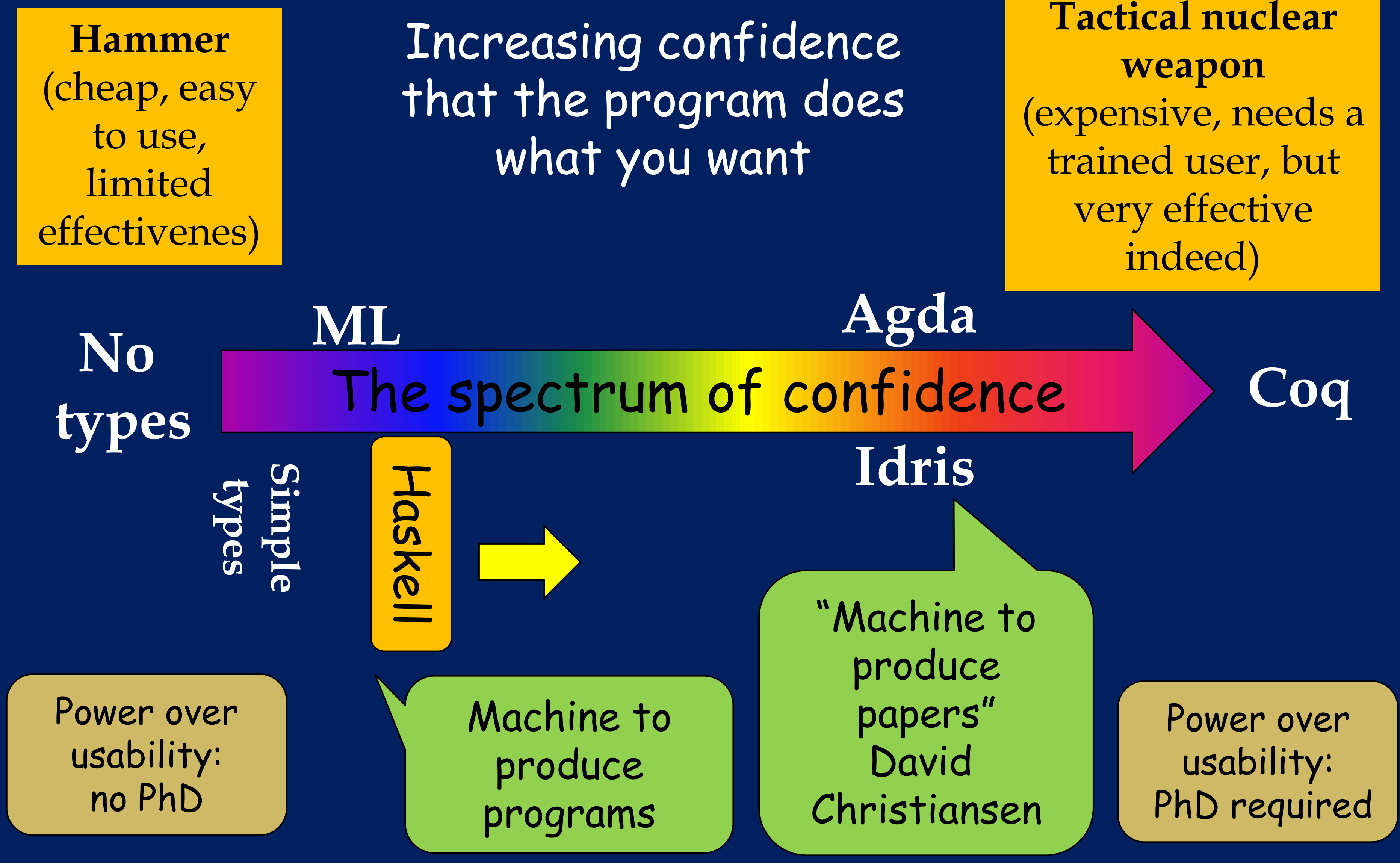


Type systems in practical use



Haskell has become a laboratory for exploring crazy type systems

Type systems vary A LOT



Plan for World Domination

- Build on the demonstrated success of static types
- ...guided by type theory, dependent types
- ...so that more good programs are accepted (and more bad ones rejected)
- ...without losing the Joyful Properties (comprehensible to programmers)

Episode 1

GADTs

GADT syntax

```
data Maybe a  = Nothing | Just a
```

Or

```
data Maybe a where
   Just :: a -> Maybe a
   Nothing :: Maybe a
```

These two declarations mean the same thing

Just like Agda

```
data Term a where
   Lit    :: Int -> Term Int
   Succ   :: Term Int -> Term Int
   IsZero :: Term Int -> Term Bool
   If     :: Term Bool -> Term a -> Term a -> Term a
```

```
eval :: Term a -> a
eval (Lit i)        = i
eval (Succ t)       = 1 + eval t
eval (IsZero i)     = eval i == 0
eval (If b e1 e2) = if eval b then eval e1
                                 else eval e2
```

In here a~Int

What about type inference?

```
data T a where
  T1 :: Bool -> T Bool
  T2 :: T a

f x y = case x of
          T1 z -> True
          T2   -> y
```

- What type should we infer for f?

What about type inference?

```
data T a where
  T1 :: Bool -> T Bool
  T2 :: T a

f x y = case x of
          T1 z -> True
          T2   -> y
```

- f doesn't have a principal type
  - f :: T a -> Bool -> Bool
  - f :: T a -> a -> a
- So reject the definition; unless programmer supplies a type signature for f
- Tricky to specify and implement (e.g. do not want to require type signatures for all functions!)

Example: Hoopl [HS2010]

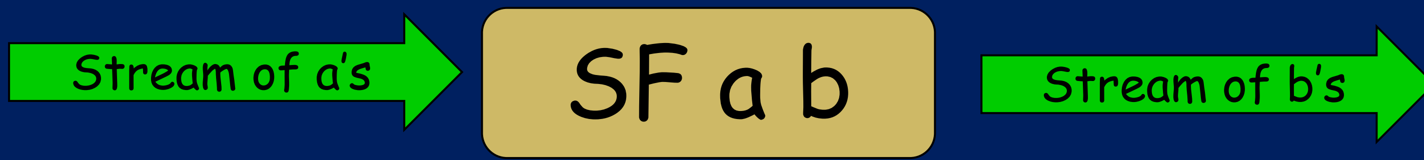
```
data OC = Open | Closed

data Stmt in out where
    Label   :: Label -> Stmt Closed Open
    Assign  :: Reg -> Expr -> Stmt Open Open
    Call    :: Expr -> [Expr] -> Stmt Open Open
    Goto    :: Label -> Stmt Open Closed

data StmtSeq in out where
    Single :: Stmt in out -> StmtSeq in out
    Join :: StmtSeq in Open -> StmtSeq Open out
          -> StmtSeq in out
```

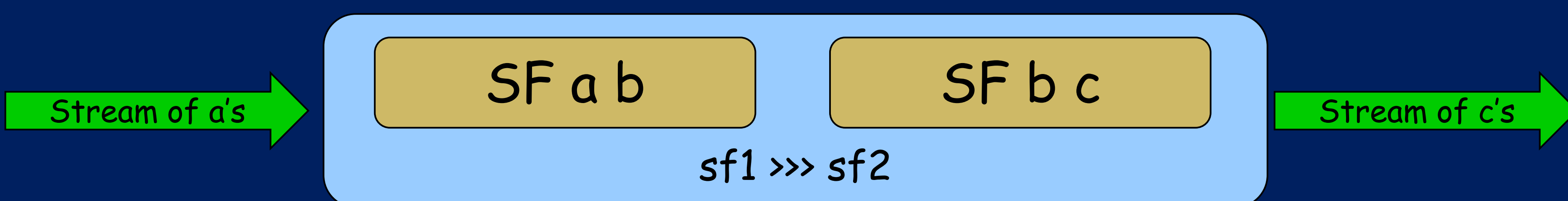
War story: Yampa [ICFP'05]

- Yampa is a DSL for describing stream functions




```
data SF a b where
   -- A function from streams of a's to streams of b's

arr   :: (a->b) -> SF a b
(>>>) :: SF a b -> SF b c -> SF a c
```



War story: Yampa [ICFP'05]

```
data SF a b where
   SF :: (a -> (b, SF a b)) -> SF a b

arr :: (a->b) -> SF a b
arr f = result
   where
      result = SF (\x -> (f x, result))

(>>>) :: SF a b -> SF b c -> SF a c
(SF f1) >>> (SF f2) = SF fr
  where
     fr x = let (r1, sf1) = f1 x
                (r2,sf2) = f2 r1
            in (r2, sf1 >>> sf2)
```

War story: Yampa [ICFP'05]

- GOAL: `arr id >>> f = f`
- This optimisation (and some others like it) is really really important in practice.

```
data SF a b where
   SF :: (a -> (b, SF a b)) -> SF a b
   SFId :: SF a a

sfId :: SF a a
sfId = SFId

(>>>) :: SF a b -> SF b c -> SF a c
SFId >>> sf          = sf
sf >>> SFId          = sf
(SF f1) >>> (SF f2) = ...as before...
```

Absolutely essential that we have a GADT, so the result type can be SF a a

Only well typed because SFId : SF a a

Big speedups [ICFP'05]

Time without optimisations

Time with optimisations

Benchmark	T_S [s]	T_G [s]
1	0.41	0.00
2	0.74	0.22
3	0.45	0.22
4	1.29	0.07
5	1.95	0.08
6	1.48	0.69
7	2.85	0.72

Table 3. Micro benchmark performance. Averages over five runs.

Episode 2

Higher Kinds

Kinds

```
data Maybe a     = Nothing
                 | Just a

data Either a b = Left a
                 | Right b
```

- Which of these user-written type signatures are ok?

```
f1 :: Maybe Int -> Maybe Bool
f2 :: Maybe -> Int
f3 :: Either Int -> Maybe Int
f4 :: Maybe Either -> Int
```

Kinds

```
data Maybe a      = Nothing
                  | Just a

data Either a b = Left a
                  | Right b
```

- Which of these user-written type signatures are ok?

```
f1 :: Maybe Int -> Maybe Bool      -- Yes
f2 :: Maybe -> Int                 -- No
f3 :: Either Int -> Maybe Int      -- No
f4 :: Either Int (Maybe Bool)      -- Yes
f4 :: Maybe Either -> Int          -- No
```

Kinds

```
data Maybe a     = Nothing
                 | Just
```

Just as we type-check user-written terms, so we must kind-check user-written types!

```
f                      Bool     -- Yes
f2                 -> Int       -- No
f3 :: Either Int -> Maybe Int   -- No
f4 :: Either Int (Maybe Bool)   -- Yes
f4 :: Maybe Either -> Int       -- No
```

Kinds

```
data Maybe a     = Nothing | Just a
  -- Maybe :: * -> *

data Either a b = Left a | Right b
  -- Either :: * -> * -> *

-- Built-in definition for (->)
-- (->) :: * -> * -> *
```

* is the kind of types

```
f2 :: Maybe -> Int                    -- No
```

Kind error
(->) requires "*" as its first argument,
but Maybe has kind (* -> *)

Kinds in Haskell

κ ::= *
   | κ -> κ

- Just as
  - **Types** classify **terms**
    eg 3 :: Int, (\x.x+1) :: Int -> Int
  - **Kinds** classify **types**
    eg Int :: *, Maybe :: * -> *, Maybe Int :: *
- Just as
  - **Types** stop you building nonsensical terms
    eg (True + 4)
  - **Kinds** stop you building nonsensical types
    eg (Maybe Maybe)

Reuse via abstraction

```
sum :: [Int] -> Int
sum [] = 0
sum (x:xs) = x + sum xs

product :: [Float] -> Float
product [] = 1
product (x:xs) = x * product xs
```

- Abstract out the common bits

```
foldr :: (a->b->b) -> b -> [a] -> b
foldr k z [] = z
foldr k z (x:xs) = x `k` foldr k z xs

sum = foldr (+) 0
product = foldr (*) 1
```

Reuse via abstraction

Note that we abstract a FUNCTION

```
foldr :: (a->b->b) -> b -> [a] -> b
foldr k z [] = z
foldr k z (x:xs) = x `k` foldr k z xs

sum = foldr (+) 0
product = foldr (*) 1
```

- A first order language does not support abstraction of functions. Sad. So sad.
- The language is "getting in the way"
- Higher order => same language with fewer restrictions

It's the same for types!

```
data RoseTree a = RLeaf a
                | RNode [RoseTree a]
data BinTree a = BLeaf a
               | BNode (Pair (BinTree a))

data Pair a = MkPair a a
```

It's the same for types!

Remove syntactic sugar

```
data RoseTree a = RLeaf a
                | RNode ([] (RoseTree a))
data BinTree a = BLeaf a
               | BNode (Pair (BinTree a))

-- [] :: * -> *            The list constructor
```

means exactly the same as

```
data RoseTree a = RLeaf a
                | RNode [RoseTree a]
```

Kinds in Haskell

```
data Tree f a = Leaf a
              | Node (f (Tree f a))

type RoseTree a = Tree []    a
type BinTree  a = Tree Pair a
type AnnTree  a = Tree AnnPair a

data Pair a    = P a a
data AnnPair a = AP String a a
```

- 'a' stands for a type
- 'f' stands for a type **constructor**

Kinds in Haskell

```
a :: *
f :: * -> *
Tree :: (*->*) -> * -> *
```

```haskell
data Tree f a = Leaf a
              | Node (f (Tree f a))
```

- 'a' stands for a type
- 'f' stands for a type **constructor**

$$\kappa ::= * \mid \kappa \to \kappa$$

- Abstracting over something of kind (*->*) is very useful (cf foldr); same language, fewer restrictions
- You can do this in Haskell (since the beginning), but not in ML, Java, .NET etc

Higher kinds support re-use

```
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b

sequence :: Monad m => [m a] -> m [a]
sequence [] = return []
sequence (a:as) = a >>= \x ->
                  sequence as >>= \xs ->
                  return (x:xs)
```

- Being able to abstract over a higher-kinded 'm' is utterly crucial to code re-use
- We can give a kind to Monad:
  Monad :: (*->*) -> Constraint

EPISODE 3

KIND POLYMORPHISM

But kinds are Too Weedy

```
data T f a = MkT (f a)
```

```
data Maybe a = Nothing | Just a
type T1 = T Maybe Int
```

Maybe :: * -> *

- But is this ok too?

```
data F f = MkF (f Int)
type T2 = T F Maybe
```

F :: (*->*) -> *

- What kind does T have?
  - T :: (* -> *) -> * -> *?
  - T :: ((* -> *) -> *) -> (* -> *) -> *?
- Haskell 98 "defaults" to the first, and hence rejects T2

But kinds are Too Weedy

```
data T f a = MkT (f a)
```

- What kind does T have?
  - T :: (* -> *) -> * -> *?
  - T :: ((* -> *) -> *) -> (* -> *) -> *?
- Haskell 98 "defaults" to the first
- This is Obviously Wrong! We want...

Kind polymorphism

```
data T f a = MkT (f a)
```

- What kind does T have?
  - T :: (* -> *) -> * -> *?
  - T :: ((* -> *) -> *) -> (* -> *) -> *?
- Haskell 98 "defaults" to the first
- This is obviously wrong! We want...

T :: ∀k. (k->*) -> k -> *

Kind polymorphism

```
data T f a = MkT (f a)
```

$T :: \forall k.\ (k \to *) \to k \to *$

Syntax of kinds

$$\kappa ::= * \mid \kappa \to \kappa$$
$$\mid \forall k.\ \kappa$$
$$\mid k$$

Kind polymorphism

```
data T f a = MkT (f a)
```

$T :: \forall k.\ (k \to *) \to k \to *$

A kind

And hence:

$MkT :: \forall k.\ \forall (f : k \to *)\ (a : k).\ f\ a \to T\ f\ a$

A type

So poly-kinded type constructors mean that terms too must be poly-kinded.

Kind inference

- Just as we infer the most general type of a function definition, so we should infer the most general kind of a type definition
- Just like for functions, the type constructor can be used only monomorphically its own RHS.

```
data T f a = MkT (f a)
           | T2 (T Maybe Int)
```

T2 forces T's kind to be (*->*) -> *

Same story for type classes

- Haskell today:

```
data TypeRep = TyCon String
             | TyApp TypeRep TypeRep

class Typeable a where
  typeOf :: a -> TypeRep

instance Typeable Int where
  typeOf _ = TyCon "Int"

instance Typeable a
      => Typeable (Maybe a) where
  typeOf _ = TyApp (TyCon "Maybe")
                   (typeOf (undefined :: a))
```

Same story for type classes

```
instance Typeable a
      => Typeable (Maybe a) where
  typeOf _ = TyApp (TyCon "Maybe")
                   (typeOf (undefined :: a))
```

No!

```
instance (Typeable f, Typeable a)
      => Typeable (f a) where
  typeOf _ = TyApp (typeOf (undefined :: f))
                   (typeOf (undefined :: a))
```

Yes!

But:

- Typeable :: * -> Constraint, but f :: *->*
- (undefined :: f) makes no sense, since f :: *->*

What we want: a poly-kinded class

```
class Typeable a where
  typeOf :: p a -> TypeRep

data Proxy a
```

- Typeable :: ∀**k**. k -> Constraint
  typeOf :: ∀k ∀a:k. Typeable a =>
  ∀(p:k->*). p a -> TypeRep
  Proxy :: ∀**k**. k -> *

```
instance (Typeable f, Typeable a)
      => Typeable (f a) where
  typeOf _
    = TyApp (typeOf (undefined :: Proxy f))
            (typeOf (undefined :: Proxy a))
```

Everything works out smoothly

- Type inference becomes a bit more tricky - but not much.
  - Instantiate f :: forall k. forall (a:k). tau with a fresh **kind** unification variable for k, and a fresh **type** unification variable for a
  - When unifying (a ~ some-type), unify a's kind with some-type's kind.
- Intermediate language (System F)
  - Already has **type** abstraction and application
  - Add **kind** abstraction and application

EPISODE 4

PROMOTING DATA TYPES

Embarrassment

```
data Vec n a where
  Vnil  :: Vec Zero a
  Vcons :: a -> Vec n a -> Vec (Succ n) a
```

- What is Zero, Succ? Kind of Vec?

```
data Zero
data Succ a
-- Vec :: * -> * -> *
```

- Yuk! Nothing to stop you writing stupid types:
  f :: Vec Int a -> Vec Bool a

In short

```
data Zero
data Succ a
-- Vec :: * -> * -> *
```

- Haskell is a strongly typed language
- But programming at the type level is entirely un-typed – or rather uni-typed, with one type, *.
- How embarrassing is that?

What we want: typed type-level programming

```
datakind Nat = Zero | Succ Nat

data Vec n a where
  Vnil  :: Vec Zero a
  Vcons :: a -> Vec n a -> Vec (Succ n) a
```

Vec :: Nat -> * -> *

- Now the type (Vec Int a) is ill-kinded; hurrah
- Nat is a **kind**, here introduced by 'datakind'

What we have implemented

```
data Nat = Zero | Succ Nat

data Vec n a where
  Vnil  :: Vec Zero a
  Vcons :: a -> Vec n a -> Vec (Succ n) a
```

Vec :: Nat -> * -> *

- Nat is an ordinary **type**, but it is automatically promoted to be a **kind as well**
- Its constuctors are promoted to be (uninhabited) types
- Mostly: simple, easy

Works for type functions of course

```
data Nat = Zero | Succ Nat

type family Add (a::Nat) (b::Nat) :: Nat

type instance Add Z        n = n
type instance Add (Succ n) m = Succ (Add n m)
```

Add :: Nat -> Nat -> Nat

Details

Type constructor

Data constructor

```
data Foo = Foo Int

f :: T Foo -> Int
```

Which?

- Where there is only one Foo (type or data constructor) use that
- If both Foo's are in scope, "Foo" in a type means the type constructor (backward compaitible)
- If both Foo's are in scope, 'Foo means the data constructor

Details

- Which data types are promoted?

```
data T where
  MkT :: a -> (a->Int) -> T

data S where
  MkS :: S Int
```

Existentials?

GADTs?

- Keep it simple: only simple, vanilla, types with kinds of form T :: * -> * -> … -> *
- Avoids the need for
  - A sort system (to classify kinds!)
  - Kind equalities (for GADTs)

Summary of kinds

- Take lessons from term :: type and apply them to type :: kind
  - Polymorphism
  - Constraint kind
  - Data types
- Hopefully: no new concepts. Re-use programmers intuitions abou how typing works, one level up.
- Fits smoothly into the IL
- Result: world peace