Classes, Jim, but not as we know them
Type classes in Haskell

Simon Peyton Jones (Microsoft Research)

ECOOP 2009

Origins

Haskell is 20 this year

The late 1979s, early 1980s

Pure functional programming: recursion, pattern matching, comprehensions etc etc (ML, SASL, KRC, Hope, Id)

Lazy functional programming (Friedman, Wise, Henderson, Morris, Turner)

Lisp machines (Symbolics, LMI)

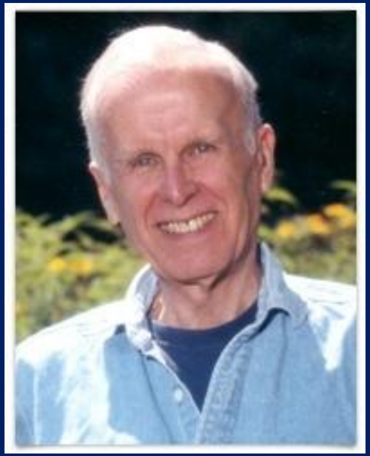
Lambda the Ultimate (Steele, Sussman)

SK combinators, graph reduction (Turner)

Dataflow architectures (Dennis, Arvind et al)

Backus 1978

Can programming be liberated from the von Neumann style?



John Backus Dec 1924 – Mar 2007

The 1980s

Functio
recurs
co
(ML

on,

Dat

tors,
ction
)

FP is respectable
(as well as cool)

Go forth and design new languages
and new computers
and rule the world

Result

Chaos

Many, many bright young things

Many conferences
(birth of FPCA, LFP)

Many languages
(Miranda, LML, Orwell, Ponder, Alfl, Clean)

Many compilers

Many architectures
(mostly doomed)

Crystalisation

FPCA, Sept 1987: initial meeting.
A dozen lazy functional programmers, wanting to agree on a common language.

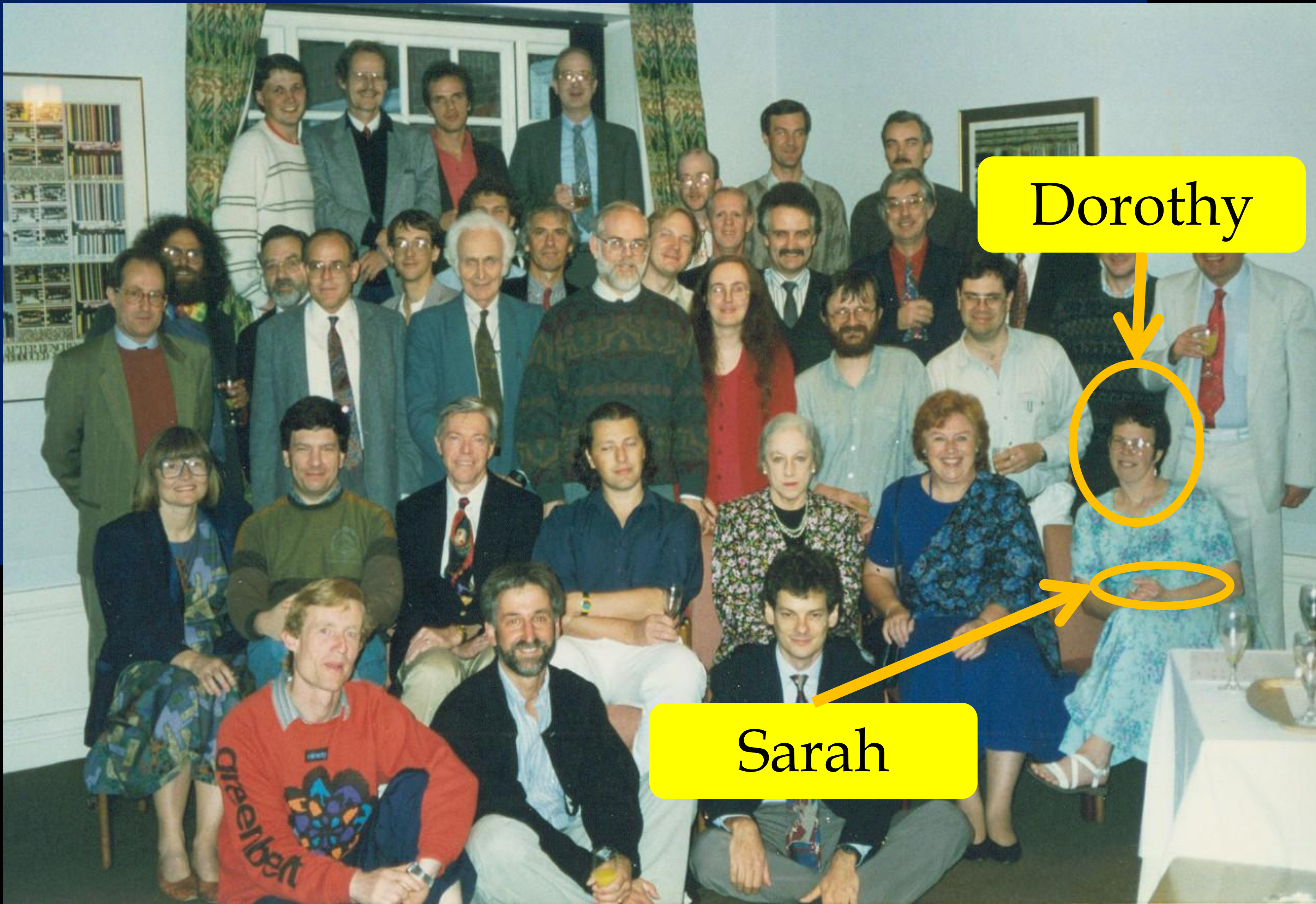
- Suitable for teaching, research, and application
- Formally-described syntax and semantics
- Freely available
- Embody the apparent consensus of ideas
- Reduce unnecessary diversity

Absolutely no clue how much work we were taking on

Led to...a succession of face-to-face meetings

WG2.8 June 1992

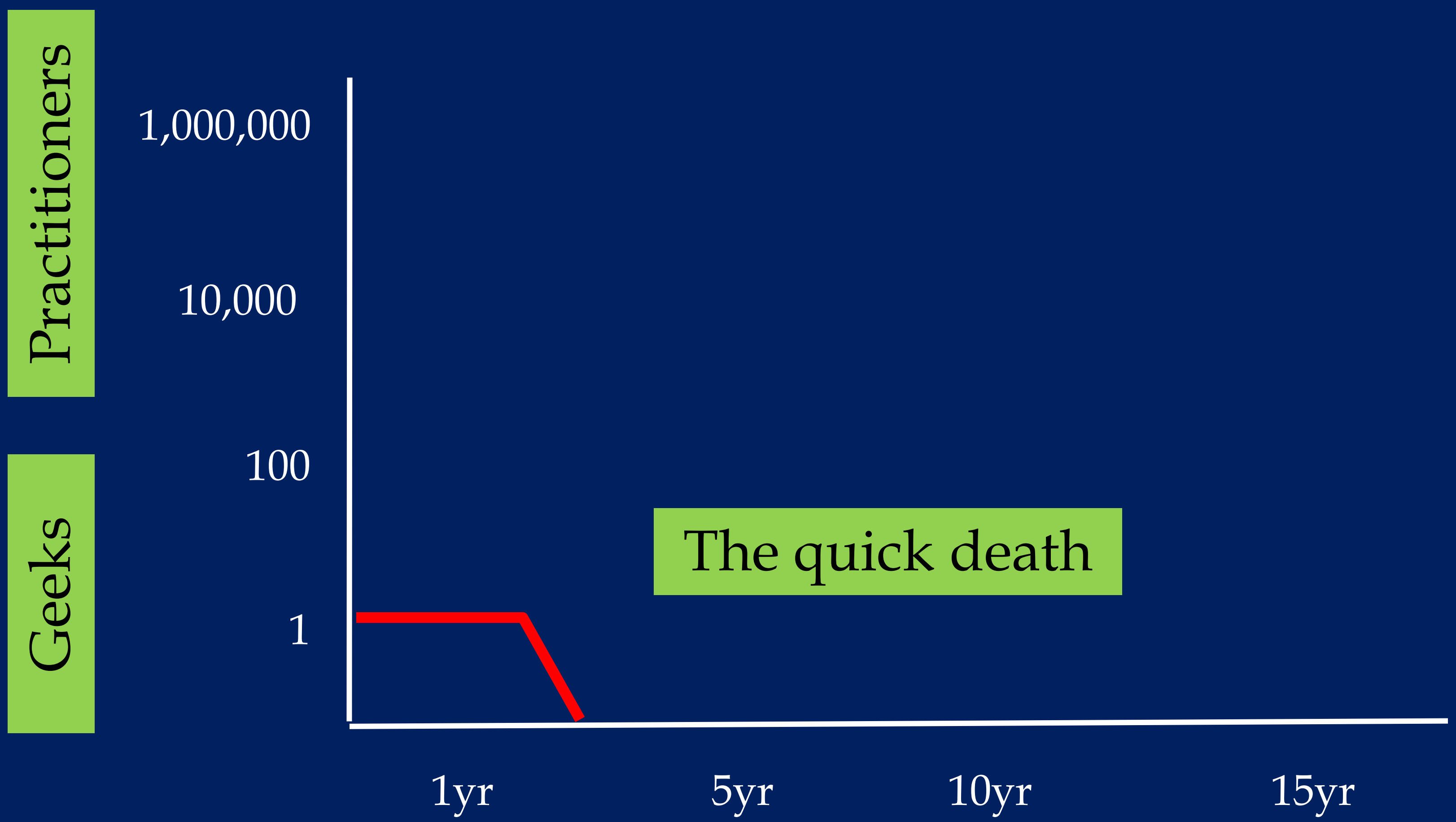
WG2.8 June 1992



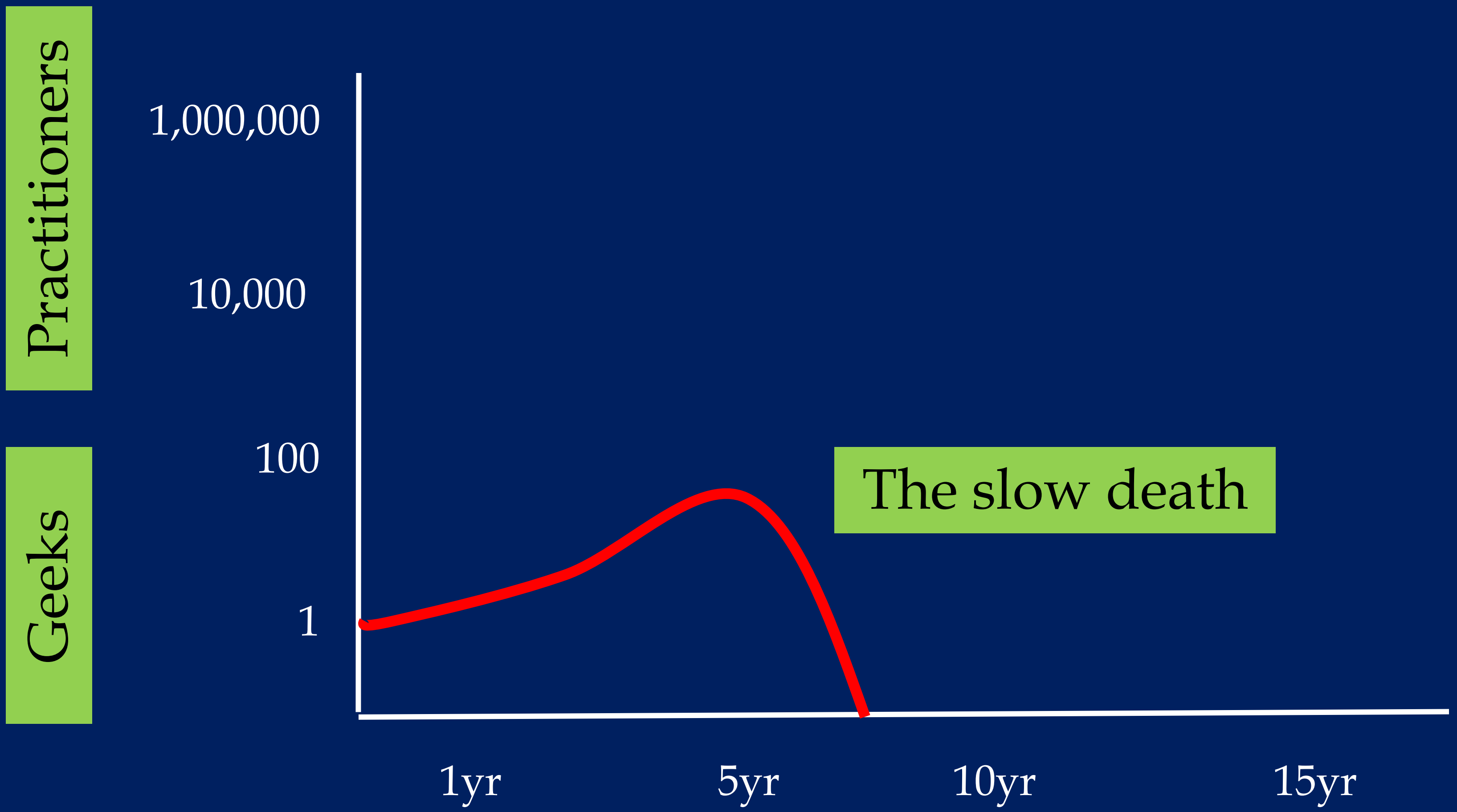
Sarah (b. 1993)


Haskell the cat (b. 2002)

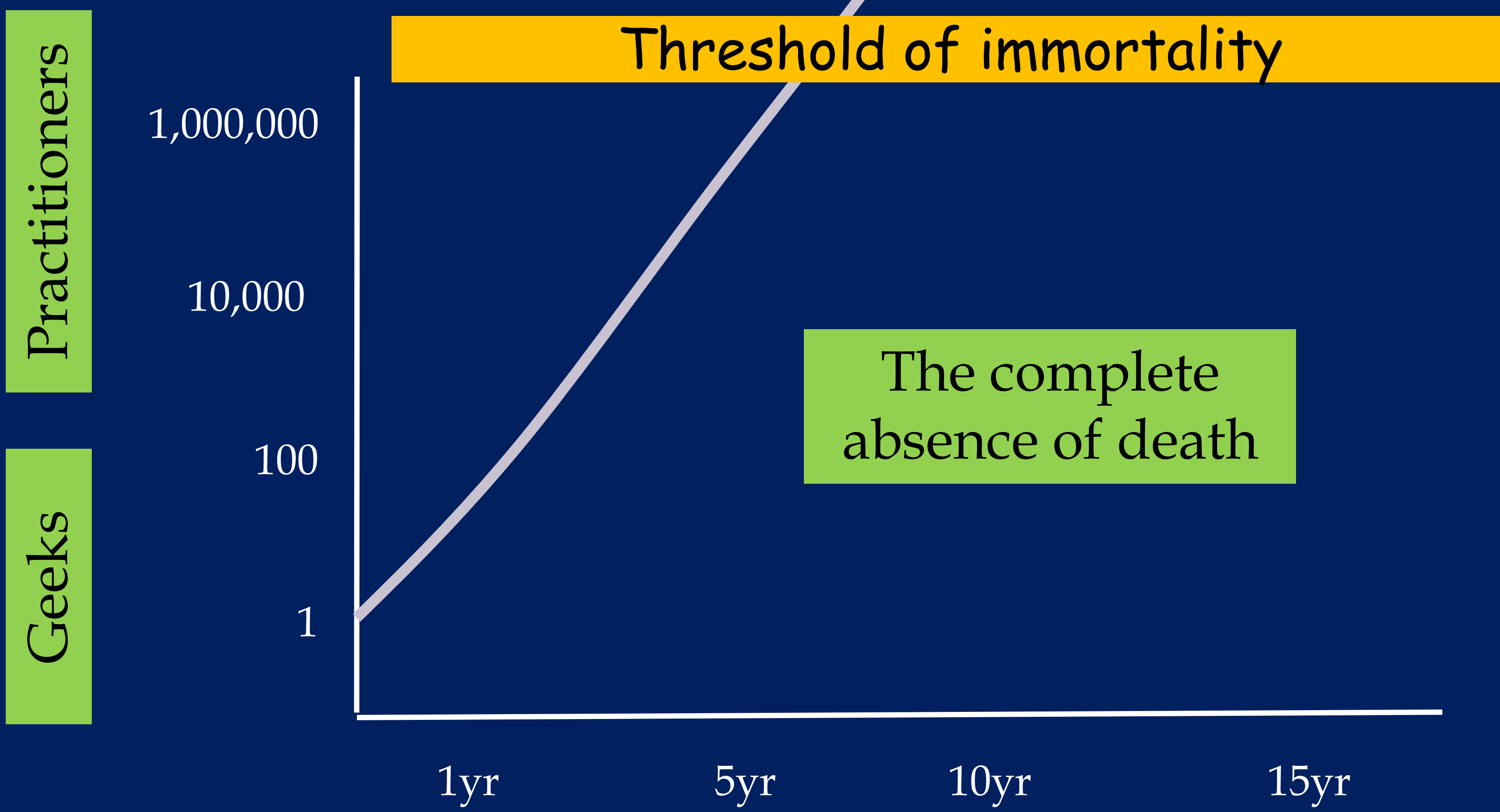
Most new programming languages



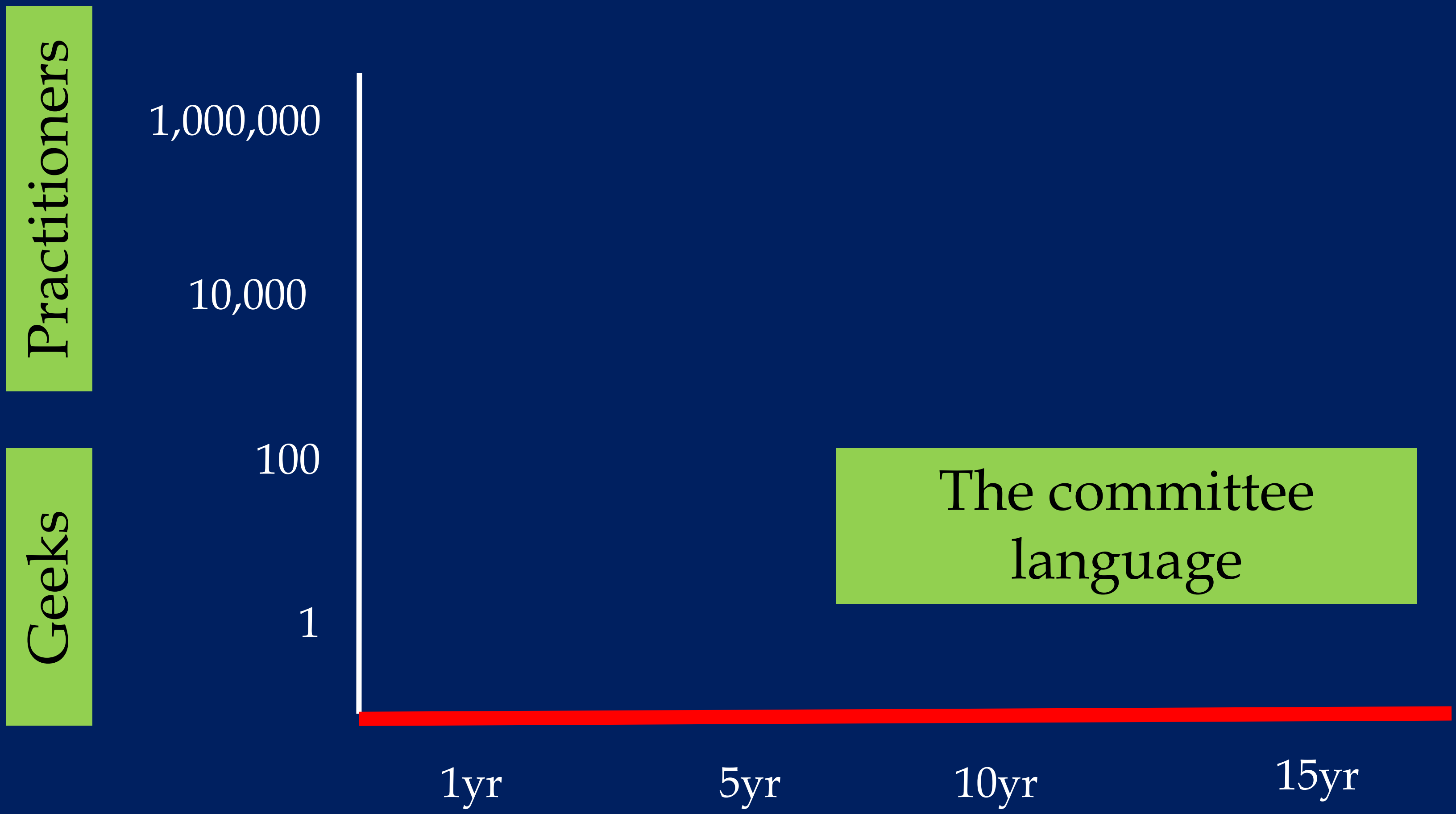
Successful research languages



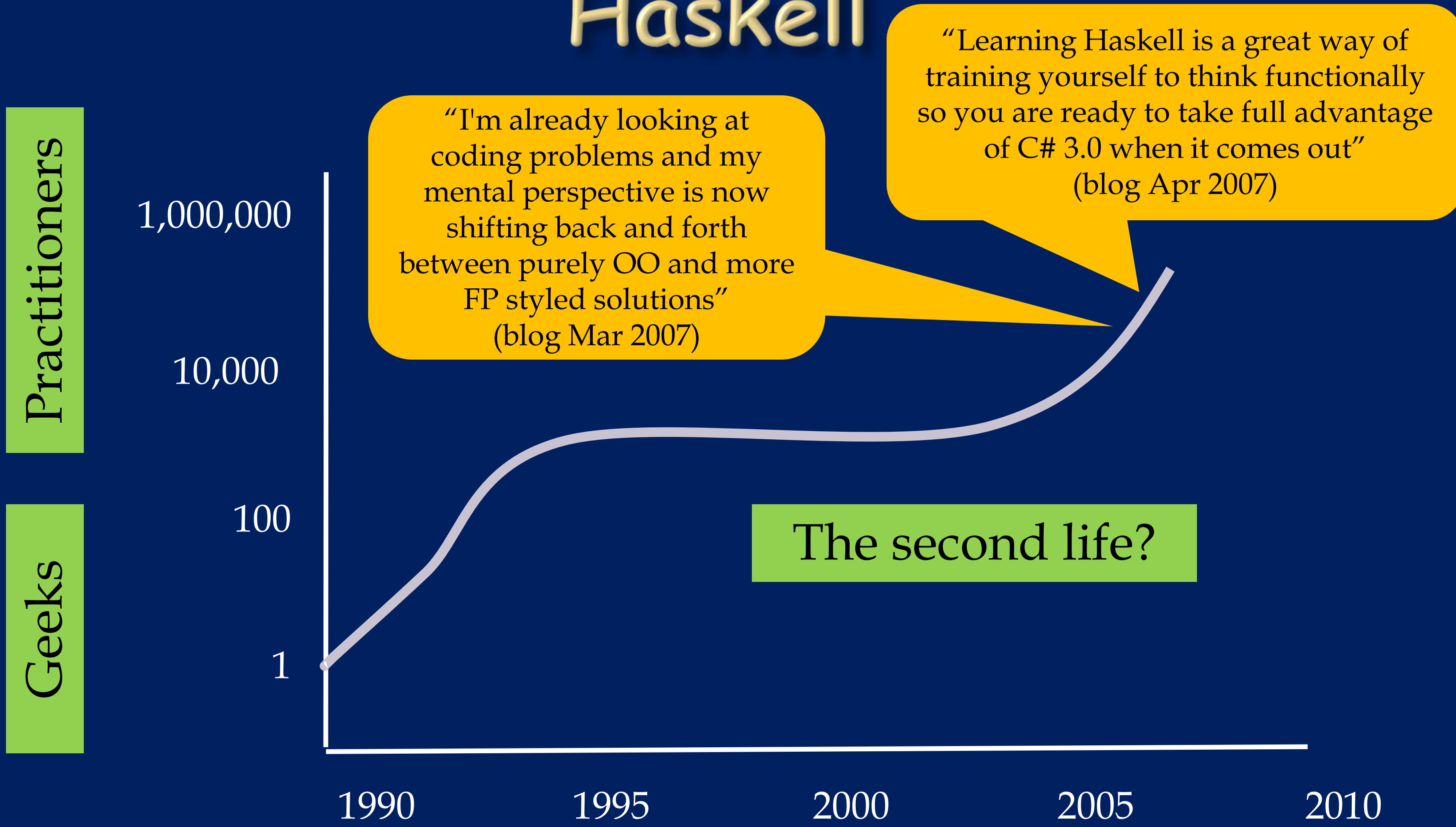
C++, Java, Perl, Ruby



Committee languages



Haskell



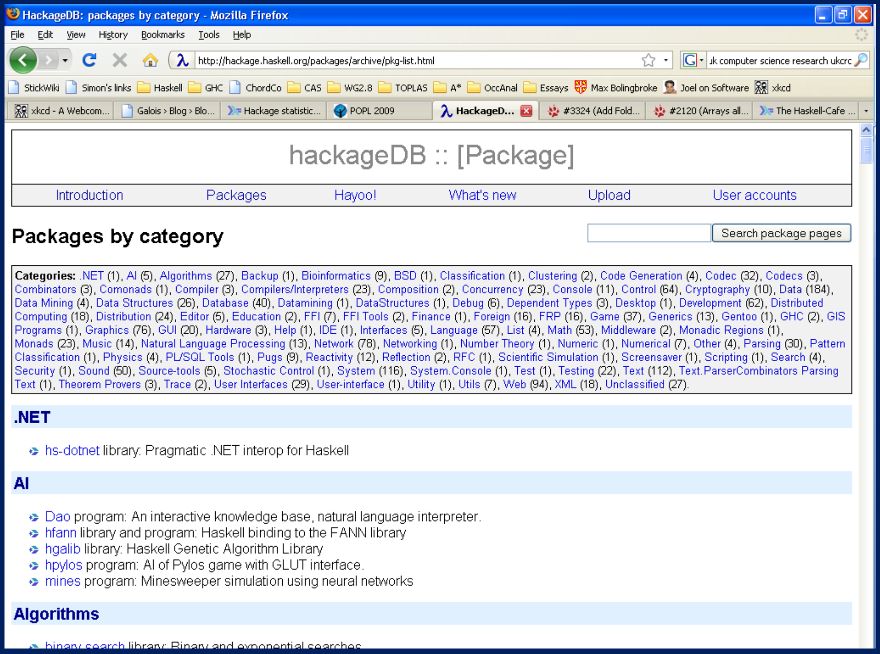
"Learning Haskell is a great way of training yourself to think functionally so you are ready to take full advantage of C# 3.0 when it comes out"
(blog Apr 2007)
"I'm already looking at coding problems and my mental perspective is now shifting back and forth between purely OO and more FP styled solutions"
(blog Mar 2007)
Practitioners
Geeks
1,000,000
10,000
100
1
The second life?
1990
1995
2000
2005
2010

Mobilising the community

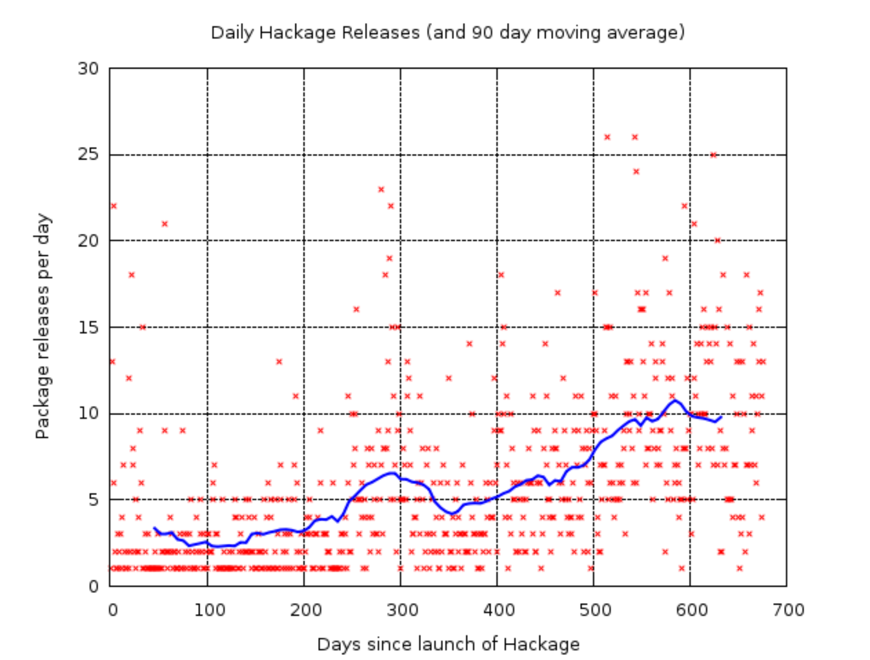
- Package = unit of distribution
- **Cabal**: simple tool to install package and all its dependencies

```
bash$ cabal install pressburger
```

- **Hackage**: central repository of packages, with open upload policy



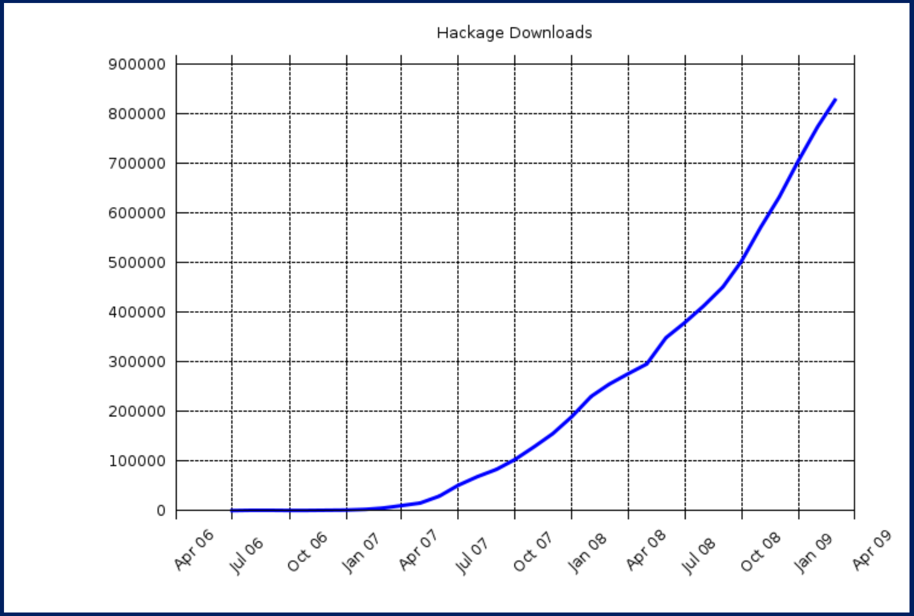
Result: staggering



Package uploads

Running at 300/month

Over 1350 packages



Package downloads

heading for 1 million

2 years

Type classes

Haskell in one slide

Type signature

Higher order

Polymorphism (works for any type a)

```
filter :: (a->Bool) -> [a] -> [a]
filter p [] = []
filter p (x:xs)
      | p x        = x : filter p xs
      | otherwise = filter p xs
```

Functions defined by pattern matching

Guards distinguish sub-cases

Problem

```
member :: a -> [a] -> Bool
member x []                    = False
member x (y:ys) | x==y         = True
                | otherwise = member x ys
```

Test for equality

- Can this really work **FOR ANY** type a?
- E.g. what about functions?
  member negate [increment, \x. 0-x]

Similar problems

- Similar problems
  - sort :: [a] -> [a]
  - (+) :: a -> a -> a
  - show :: a -> String
  - serialise :: a -> BitString
  - hash :: a -> Int
- Unsatisfactory solutions
  - Local choice
  - Provide equality, serialisation for everything, with runtime error for (say) functions

Unsatisfactory solutions

- Local choice
  - Write (a + b) to mean (a `plusFloat` b) or (a `plusInt` b) depending on type of a,b
  - Loss of abstraction; eg member is monomorphic
- Provide equality, serialisation for everything, with runtime error for (say) functions
  - Not extensible: just a baked-in solution for certain baked-in functions
  - Run-time errors

Type classes

Works for any type 'a', **provided 'a' is an instance of class Num**

```
square :: Num a => a -> a
square x = x*x
```

Similarly:

```
sort      :: Ord a  => [a] -> [a]
serialise :: Show a => a -> String
member    :: Eq a   => a -> [a] -> Bool
```

Type classes

Works for any type 'n' that supports the Num operations

```
square :: Num n  => n -> n
square x = x*x
```

FORGET all you know about OO classes!

```
class Num a where
  (+)    :: a -> a -> a
  (*)    :: a -> a -> a
  negate :: a -> a
  ...etc..
```

The class declaration says what the Num operations are

```
instance Num Int where
  a + b    = plusInt a b
  a * b    = mulInt a b
  negate a = negInt a
  ...etc..
```

An instance declaration for a type T says how the Num operations are implemented on T's

```
plusInt :: Int -> Int -> Int
mulInt  :: Int -> Int -> Int
etc, defined as primitives
```

How type classes work

When you write this...

```
square :: Num n => n -> n
square x = x*x
```

...the compiler generates this

```
square :: Num n -> n -> n
square d x = (*) d x x
```

The "**Num n =>**" turns into an extra value argument to the function.
It is a value of data type **Num n**

A value of type (Num T) is a vector of the Num operations for type T

How type classes work

When you write this...

```
square :: Num n => n -> n
square x = x*x
```

```
class Num a where
  (+)    :: a -> a -> a
  (*)    :: a -> a -> a
  negate :: a -> a
  ...etc..
```

...the compiler generates this

```
square :: Num n -> n -> n
square d x = (*) d x x
```

```
data Num a
  = MkNum (a->a->a)
          (a->a->a)
          (a->a)
          ...etc...

(*) :: Num a -> a -> a -> a
(*) (MkNum _ m _ ...) = m
```

The class decl translates to:

- A data type decl for Num
- A selector function for each class operation

A value of type (Num T) is a vector of the Num operations for type T

How type classes work

When you write this...

```
square :: Num n => n -> n
square x = x*x
```

...the compiler generates this

```
square :: Num n -> n -> n
square d x = (*) d x x
```

```
instance Num Int where
  a + b    = plusInt a b
  a * b    = mulInt a b
  negate a = negInt a
  ...etc..
```

```
dNumInt :: Num Int
dNumInt = MkNum plusInt
                mulInt
                negInt
                ...
```

An instance decl for type T translates to a value declaration for the Num dictionary for T

A value of type (Num T) is a vector of the Num operations for type T

All this scales up nicely

- You can build big overloaded **functions** by calling smaller overloaded **functions**

```
sumSq :: Num n => n -> n -> n
sumSq x y = square x + square y
```

```
sumSq :: Num n -> n -> n -> n
sumSq d x y = (+) d (square d x)
                    (square d y)
```

Extract addition operation from d

Pass on d to square

All this scales up nicely

- You can build big **instances** by building on smaller **instances**

```
class Eq a where
  (==) :: a -> a -> Bool

instance Eq a => Eq [a] where
  (==) []     []     = True
  (==) (x:xs) (y:ys) = x==y && xs == ys
  (==) _      _      = False
```

```
data Eq = MkEq (a->a->Bool)
(==) (MkEq eq) = eq

dEqList :: Eq a -> Eq [a]
dEqList d = MkEq eql
  where
    eql []     []     = True
    eql (x:xs) (y:ys) = (==) d x y && eql xs ys
    eql _      _      = False
```

Overloaded constants

```
class Num a where
  (+) :: a -> a -> a
  (-) :: a -> a -> a
  fromInteger :: Integer -> a
  ....

inc :: Num a => a -> a
inc x = x + 1
```

Even literals are overloaded

"1" means "fromInteger 1"

```
inc :: Num a -> a -> a
inc d x = (+) d x (fromInteger d 1)
```

Quickcheck

```
propRev :: [Int] -> Bool
propRev xs = reverse (reverse xs) == xs

propRevApp :: [Int] -> [Int] -> Bool
propRevApp xs ys = reverse (xs++ys) ==
                   reverse ys ++ reverse xs
```

Quickcheck (which is just a Haskell 98 library)

- Works out how many arguments
- Generates suitable test data
- Runs tests

```
ghci> quickCheck propRev
OK: passed 100 tests

ghci> quickCheck propRevApp
OK: passed 100 tests
```

A completely different example: Quickcheck

```
quickCheck :: Testable a => a -> IO ()

class Testable a where
  test :: a -> RandSupply -> Bool

class Arbitrary a where
  arby :: RandSupply -> a

instance Testable Bool where
  test b r = b

instance (Arbitrary a, Testable b)
      => Testable (a->b) where
  test f r = test (f (arby r1)) r2
           where (r1,r2) = split r
```

```
split :: RandSupply -> (RandSupply, RandSupply)
```

A completely different example: Quickcheck

```
propRev :: [Int] -> Bool
```

```
test propRev r

= test (propRev (arby r1)) r2
where (r1,r2) = split r

= propRev (arby r1)
```

Using instance for (->)

Using instance for Bool

Type classes have proved extraordinarily convenient in practice

- Equality, ordering, serialisation
- Numerical operations. Even numeric constants are overloaded
- Monadic operations

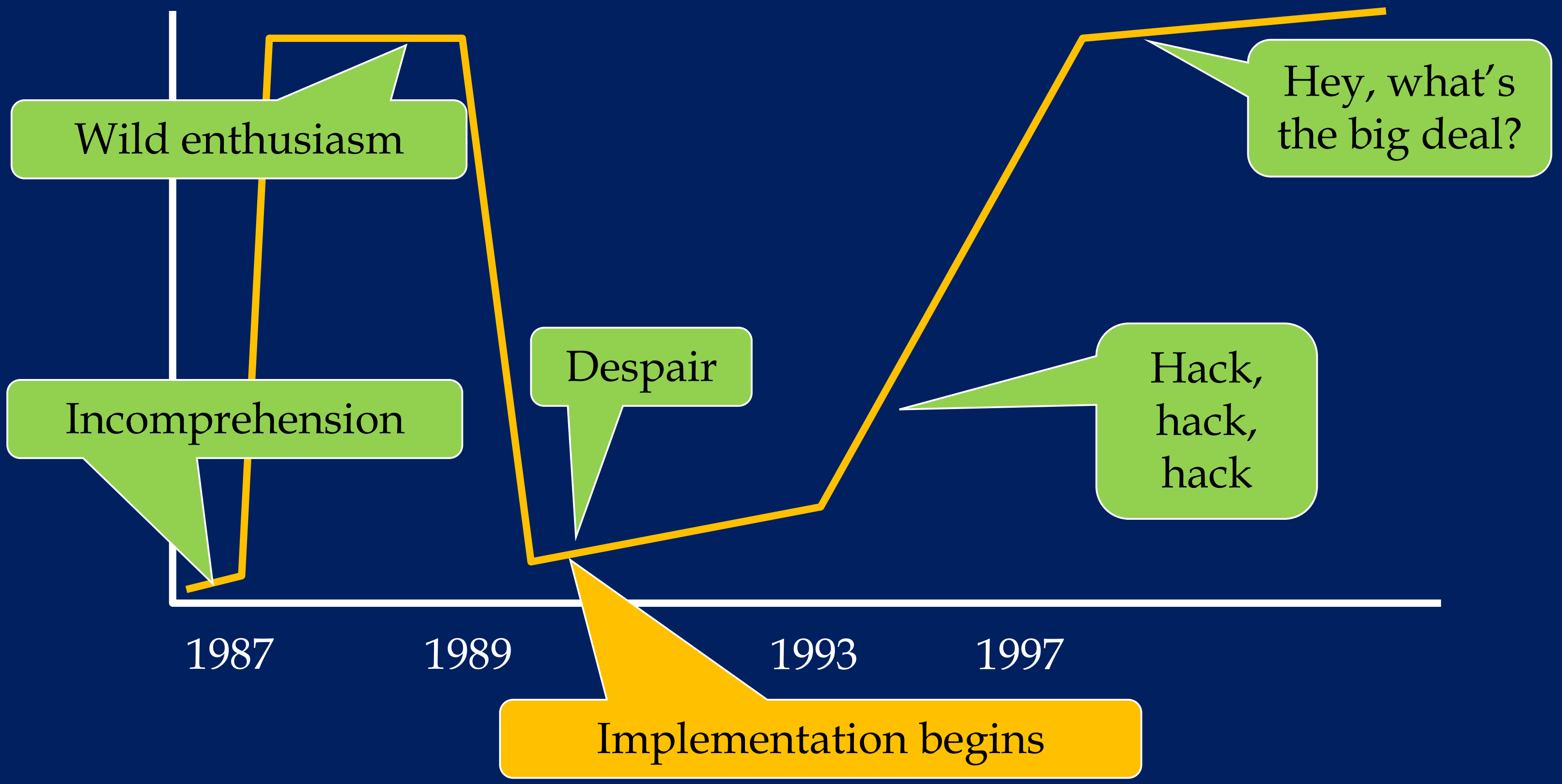
```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
```

- And on and on....time-varying values, pretty-printing, collections, reflection, generic programming, marshalling, monad transformers....

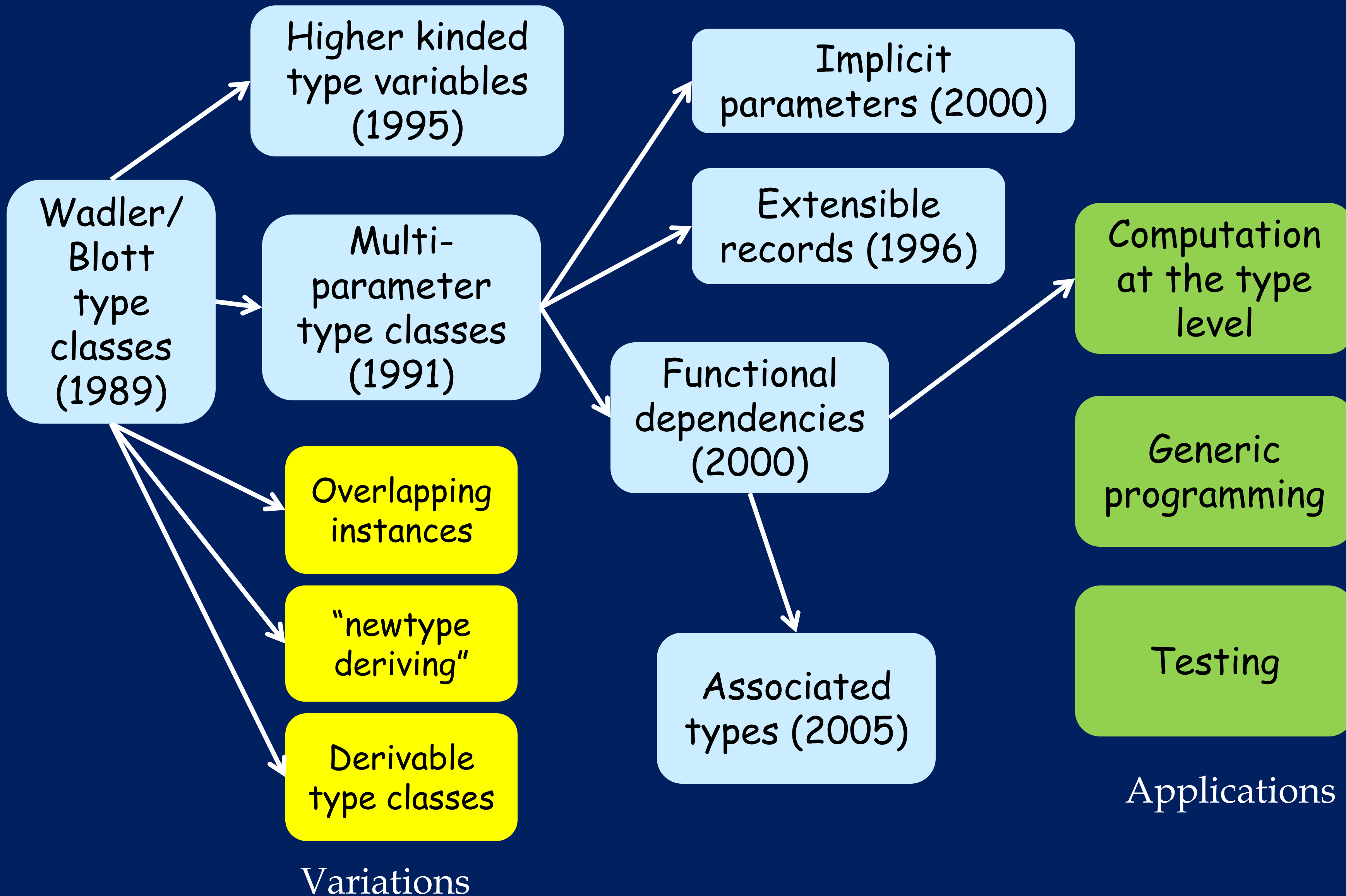
Note the higher-kinded type variable, m

Type classes over time

- Type classes are the most unusual feature of Haskell's type system



Type-class fertility



Type classes and object-oriented programming

1. Type-based dispatch, not value-based dispatch

Type-based dispatch

- A bit like OOP, except that method suite passed separately?

```
class Show where
  show :: a -> String

f :: Show a => a ->
  ...
```

- No!! Type classes implement **type-based dispatch**, not **value-based dispatch**

Type-based dispatch

```
class Read a where
  read :: String -> a

class Num a where
  (+) :: a -> a -> a
  fromInteger :: Integer -> a
```

```
read2 :: (Read a, Num a) => String -> a
read2 s = read s + 2
```

```
read2 dr dn s = (+) dn (read dr s)
                       (fromInteger dn 2)
```

- The overloaded value is *returned by* `read2`, not passed to it.
- It is the dictionaries (and type) that are passed as argument to `read2`

Type based dispatch

So the links to **intensional polymorphism** are closer than the links to *OOP*.

The dictionary is like a proxy for the (interesting aspects of) the type argument of a polymorphic function.

Intensional polymorphism

```
f :: forall a. a -> Int
f t (x::t) = ...typecase t...

f :: forall a. C a => a -> Int
f x = ...(call method of C)...
```

Haskell

Type classes and object-oriented programming

1. Type-based dispatch, not value-based dispatch
2. Haskell "class" ~ OO "interface"

Haskell "class" ~ OO "interface"

A Haskell class is more like a Java **interface** than a Java **class**: it says what operations the type must support.

```
class Show a where
  show :: a -> String

f :: Show a => a -> ...
```

```
interface Showable {
  String show();
}

class Blah {
  f( Showable x ) {
      ...x.show()...
} }
```

Haskell "class" ~ OO "interface"

- No problem with multiple constraints:

```
f :: (Num a, Show a)
  => a -> ...
```

```
class Blah {
  f( ??? x ) {
      ...x.show()...
} }
```

- Existing types can retroactively be made instances of new type classes (e.g. introduce new Wibble class, make existing types an instance of it)

```
class Wibble a where
  wib :: a -> Bool

instance Wibble Int where
  wib n = n+1
```

```
interface Wibble {
  bool wib()
}

...does Int support
  Wibble?....
```

Type classes and object-oriented programming

1. Type-based dispatch, not value-based dispatch
2. Haskell “class” ~ OO “interface”
3. Generics (i.e. parametric polymorphism) , not subtyping

Generics, not subtyping

- Haskell has no sub-typing

```
data Tree = Leaf | Branch Tree Tree

f :: Tree -> Int
f t = ...
```

f's argument must be (**exactly**) a Tree

- Ability to act on argument of various types achieved via type classes:

```
square :: (Num a) => a -> a
square x = x*x
```

Works for any type supporting the Num interface

Generics, not subtyping

- Means that in Haskell you must anticipate the need to act on arguments of various types

```
f :: Tree -> Int
vs
f' :: Treelike a => a -> Int
```

(in OO you can retroactively sub-class Tree)

No subtyping: inference

- Type annotations:
  - Implicit = the type of a fresh binder is inferred

```
f x = ...
```

  - Explicit = each binder is given a type at its binding site

```
void f( int x ) { ... }
```

- Cultural heritage:
  - Haskell: everything implicit
    type annotations occasionally needed
  - Java: everything explicit;
    type inference occasionally possible

No subtyping : inference

- Type annotations:
  - Implicit = the type of a fresh binder is inferred

```
f x = ...
```

  - Explicit = each binder is given a type at its binding site

```
void f( int x ) { ... }
```

- Reason:
  - Generics alone => type engine generates **equality constraints**, which it can solve
  - Subtyping => type engine generates **subtyping constraints**, which it cannot solve (uniquely)

No subtyping : binary methods

```
class Eq a where
   (==) :: a -> a -> Bool

instance Eq a => Eq [a] where
   (==) []     []     = True
   (==) (x:xs) (y:ys) = x==y && xs == ys
   (==) _      _      = False
```

Here we know that the two arguments have exactly the same type

No subtyping : variance

- In Java (ish):

Any sub-type of Numable

Any super-type of Numable

```
inc :: Numable -> Numable
```

- In Haskell:

Result has precisely same type as argument

```
inc :: Num a => a -> a
```

- Compare...

```
x::Float

...(x.inc)...
```

Numable

```
x::Float

...(inc x)...
```

Float

Variance in OOP

- In practice, because many operations work by side effect, result contra-variance doesn't matter too much

```
x.setColour(Blue);
x.setPosition(3,4);
```

None of this changes x's type

- In a purely-functional world, where setColour, setPosition return a new x, result contra-variance might be much more important

Variance in OOP

- Nevertheless, Java and C# both (now) support **constrained generics**

```
class Blah {
  <A extends Numable> A inc( A x)
}
```

- Very like `inc :: Num a => a -> a`

Variance

- Variance simply does not arise in Haskell.
- OOP: must embrace variance
  - Side effects => invariance
  - Generics: type parameters are co/contra/invariant (Java wildcards, C#4.0 variance annotations)
  - Interaction with higher kinds?

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

(Scala is about to remove them!)

- Need constraint polymorphism anyway!

Open question

In a language with

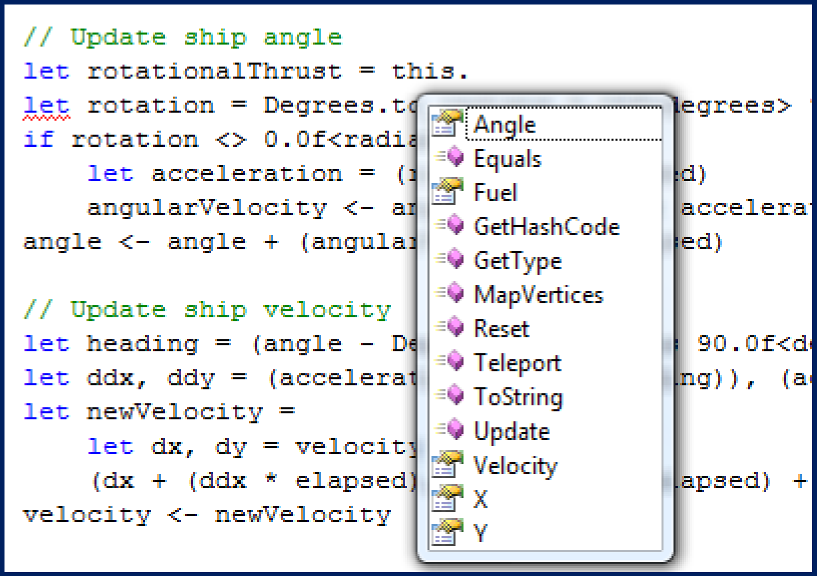
- Generics
- Constrained polymorphism

do you need subtyping too?

What I envy about OOP

What I envy about OOP

- The power of the dot
  - IDE magic
  - overload short function names



- That is:

> Use the type of the first (self) argument to
> (a)"x.": display candidate functions
> (b)"x.reset": fix which "reset" you mean

- (Yes there is more: use argument syntax to further narrow which function you mean.)

What I envy about OOP

- Curiously, this is not specific to OOP, or to sub-typing, or to dynamic dispatch
- Obvious thought: steal this idea and add this to Haskell

```
module M where
  import Button -- reset :: Button -> IO ()
  import Canvas -- reset :: Canvas -> IO ()

  fluggle :: Button -> ...
  fluggle b = ...(b.reset)...
```

Simulating objects

- OOP lets you have a collection of heterogeneous objects

```
void f( Shape[] x );

a::Circle
b::Rectangle

....f (new Shape[] {a, b})...
```

Simulating objects

```
void f( Shape[] x );

a::Circle
b::Rectangle

....f (new Shape[] {a, b})...
```

- You can encode this in Haskell, although it is slightly clumsy

```
data Shape where
   MkShape :: Shapely a => a -> Shape

a :: Circle
b :: Rectangle

....(f [MkShape a, MkShape b])...
```

Reflection, generic programming

- The ability to make run-time type tests is hugely important in OOP.
- We have (eventually) figured out to do this in Haskell:

```
cast :: (Typeable a, Typeable b) => a -> Maybe b
```

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

instance Typeable Bool where
  typeOf _ = MkTypeRep "Bool" []

instance Typeable a => Typeable [a] where
  typeOf xs = MkTypeRep "List" [typeOf (head xs) ]
```

New developments in type classes

Generalising Num

```
plusInt    :: Int -> Int -> Int
plusFloat  :: Float -> Float -> Float
intToFloat :: Int -> Float
```

```
class GNum a b where
  (+) :: a -> b -> ???

instance GNum Int Int where
  (+) x y = plusInt x y

instance GNum Int Float where
  (+) x y = plusFloat (intToFloat x) y

test1 = (4::Int) + (5::Int)
test2 = (4::Int) + (5::Float)
```

Generalising Num

```
class GNum a b where
   (+) :: a -> b -> ???
```

- Result type of (+) is a **function of the argument types**

```
class GNum a b where
   type SumTy a b :: *
   (+) :: a -> b -> SumTy a b
```

SumTy is an associated type of class GNum

- Each method gets a type signature
- Each associated type gets a kind signature

Generalising Num

```
class GNum a b where
  type SumTy a b :: *
  (+) :: a -> b -> SumTy a b
```

- Each instance declaration gives a "witness" for SumTy, matching the kind signature

```
instance GNum Int Int where
  type SumTy Int Int = Int
  (+) x y = plusInt x y

instance GNum Int Float where
  type SumTy Int Float = Float
  (+) x y = plusFloat (intToFloat x) y
```

Type functions

```
class GNum a b where
  type SumTy a b :: *
instance GNum Int Int where
  type SumTy Int Int = Int
instance GNum Int Float where
  type SumTy Int Float = Float
```

- SumTy is a type-level function
- The type checker simply rewrites
  - SumTy Int Int --> Int
  - SumTy Int Float --> Float

  whenever it can
- But (SumTy t1 t2) is still a perfectly good type, even if it can't be rewritten. For example:

```
data T a b = MkT a b (SumTy a b)
```

Type functions...

- Inspired by associated types from OOP
- Fit beautifully with type classes
- Push the type system a little closer to dependent types, but not too close!
- Generalise "functional dependencies"
- ...still developing...

Conclusions

- It's a complicated world.
- Rejoice in diversity. Learn from the competition.
- What can Haskell learn from OOP?
  - The power of the dot (IDE, name space control)
- What can OOP learn from Haskell?
  - The big question for me is: once we have wholeheartedly adopted generics, do we still really need subtyping?

Backup slides about type functions

- See paper “Fun with type functions” [2009] on Simon PJ’s home page

Optimising data structures

- Consider a finite map, mapping **keys** to **values**
- Goal: the data representation of the map depends on the type of the key
  - Boolean key: store two values (for F,T resp)
  - Int key: use a balanced tree
  - Pair key (x,y): map x to a finite map from y to value; ie use a trie!
- Cannot do this in Haskell...a good program that the type checker rejects

Optimising data structures

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

```
class Key k where
  data Map k :: * -> *
  empty  :: Map k v
  lookup :: k -> Map k v -> Maybe v
  ...insert, union, etc....
```

Map is **indexed** by k, but **parametric** in its second argument

Optimising data structures

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

```
class Key k where
  data Map k :: * -> *
  empty  :: Map k v
  lookup :: k -> Map k v -> Maybe v
  ...insert, union, etc....

instance Key Bool where
  data Map Bool v = MB (Maybe v)  (Maybe v)
  empty = MB Nothing Nothing
  lookup True  (MB _ mt) = mt
  lookup False (MB mf _) = mf
```

Optional value for False

Optional value for True

Optimising data structures

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

```
class Key k where
  data Map k :: * -> *
  empty  :: Map k v
  lookup :: k -> Map k v -> Maybe v
  ...insert, union, etc....

instance (Key a, Key b) => Key (a,b) where
  data Map (a,b) v = MP (Map a (Map b v))
  empty = MP empty
  lookup (ka,kb) (MP m) = case lookup ka m of
                            Nothing -> Nothing
                            Just m2 -> lookup kb m2
```

Two-level map

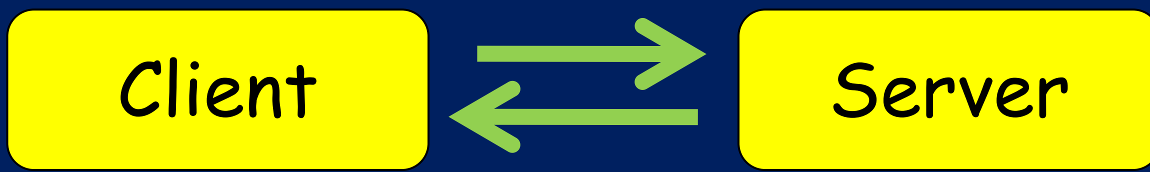
Two-level lookup

See paper for lists as keys: arbitrary depth tries

Optimising data structures

- Goal: the data representation of the map depends on the type of the key
  - Boolean key: SUM
  - Pair key (x,y): PRODUCT
  - List key [x]: SUM of PRODUCT + RECURSION
- Easy to extend to other types at will

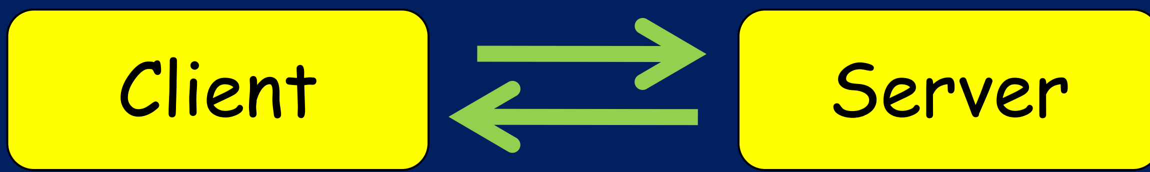
Baby session types (BST)



- addServer :: In Int (In Int (Out Int End))
  addClient   :: Out Int (Out Int (In Int End))
- Type of the process expresses its protocol
- Client and server should have dual protocols:

```
run addServer addClient          -- OK!
run addServer addServer          -- BAD!
```

Baby session types



- addServer :: In Int (In Int (Out Int End))
  addClient :: Out Int (Out Int (In Int End))

```
data In v p  = In (v -> p)
data Out v p = Out v p
data End     = End
```

NB punning

Baby session types

```
data In v p  = In (v -> p)
data Out v p = Out v p
data End     = End
```

```
addServer :: In Int (In Int (Out Int End))
addServer = In (\x -> In (\y ->
              Out (x + y) End))
```

- Nothing fancy here
- addClient is similar

But what about run???

```
run :: ??? -> ??? -> End
```

A process

A co-process

```
class Process p where
  type Co p
  run :: p -> Co p -> End
```

- Same deal as before: `Co` is a type-level function that transforms a process type into its dual

Implementing run

```
class Process p where
  type Co p
  run :: p -> Co p -> End
```

```
data In v p  = In (v -> p)
data Out v p = Out v p
data End     = End
```

```
instance Process p => Process (In v p) where
  type Co (In v p) = Out v (Co p)
  run (In vp) (Out v p) = run (vp v) p

instance Process p => Process (Out v p) where
  type Co (Out v p) = In v (Co p)
  run (Out v p) (In vp) = run p (vp v)
```

Just the obvious thing really