Haskell and Erlang Growing up together

Simon Peyton Jones, Microsoft Research

Haskell and Erlang

- Born late 1980s
- Childhood 1990s
- Growing fast 2000s

Haskell and Erlang

- Born late 1980s
- Childhood 1990s
- Growing fast 2000s

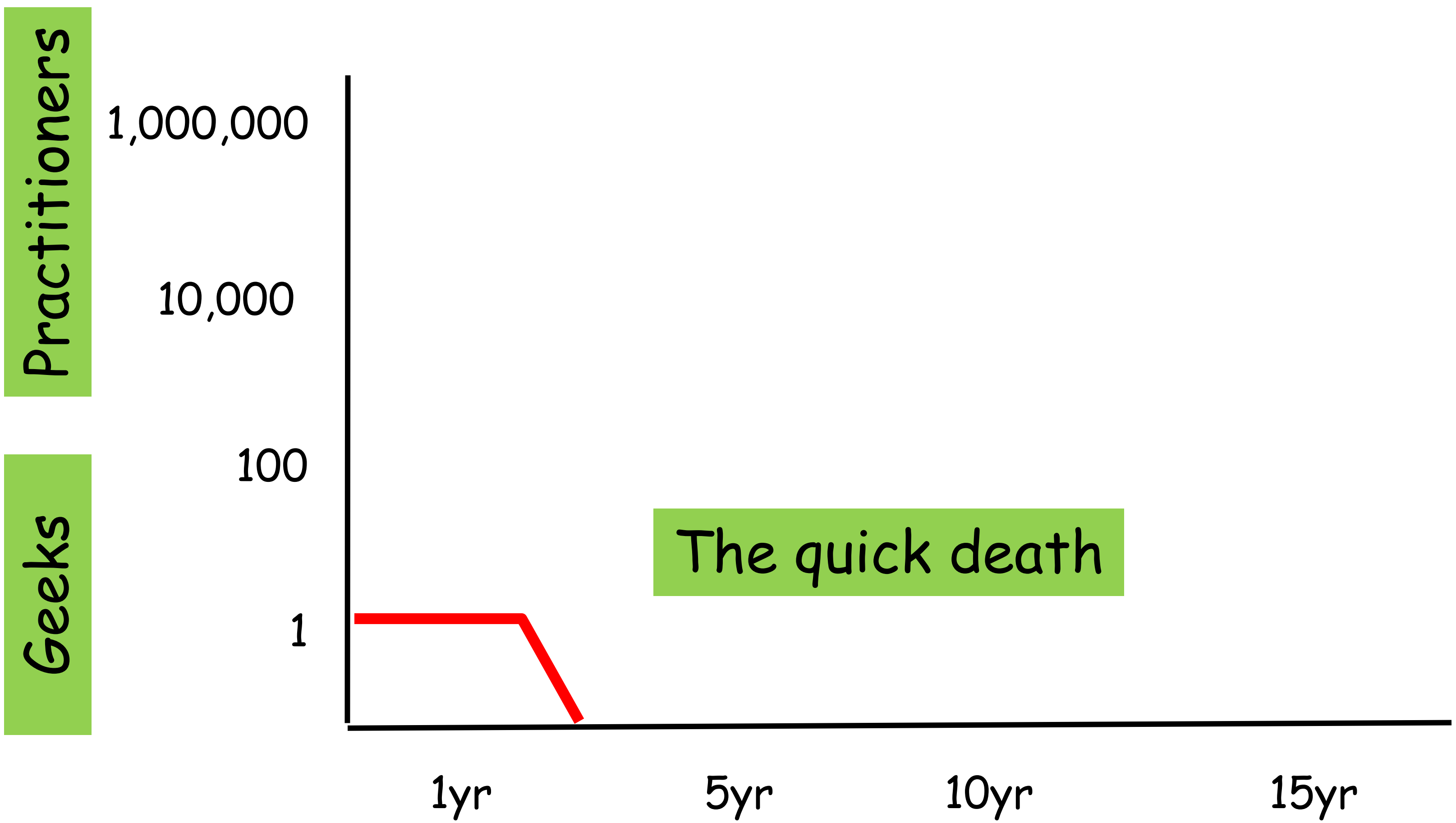
	Haskell	Erlang
Context	Academic	Industrial
Designers	Committee	Joe and Robert
War-cry	Laziness	Concurrency
Original substrate	Lambda calculus	Logic programming
Types	Yes!!!!!!	No!!!!!!

Haskell and Erlang

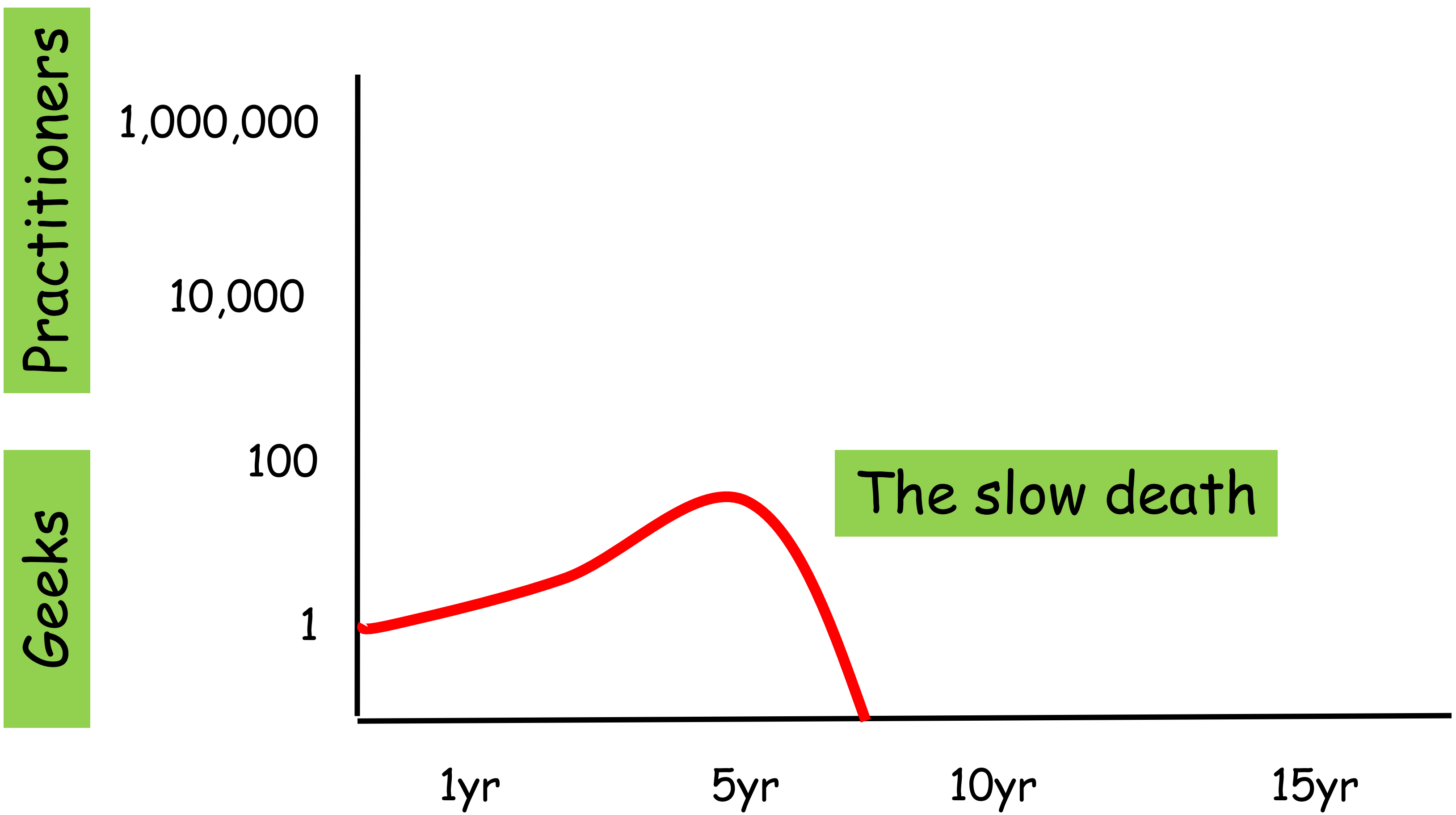
- Born late 1980s
- Childhood 1990s
- Growing fast 2000s

Still thriving

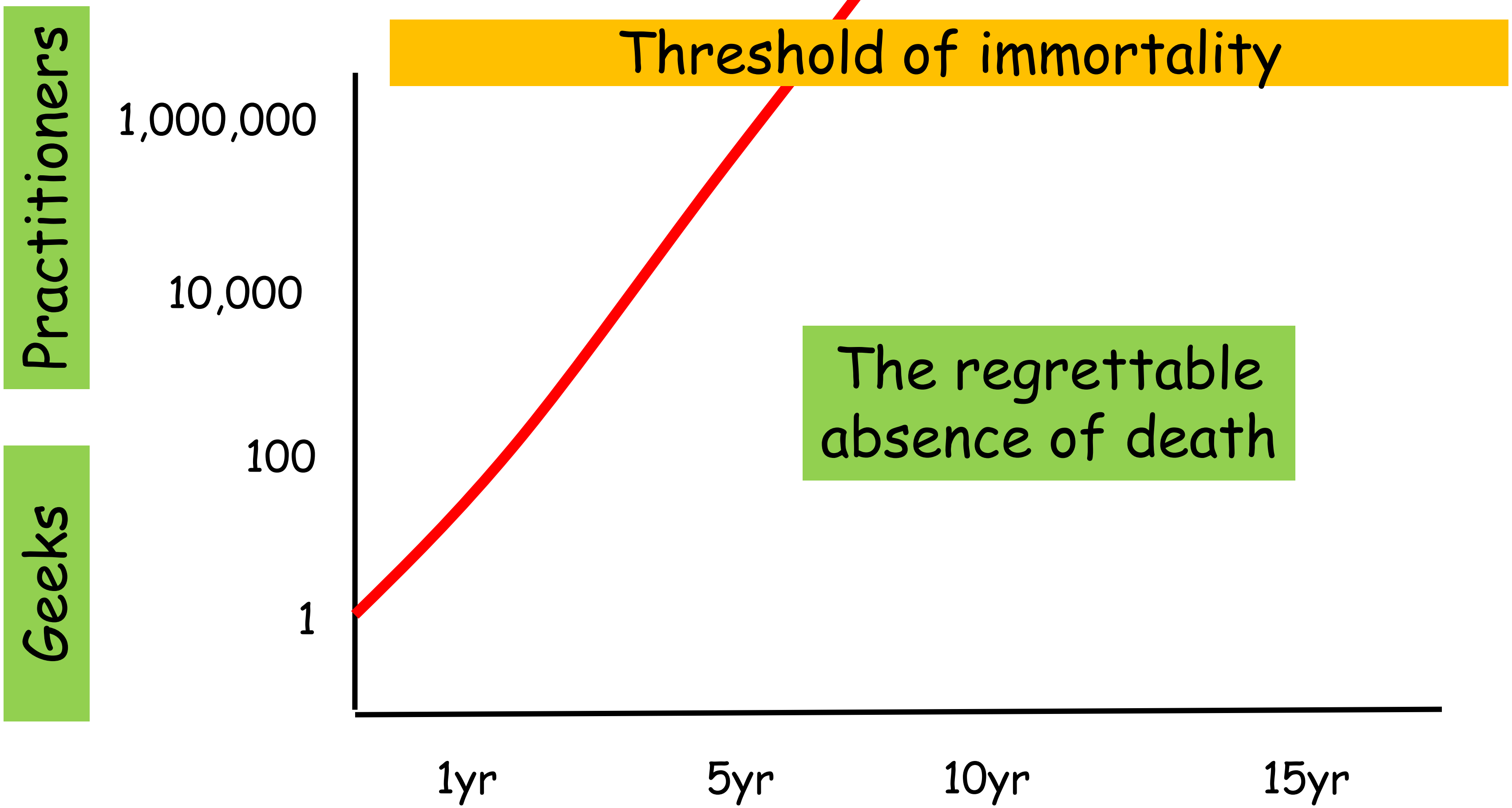
Most new programming languages



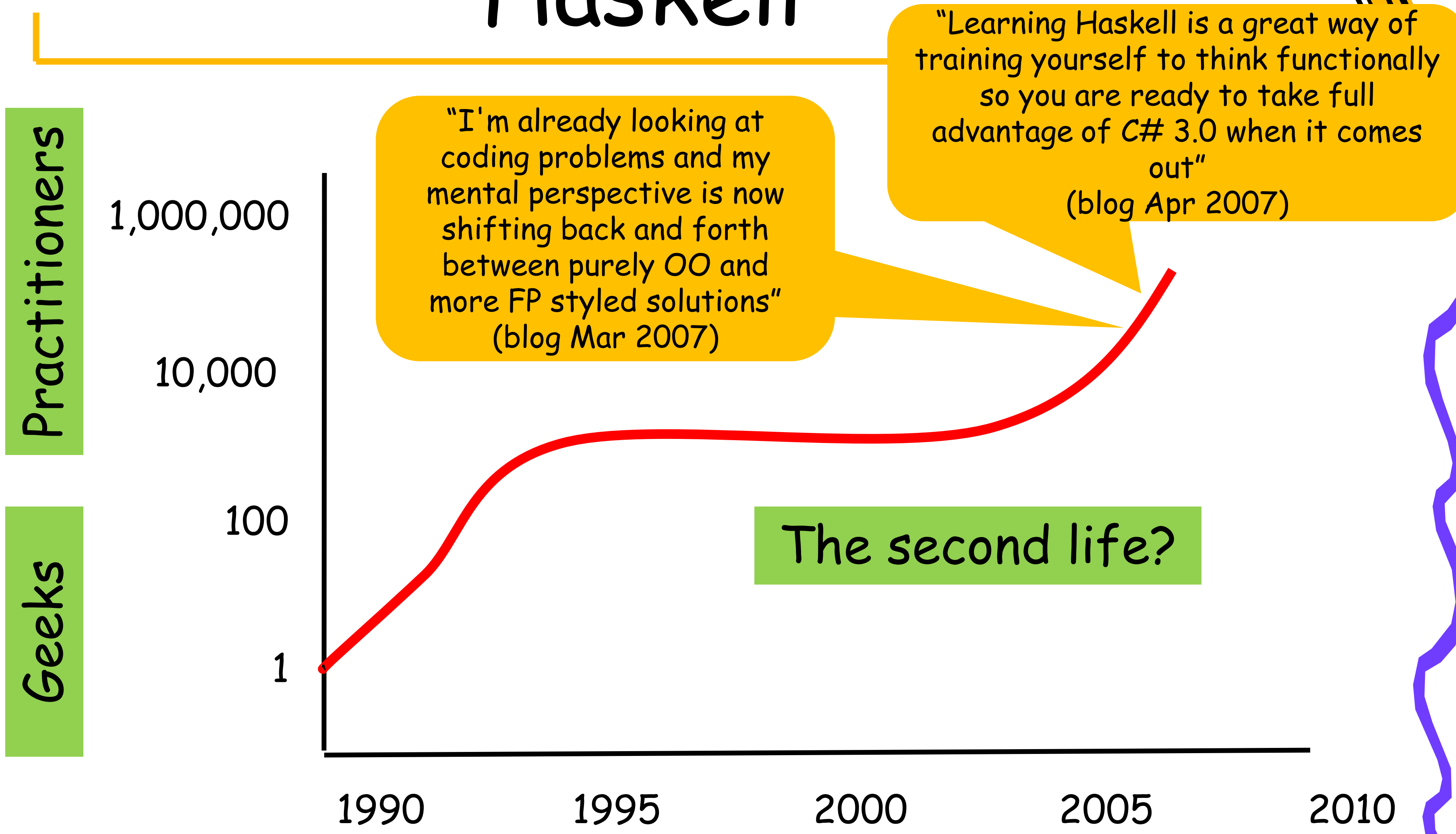
Successful research languages



C++, Java, Perl, Ruby



Haskell



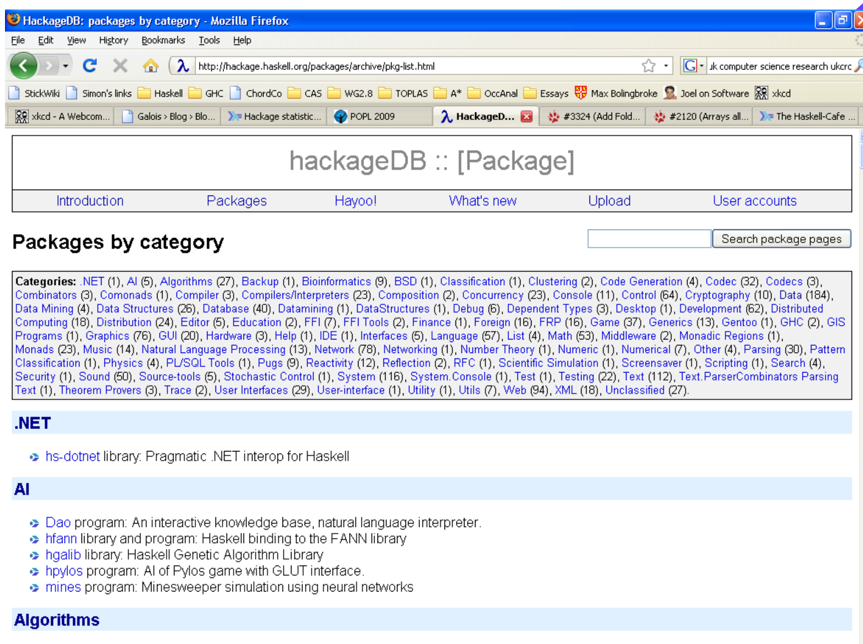
Practitioners
Geeks
1,000,000
10,000
100
1
1990
1995
2000
2005
2010
"I'm already looking at coding problems and my mental perspective is now shifting back and forth between purely OO and more FP styled solutions"
(blog Mar 2007)
"Learning Haskell is a great way of training yourself to think functionally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

Mobilising the community

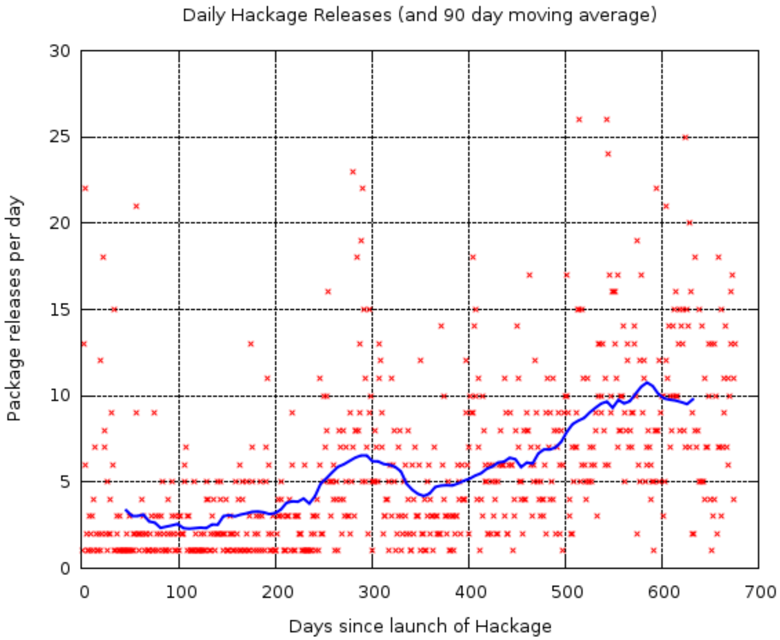
- Package = unit of distribution
- **Cabal**: simple tool to install package and all its dependencies

```
bash$ cabal install pressburger
```

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



Result: staggering



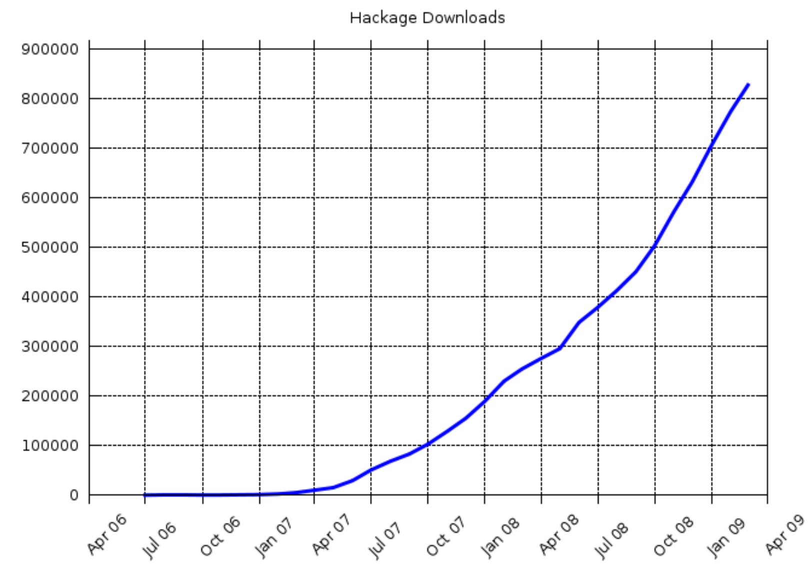
Package uploads

Running at 300/month

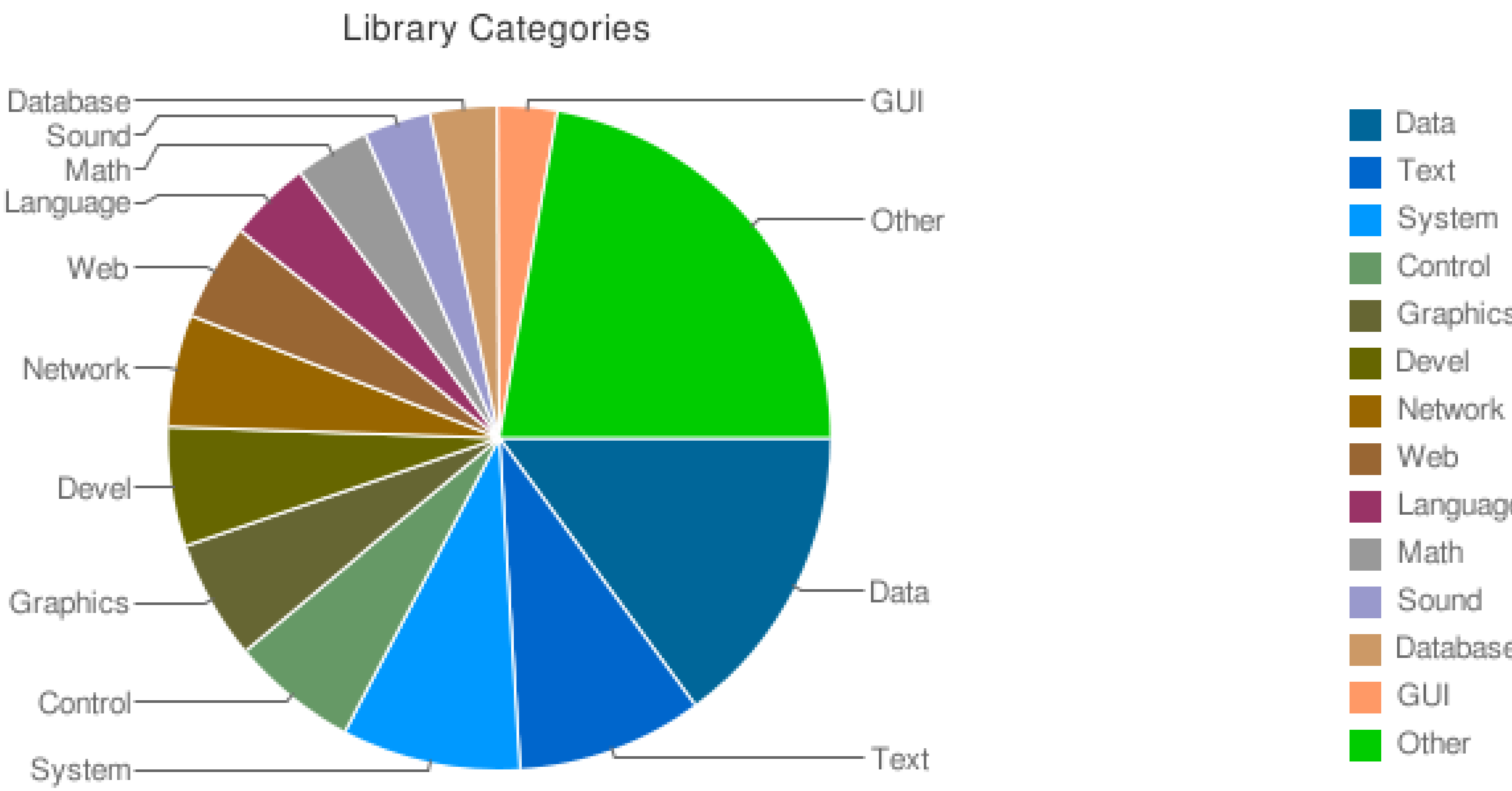
Over 650 packages

Package downloads

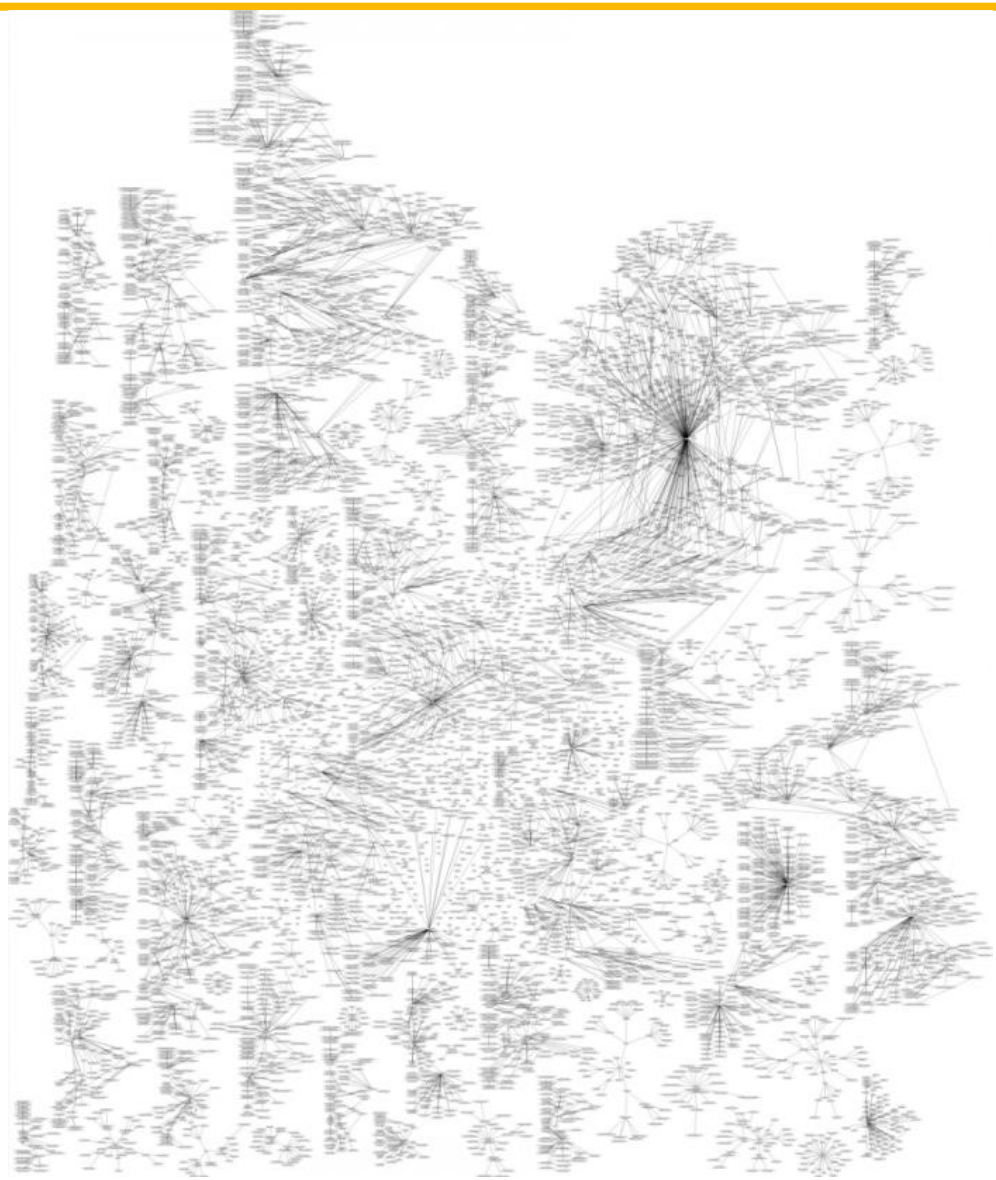
heading for 1 million downloads



The packages on Hackage



The packages on Hackage



Origins

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)

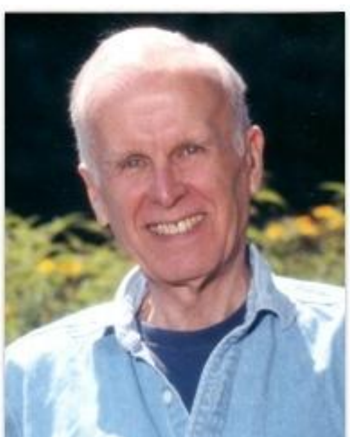
Lambda the Ultimate (Steele, Sussman)

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

Functi
recurs
co
(ML,

er)

Data

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

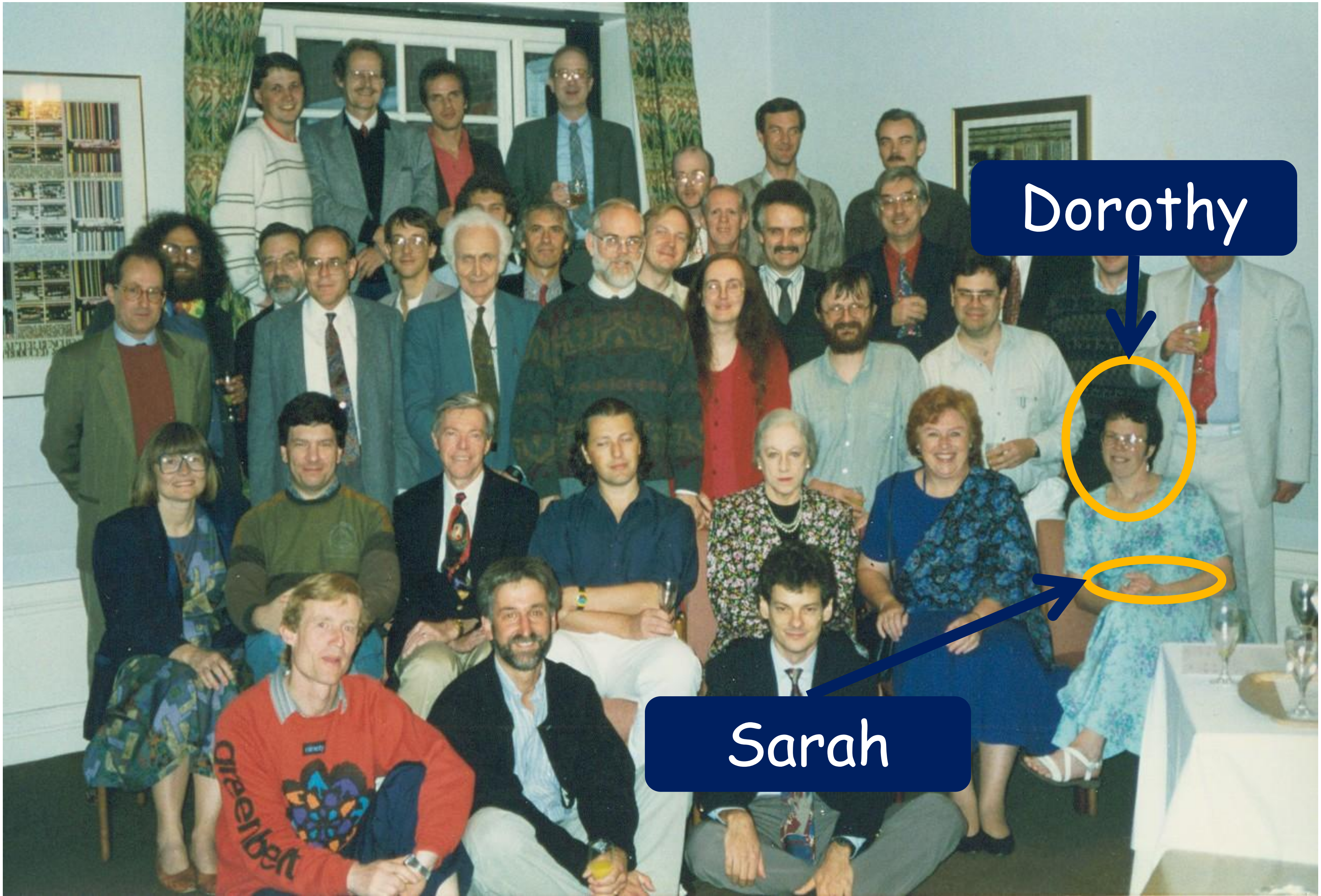
Absolutely no clue how much work we were taking on

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

WG2.8 June 1992

WG2.8 June 1992

WG2.8 June 1992



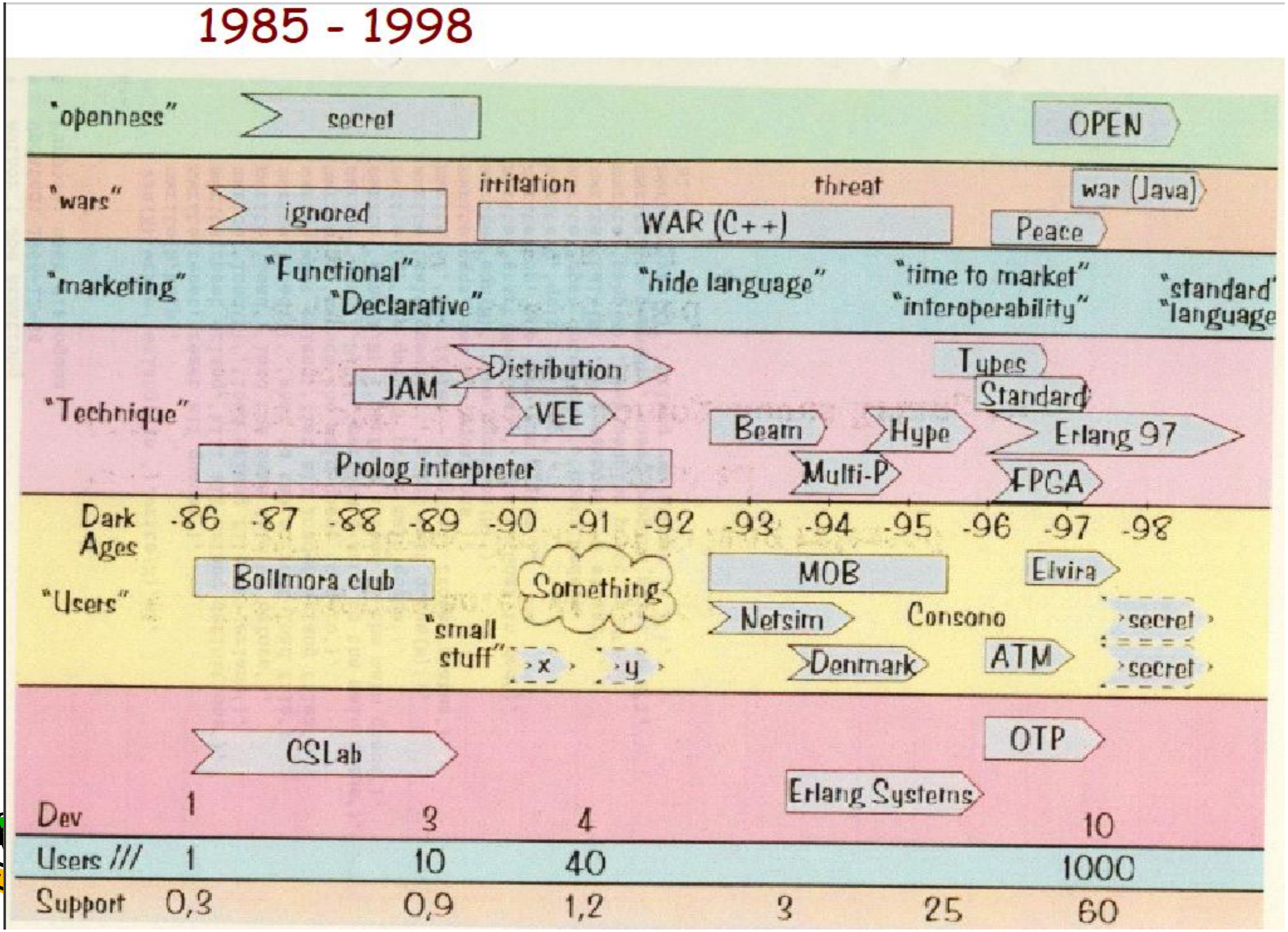

Sarah (b. 1993)


Haskell the cat (b. 2002)

Haskell Timeline

- Sept 87: **kick off**
- Apr 90: **Haskell 1.0**
- Aug 91: **Haskell 1.1 (153pp)**
- May 92: **Haskell 1.2 (SIGPLAN Notices) (164pp)**
  (thank you Richard Wexelblat)
- May 96: **Haskell 1.3. Monadic I/O,**
  separate library report
- Apr 97: **Haskell 1.4 (213pp)**
- Feb 99: **Haskell 98 (240pp)**
- Dec 02: **Haskell 98 revised (260pp)**
- **2003-2007 Growth spurt**

Erlang timeline



A taste of Haskell, flavoured with types

What is Haskell?

Example: lookup in a binary tree

```
lookup :: Tree key val -> key -> val
```

- What if lookup fails?

```
lookup :: Tree key val -> key -> Maybe val

data Maybe a = Nothing | Just a
```

- Failure is represented by data (Nothing), not control (exception)

eg suppose t :: Tree String Int

- lookup t "Fred" = Nothing
- lookup t "Bill" = Just 103

What is Haskell?

```
lookup :: Tree key val -> key -> Maybe val
```

- Can this work for ANY type key?
- No: only those that support ordering
  eg no lookup in Tree (Int->Int) Bool

```
lookup :: Ord key => Tree key val -> key -> Maybe val
```

What is Haskell?

```
lookup :: Tree key val -> key -> Maybe val
```

- Can this work for **ANY** type `key`?
- No: only those that support ordering eg no lookup in Tree (Int->Int) Bool

```
lookup :: Ord key => Tree key val -> key -> Maybe val
```

- Types tell you what the function **does not** do, as well as what it **does** do

```
reverse:: [a] -> [a]
```

implies

```
reverse (map f xs) = map f (reverse xs)
```

Implementing lookup

```
lookup :: Ord key => Tree key val -> key -> Maybe val

data Tree key val
   = Empty
   | Node key val (Tree key val) (Tree key val)
```

Implementing lookup

```
lookup :: Ord key => Tree key val -> key -> Maybe val

data Tree key val
   = Empty
   | Node key val (Tree key val) (Tree key val)

lookup Empty x = Nothing
lookup (Node k v t1 t2) x
  | x < k      = lookup t1 x
  | x == k     = Just v
  | otherwise = lookup t2 x
```

- Pattern matching just like Erlang
- Compiler checks exhaustiveness
- Guards distinguish sub-cases

Haskell is typed, Erlang is not

Conventional wisdom (types are like going to the gym 2 hrs/day)

static type systems detect errors early

- Yes, and that is super-important
- But you can do much of that using other techniques: remorseless testing, code review, agile sumo wrestling etc etc
- And yes, types do get in the way sometimes (eg generic programming)

Why types?

- Types are Haskell's (machine-checked) **design language**
  - they say a lot, but not too much
  - programmers *start* by writing down lots of type signatures and data type declarations
- Types dramatically ease **maintenance**
  - Change the data type declaration, recompile, fix errors. Forces the change to be accounted for everywhere

Why types?

- Types ease **testing**
  - Quickcheck was born in Haskell

```
prop_insert :: Tree Int Char -> Int -> Char -> Bool
prop_insert t x v = case lookup (insert t x v) x of
                      Just w  -> v==w
                      Nothing -> False
```

- Test case generation based on the types:

```
ghci> quickCheck prop_insert
OK! Passed 100 tests!
ghci>
```

Why types?

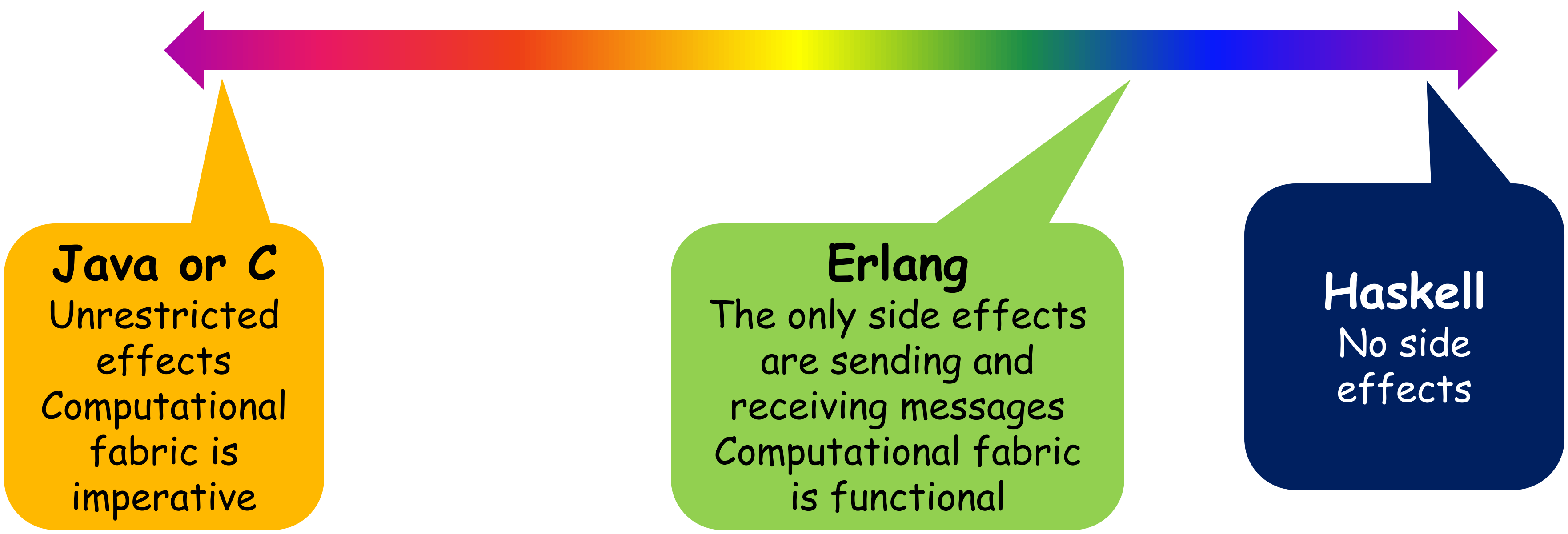
- Types are **fun.** To avoid the "types getting in the way" problem, you need a more expressive type system
- Haskell has turned out to be a laboratory for new type-system ideas.
  - Type classes
  - Existentials
  - Higher-kinded polymorphism
  - Higher rank types
  - Generalised algebraic data types
  - Associated types
  - Type functions

Concurrency

Common ground

- **Embrace** concurrency: millions of lightweight threads
- **Tame** concurrency by

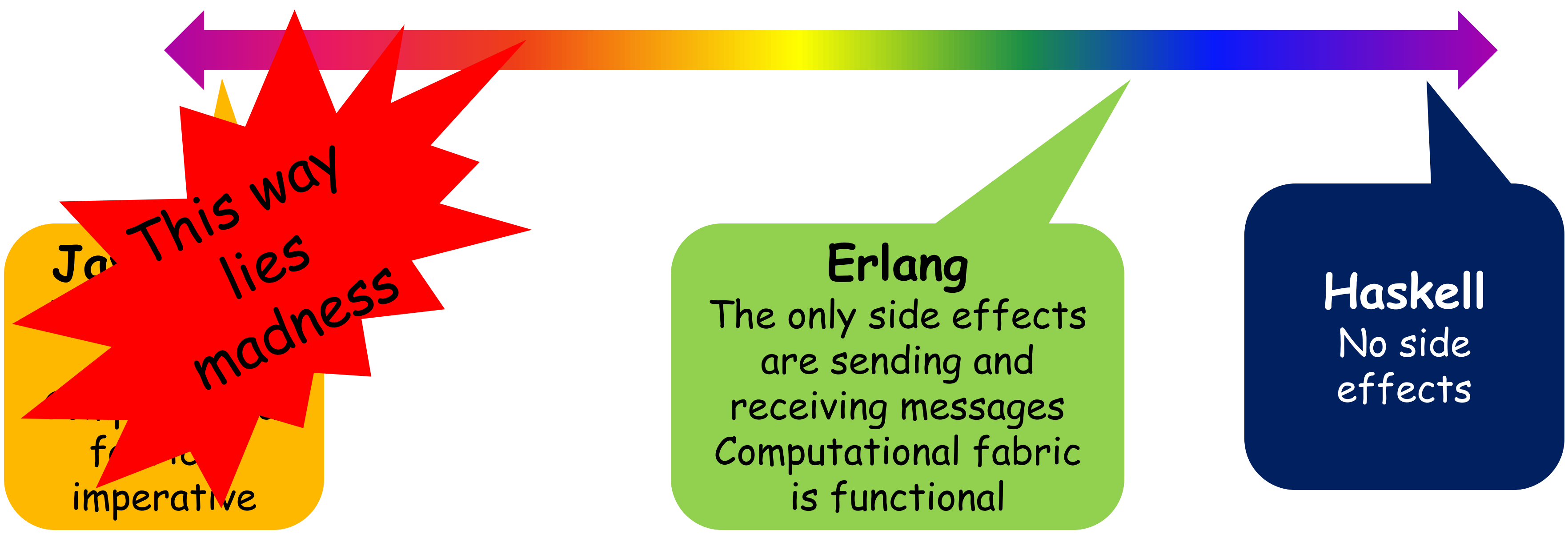
Limiting side effects



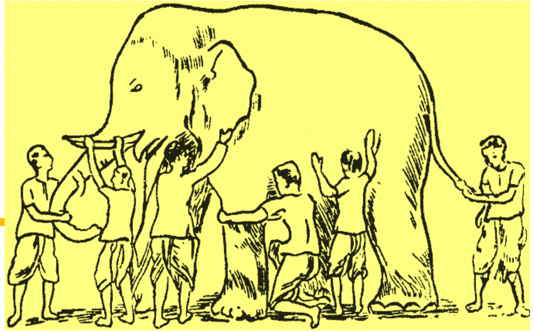
Common ground

- **Embrace** concurrency: millions of lightweight threads
- **Tame** concurrency by

Limiting side effects



"Concurrency" is not one thing



- **Performance**: use many processors to make programs run faster
  - Issues: granularity, locality
- **Programmability**: use threads to express the natural concurrency of the application (eg one thread per phone call)
  - Issues: non-determinism
- **Distribution**: different parts of the program must run in different places
  - Issues: latency, failure, trust, protocols
- **Robustness**: a thread is a plausible unit of kill-and-recover

Concurrency in Haskell

Haskell has at least three concurrency paradigms

- Semi-implicit parallelism (par/seq)
- Explicit threads, and STM
- Data parallelism

Performance: plan A

`e1 + e2` Just evaluate e1 and e2 in parallel

`let x = e1 in e2` Well, maybe....

Evaluating absolutely everything in parallel

- is safe
- but gives WAY too much parallelism
- and WAY too fine grain

Lots of doomed efforts in 1980s to solve this

Performance: plan B

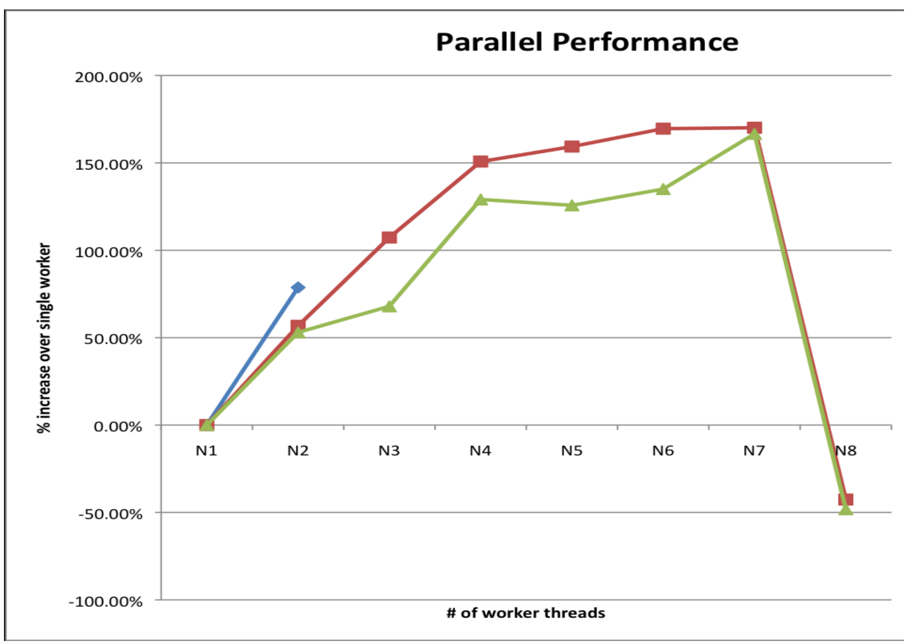
```
f :: Int -> Int
f x = a `par` b `seq` a + b
    where
        a = f (x-1)
        b = f (x-2)
```

- Programmer assistance
  - (x `par` x) tells RTS that x will be needed later
  - (x `seq` y) evaluates x then y
- Result is still deterministic, which is a Huge Win for parallel programming

Happy customers

Ray tracer
527 lines of code
30 hrs work

"I originally planned to spend a few hours working on parallelization. I started playing around with it for fun while I was waking up with coffee one morning. Half an hour and 53 characters later I had around a 40% speedup on two cores. In this, Haskell kind of ruined the project for me. It was too easy to introduce parallelization into the program and have it *just work*. "
18 June 2009
http://blog.finiteimprobability.com/2009/06/18/experience-writing-a-ray-tracer-in-haskell/



- Very modest investment
- Somewhat modest speedup
- Getting really good performance is still an art form

Data parallelism

- \`par\` is too undisciplined
  - pointers everywhere, no locality worth a damn
  - granularity varies massively, even for a single \`par\`
- More promising: data parallelism

```
pmap f [x1, ... , x1000000]
```

  - Locality: lay out the array across the machine
  - Granularity: divide array into chunks, one per processor, run a sequential (map f chunk) on each processor
  - Results still deterministic
  - But programming model is much more restricted

Data parallelism

- `par` is too undisciplined
  - pointers everywhere, no locality worth a damn
  - granularity varies massively, ev... for a single `par`
- More promising: data pa...

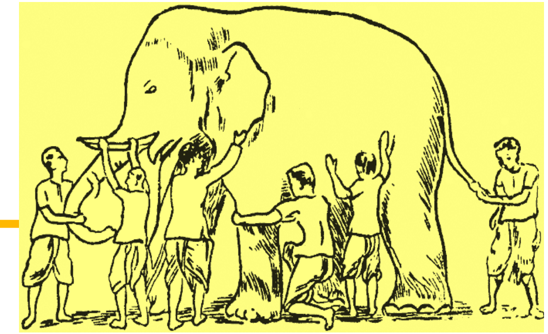
```
pmap f [x1, ... ]
```

  - Locality: lay ou... ...e machine
  - Granularity... ...unks, one per processor, run a se... ...) on each processor
  - Res... ...c
  - B... ...el is much more restricted

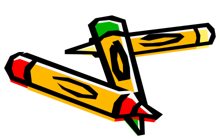
My great white hope

Nested Data Parallelism

"Concurrency" is not one thing



- **Performance**: use many processors to make programs run faster
  - Issues: granularity, locality
- **Programmability**: use threads to express the natural concurrency of the application (eg one thread per phone call)
  - Issues: non-determinism
- **Distribution**: different parts of the program must run in different places
  - Issues: latency, failure, trust, protocols
- **Robustness**: a thread is a plausible unit of kill-and-recover



I/O in Haskell

- How do you do I/O in a language that has **no side effects**?
- Good for making computer hot, but not much else
- Result: **prolonged embarrassment**. Stream-based I/O, continuation I/O... but NO DEALS WIH THE DEVIL

Salvation through monads

A value of type (`IO t`) is an "action" that, when performed, may do some input/output before delivering a result of type t.

```
getChar :: IO Char
putChar :: Char -> IO ()
```

- The main program is an action of type IO ()

```
main :: IO ()
main = putChar 'x'
```

Sequencing I/O operations

```
(>>=)  :: IO a -> (a -> IO b) -> IO b
return :: a -> IO a
```

```
echo :: IO Char
echo = getChar    >>= (\a ->
       putChar a >>= (\() ->
       return a)))
```

The do-notation

```
getChar   >>= \a ->
putchar a >>= \()->
return a
```

==

```
do {
   a <- getChar;
   putchar a;
   return a
}
```

- Syntactic sugar only
- Easy translation into (>>=), return
- Deliberately imperative “look and feel”

Control structures

Values of type (IO t) are first class

So we can define our own “control structures”

```
forever :: IO () -> IO ()
forever a = do { a; forever a }

repeatN :: Int -> IO () -> IO ()
repeatN 0 a = return ()
repeatN n a = do { a; repeatN (n-1) a }
```

```
main = repeatN 10 (putChar 'x')
```

Concurrency

```
forkIO :: IO a -> IO ThreadId
```

- (forkIO m) spawns a new thread that runs m concurrently, and immediately returns its ThreadId

```
main = do { forkIO (print "Hello");
            print "Goodbye" }
```

HeGoodlldbyoe

- The big question: how do threads coordinate?

STM

```
newRef   :: a -> IO (Ref a)
readRef  :: Ref a -> IO a
writeRef :: Ref a -> a -> IO ()
```

```
main :: IO ()
main = do { r <- newRef 0
          ; forkIO (addR r 1)
          ; addR r 10
          ; v <- readRef r
          ; print v}

addR :: Ref Int -> Int -> IO ()
addR r n = do { v <- readRef r
              ; writeRef r (v+n)}
```

- Bad interleaving => prints 1 (not 10 or 11)

STM

```
atomic :: IO a -> IO a
```

```
main :: IO ()
main = do { r <- newTVar 0
          ; forkIO (atomic (addR r 1))
          ; atomic (addR r 10)
          ; v <- readTVar r
          ; print v}

addR :: Ref Int -> Int -> IO ()
addR r n = do { v <- readTVar r
              ; writeTVar r (v+n)}
```

- (atomic m) runs m atomically wrt all other threads

STM in practice

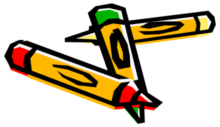
- Want to allow the implementation the opportunity of using **optimistic concurrency**
  - run the transaction in the expectation of no conflict, keeping effects invisible to other threads
  - at the end, check for conflict
    - no conflict: commit the effects
    - conflict: undo private effects, and re-rerun from the start
- Consequences
  - Track every read and write to mutable state (easy in Haskell, not so easy in C#)
  - Do not allow I/O inside a transaction
  - Hence: classify effects into:
    - Reads and writes of tracked mutable variables
    - Arbitrary I/O

STM

```
newRef    :: a -> STM (TVar a)
readRef   :: TVar a -> STM a
writeRef  :: TVar a -> a -> STM ()
atomic    :: STM a -> IO a
```

```
main :: IO ()
main = do { r <- atomic (newTVar 0)
          ; forkIO (atomic (addR r 1))
          ; atomic (addR r 10)
          ; v <- atomic(readTVar r)
          ; print v}

addR :: Ref Int -> Int -> STM ()
addR r n = do { v <- readTVar r
              ; writeTVar r (v+n)}
```

- Type system guarantees
  - no I/O inside transaction
  - no mutation of TVars outside transaction



More STM

- Studying STM led to an elegant, compositional mechanism for
  - blocking
  - choice

```
retry  :: STM a
orElse :: STM a -> STM a -> STM a
```

- Now being adopted by the mainstream

Actor concurrency

- Using STM (or MVars) it is very easy to build buffered channels

```
newChan :: Chan a
send    :: Chan a -> a -> STM ()
receive :: Chan a -> STM a
```

- ...which in turn lets you write programs Erlang-style if you want
- ...but with new forms of composition

```
receive c1 `orElse` receive c2
```

What I envy about Erlang

- **Share-nothing threads** are part of Erlang's core design
- That is a limitation, but it has many useful payoffs:
  - Easy distribution across multicore
  - Per-thread garbage collection
  - Excellent failure model

The future

1. Scheme, Erlang, Haskell, Ocaml, F#, Scala are all demonstrably valuable to Hard Nosed Developers, in interestingly different ways.
   - Functional programming is still a niche... but it is fast becoming a shelf.
   - Diversity is good
2. We may not rule the world, but the world is increasingly listening. That is a privilege and a responsibility.
3. Concurrency is complicated; no free lunch
4. **The highly-concurrent languages of the future will be functional.** (Although they many not be *called* functional.)

Backup slides

What have we achieved?

- The ability to mix imperative and purely-functional programming, without ruining either
- All laws of pure functional programming remain unconditionally true, even of actions

e.g.

let x=e in ...x....x...

=

....e....e.....

Type classes

Type classes

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

instance Eq Int where
  i1 == i2 = eqInt i1 i2

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


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

Initially, just a neat way to get systematic overloading of (==), read, show.

Implementing type classes

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

dEqInt :: Eq Int
dEqInt = MkEq eqInt

dEqList :: Eq a -> Eq [a]
dEqList (MkEq e) = MkEq el
  where el []     []     = True
        el (x:xs) (y:ys) = x `e` y && xs `el` ys



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

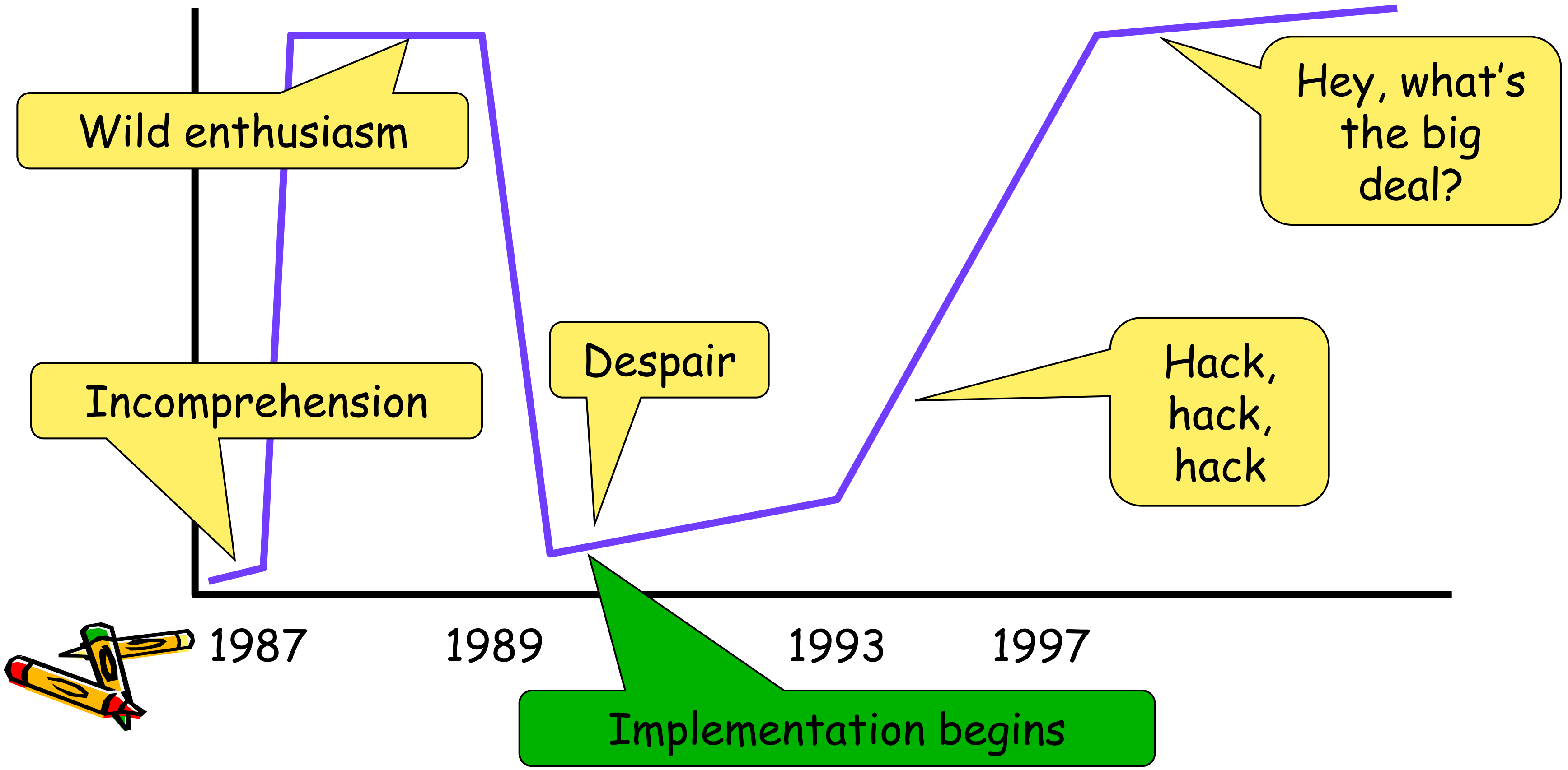
Class witnessed by a "dictionary" of methods

Instance declarations create dictionaries

Overloaded functions take extra dictionary parameter(s)

Type classes over time

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



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....

In Haskell, my 17 can definitely be your 23

Note the higher-kinded type variable, m

Quickcheck

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

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

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

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

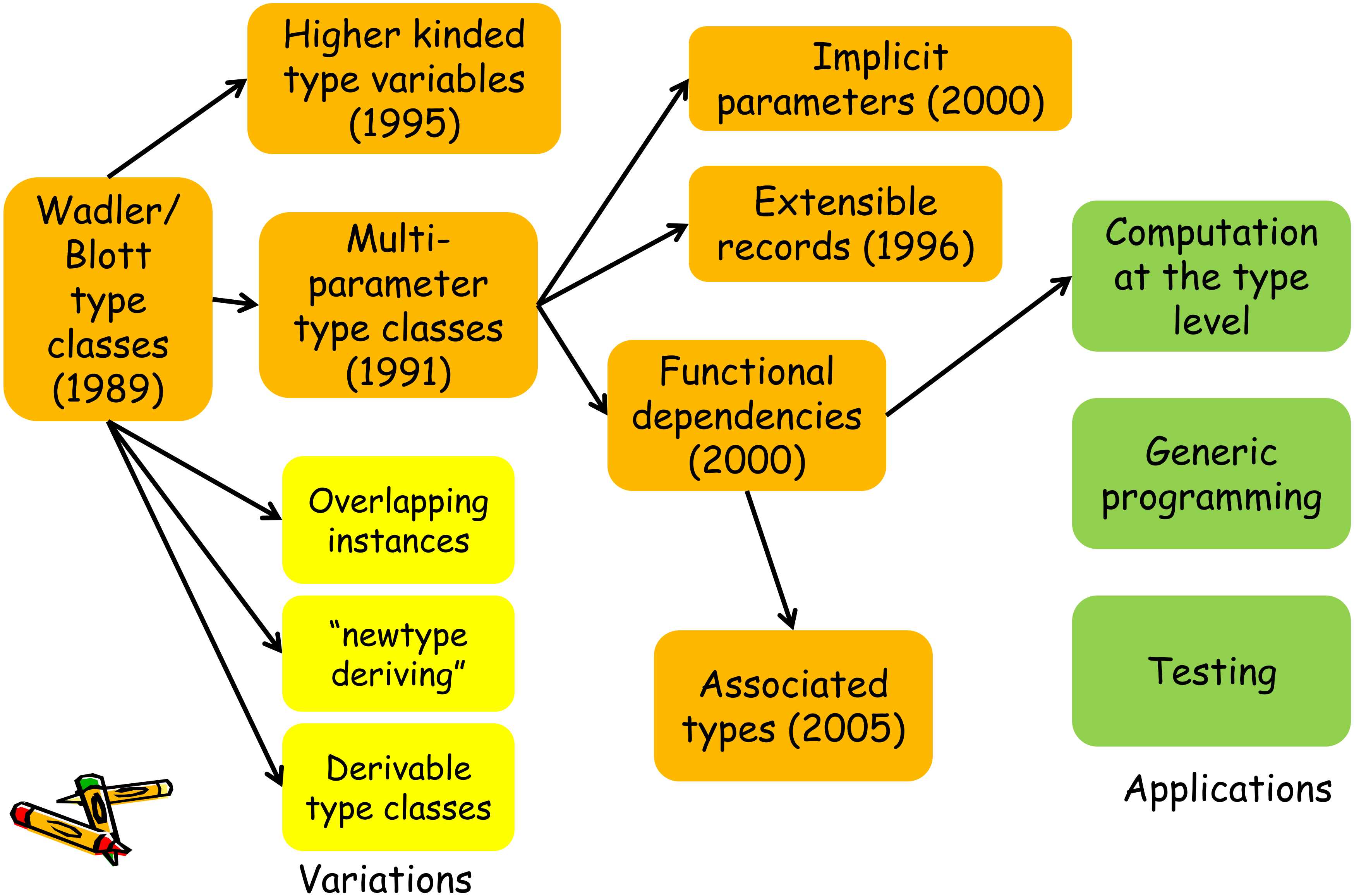
Quickcheck (which is just a Haskell 98 library)

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

Quickcheck

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

class Test a where
  test :: a -> Rand -> Bool

class Arby a where
  arby :: Rand -> a

instance (Arby a, Test b) => Test (a->b) where
  test f r = test (f (arby r1)) r2
             where (r1,r2) = split r

instance Test Bool where
  test b r = b
```

Type-class fertility



Type classes summary

- A much more far-reaching idea than we first realised: the automatic, type-driven generation of executable "evidence"
- Many interesting generalisations, still being explored
- Variants adopted in Isabel, Clean, Mercury, Hal, Escher
- Danger of Heat Death
- Long term impact yet to become clear

Process and community

A committee language

- No Supreme Leader
- A powerfully motivated design group who trusted each other
- The Editor and the Syntax Tzar
- Committee explicitly disbanded 1999

Language complexity

- "Languages are too complex, fraught with dispensable features and facilities." (Wirth, HOPL 2007)
- Much superficial complexity (e.g. redundant syntactic forms),
- No formal semantics
- Nevertheless, underpinned by Deeply Held Principles

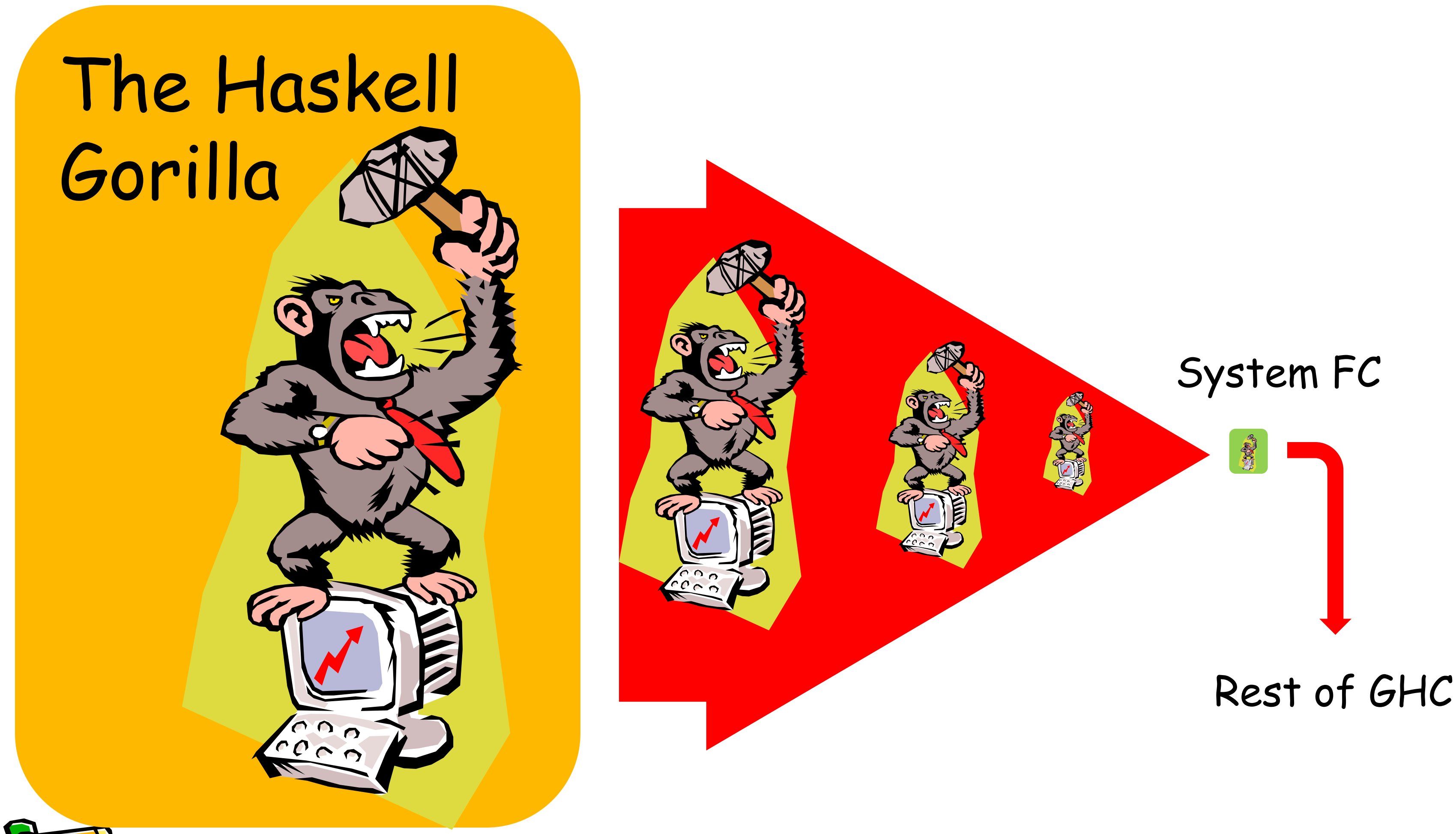
"Deeply held principles"

- System F is GHC's intermediate language

(Well, something very like System F.)

```
data Expr
  = Var    Var
  | Lit    Literal
  | App    Expr Expr
  | Lam    Var Expr
  | Let    Bind Expr
  | Case   Expr Var Type [(AltCon, [Var], Expr)]
  | Cast   Expr Coercion
  | Note   Note Expr
  | Type   Type
type Coercion = Type
data Bind    = NonRec Var Expr | Rec [(Var,Expr)]
data AltCon = DEFAULT | LitAlt Lit | DataAlt DataCon
```

Sanity check on wilder excesses



Haskell users

- A smallish,
  tolerant,
  rather pointy-headed, and
  extremely friendly
  user-base makes Haskell nimble. Haskell has evolved rapidly and continues to do so.

- Haskell users react to new features like hyenas react to red meat

Lesson: **avoid success at all costs**

The price of usefulness

- Libraries increasingly important:
  - 1996: Separate libraries Report
  - 2001: Hierarchical library naming structure, increasingly populated
  - 2006: Cabal and Hackage: packaging and distribution infrastructure
- Foreign-function interface increasingly important
  - 1993 onwards: a variety of experiments
  - 2001: successful effort to standardise a FFI across implementations
- Lightweight concurrency, asynchronous exceptions, bound threads, transactional memory, data parallelism...

Any language large enough to be useful becomes dauntingly complex

Conclusion

- Haskell does not meet Bjarne's criterion (be good enough on all axes)
- Instead, like Self, it aspires to take a few beautiful ideas (esp: purity and polymorphism), pursue them single-mindedly, and see how far they can take us.
- In the end, we want to infect your brain, not your hard drive

Luck

- Technical excellence helps, but is neither necessary nor sufficient for a language to succeed
- Luck, on the other hand, is definitely necessary
- We were certainly lucky: the conditions that led to Haskell are hard to reproduce (witness Haskell')

Fun

- Haskell is rich enough to be useful
- But above all, Haskell is a language in which people **play**
  - Programming as an art form
  - Embedded domain-specific languages
  - Type system hacks
- Play leads to new discoveries

Encapsulating it all

```
data ST s a     -- Abstract
newRef :: a -> ST s (STRef s a)
read   :: STRef s a -> ST s a
write  :: STRef s a -> a -> ST s ()
```

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

Stateful computation

Pure result

```
sort :: Ord a => [a] -> [a]
sort xs = runST (do { ..in-place sort.. })
```

Encapsulating it all

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

Higher rank type

Security of encapsulation depends on parametricity

Monads

And that depends on type classes to make non-parametric operations explicit (e.g. f :: Ord a => a -> a)

Parametricity depends on there being few polymorphic functions (e.g.. f:: a->a means f is the identity function or bottom)

And it also depends on purity (no side effects)

The Haskell committee

Arvind
Lennart Augustsson
Dave Barton
Brian Boutel
Warren Burton
Jon Fairbairn
Joseph Fasel
Andy Gordon
Maria Guzman
Kevin Hammond
Ralf Hinze
Paul Hudak [editor]
John Hughes [editor]

Thomas Johnsson
Mark Jones
Dick Kieburtz
John Launchbury
Erik Meijer
Rishiyur Nikhil
John Peterson
Simon Peyton Jones [editor]
Mike Reeve
Alastair Reid
Colin Runciman
Philip Wadler [editor]
David Wise
Jonathan Young

Syntax

Syntax

- Syntax is not important

- Parsing is the easy bit of a compiler

Syntax

~~Syntax is not important~~

Syntax is the user interface of a language

~~Parsing is the easy bit of a compiler~~

The parser is often the trickiest bit of a compiler

Good ideas from other languages

List comprehensions

```
[(x,y) | x <- xs, y <- ys, x+y < 10]
```

Separate type signatures

```
head :: [a] -> a
head (x:xs) = x
```

Upper case constructors

```
f True true = true
```

DIY infix operators

```
f `map` xs
```

Optional layout

```
let x = 3
    y = 4
in x+y
```

```
let { x = 3; y = 4} in x+y
```

"Declaration style"

Define a function as a series of independent equations

```
map f []      = []
map f (x:xs) = f x : map f xs
```

```
sign x  | x>0        = 1
        | x==0       = 0
        | x<0        = -1
```

"Expression style"

Define a function as an expression

```
map = \f xs -> case xs of
                 []     -> []
                 (x:xs) -> map f xs
```

```
sign = \x -> if x>0 then 1
             else if x==0 then 0
             else -1
```

Fat vs thin

Expression style

- Let
- Lambda
- Case
- If

Declaration style

- Where
- Function arguments on lhs
- Pattern-matching
- Guards

SLPJ's conclusion
syntactic redundancy is a big win

Tony Hoare's comment "I fear that Haskell is doomed to succeed"

Example (ICFP02 prog comp)

Pattern match

Guard

Pattern guard

Conditional

Where clause

```
sp_help item@(Item cur_loc cur_link _) wq vis
  | cur_length > limit          -- Beyond limit
  = sp wq vis
  | Just vis_link <- lookupVisited vis cur_loc
  =     -- Already visited; update the visited
        -- map if cur_link is better
    if cur_length >= linkLength vis_link then
        -- Current link is no better
         sp wq vis
    else
        -- Current link is better
         emit vis item ++ sp wq vis'

  | otherwise  -- Not visited yet
  = emit vis item ++ sp wq' vis'
  where
   vis’ = ...
   wq   = ...
```

So much for syntax...

What is important or interesting about Haskell?

What really matters?

Laziness

Type classes

Sexy types

In favour of laziness

Laziness is jolly convenient

```
sp_help item@(Item cur_loc cur_link _) wq vis
  | cur_length > limit          -- Beyond limit
  = sp wq vis
  | Just vis_link <- lookupVisited vis cur_loc
  = if cur_length >= linkLength vis_link then
        sp wq vis
    else
        emit vis item ++ sp wq vis'

  | otherwise
  = emit vis item ++ sp wq' vis'
  where
   vis' = ...
   wq'  = ...
```

Used in two cases

Used in one case

Combinator libraries

Recursive values are jolly useful

```
type Parser a = String -> (a, String)

exp :: Parser Expr
exp = lit "let" <+> decls <+> lit "in" <+> exp
   ||| exp <+> aexp
   ||| ...etc...
```

This is illegal in ML, because of the value restriction

Can only be made legal by eta expansion.

But that breaks the Parser abstraction, and is extremely gruesome:

```
exp x = (lit "let" <+> decls <+> lit "in" <+> exp
     ||| exp <+> aexp
     ||| ...etc...) x
```

Sexy types

Sexy types

Haskell has become a laboratory and playground for advanced type hackery

- Polymorphic recursion
- Higher kinded type variables
  ```
  data T k a = T a (k (T k a))
  ```
- Polymorphic functions as constructor arguments
  ```
  data T = MkT (forall a. [a] -> [a])
  ```
- Polymorphic functions as arbitrary function arguments (higher ranked types)
  ```
  f :: (forall a. [a]->[a]) -> ...
  ```
- Existential types
  ```
  data T = exists a. Show a => MkT a
  ```

Is sexy good? Yes!

- Well typed programs don't go wrong
- Less mundanely (but more allusively) sexy types let you think higher thoughts and still stay [almost] sane:
  - deeply higher-order functions
  - functors
  - folds and unfolds
  - monads and monad transformers
  - arrows (now finding application in real-time reactive programming)
  - short-cut deforestation
  - bootstrapped data structures

How sexy?

- Damas-Milner is on a cusp:
  - Can infer most-general types without any type annotations at all
  - But virtually any extension destroys this property
- Adding type quite modest type annotations lets us go a LOT further (as we have already seen) without losing inference for most of the program.
- Still missing from even the sexiest Haskell impls
  - λ at the type level
  - Subtyping
  - Impredicativity

Destination = $F^{w}_{<:}$

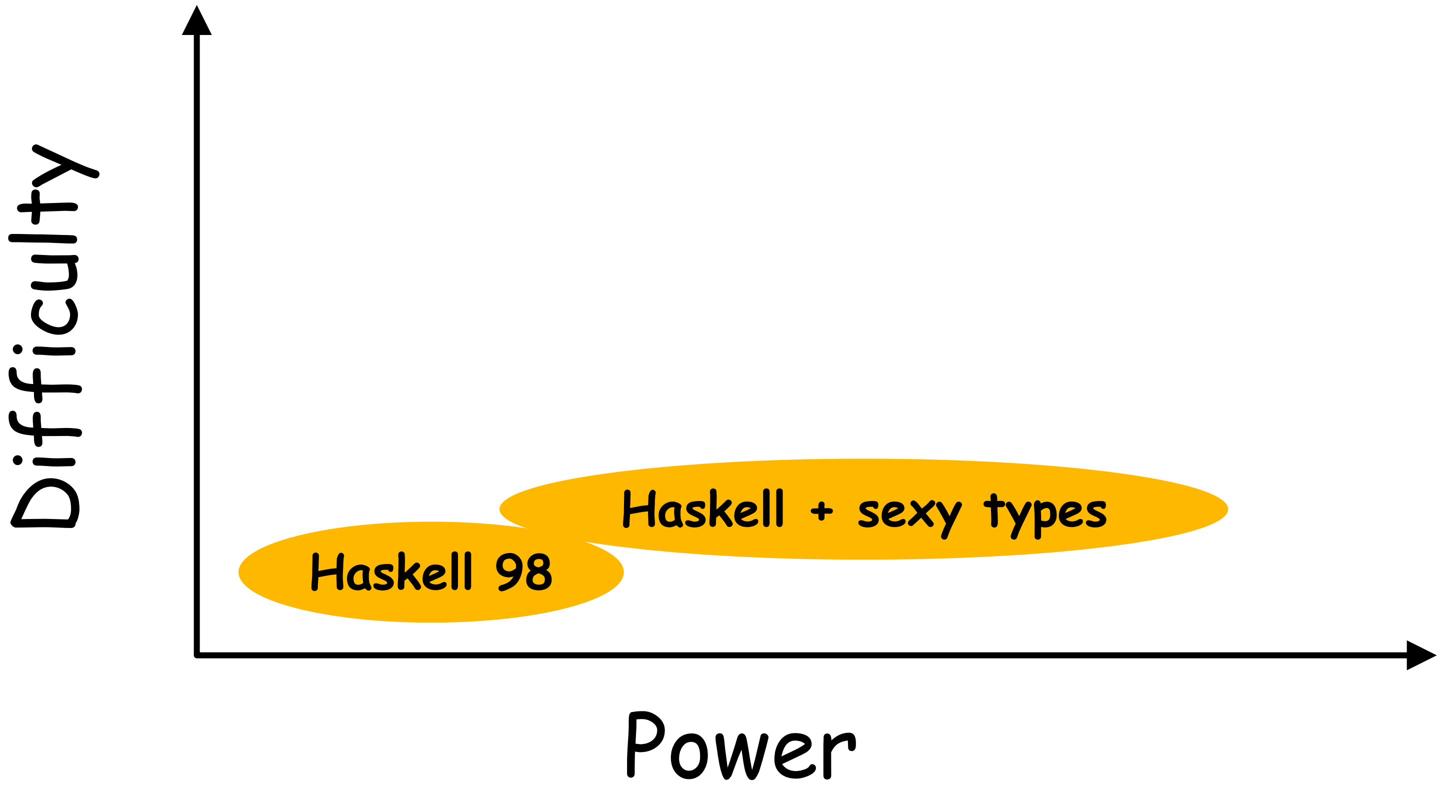
Open question

What is a good design for user-level type annotation that exposes the power of F^{w} or $F^{w}_{<:}$, but co-exists with type inference?

C.f. Didier & Didier's MLF work

Modules

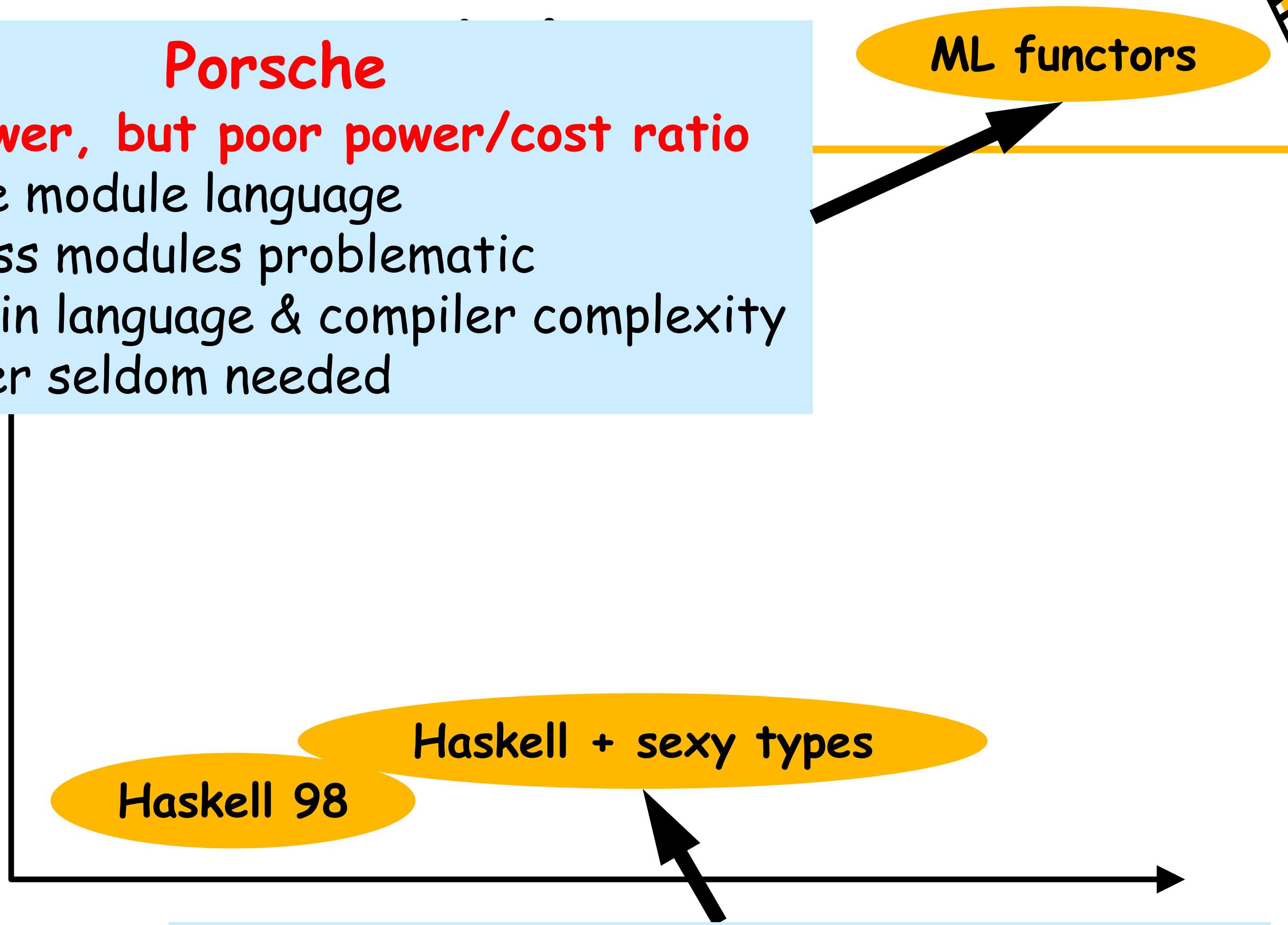
ML functors



Porsche

High power, but poor power/cost ratio

- Separate module language
- First class modules problematic
- Big step in language & compiler complexity
- Full power seldom needed



Ford Cortina with alloy wheels

Medium power, with good power/cost

- Module parameterisation too weak
- No language support for module signatures

Modules

- Haskell has many features that overlap with what ML-style modules offer:
  - type classes
  - first class universals and existentials
- Does Haskell need functors anyway? No: one seldom needs to instantiate the same functor at different arguments
- But Haskell lacks a way to distribute "open" libraries, where the client provides some base modules; need module signatures and type-safe linking (e.g. PLT,Knit?). π not λ!
- **Wanted: a design with better power, but good power/weight.**

Monads

- Exceptions

```
type Exn a = Either String a
fail :: String -> Exn a
```

- Unique supply

```
type Uniq a = Int -> (a, Int)
new :: Uniq Int
```

- Parsers

```
type Parser a = String -> [(a,String)]
alt :: Parser a -> Parser a -> Parser a
```

Monad combinators (e.g. sequence, fold, etc), and do-notation, work over all monads

Example: a type checker

```
tcExpr :: Expr -> Tc Type
tcExpr (App fun arg)
    = do { fun_ty <- tcExpr fun
         ; arg_ty <- tcExpr arg
         ; res_ty <- newTyVar
         ; unify fun_ty (arg_ty --> res_ty)
         ; return res_ty }
```

Tc monad hides all the plumbing:

- Exceptions and failure
- Current substitution (unification)
- Type environment
- Current source location
- Manufacturing fresh type variables

Robust to changes in plumbing

The IO monad

The IO monad allows controlled introduction of other effect-ful language features (not just I/O)

- State

```
newRef :: IO (IORef a)
read   :: IORef s a -> IO a
write  :: IORef s a -> a -> IO ()
```

- Concurrency

```
fork     :: IO a -> IO ThreadId
newMVar  :: IO (MVar a)
takeMVar :: MVar a -> IO a
putMVar  :: MVar a -> a -> IO ()
```

Performing I/O

```
main :: IO a
```

- A program is a single I/O action
- Running the program performs the action
- The type tells the effects:
  - reverse :: String -> String
  - searchWeb :: String -> IO [String]

What we have not achieved

- Imperative programming is no easier than it always was

e.g. `do { ...; x <- f 1; y <- f 2; ...}`

?=?

`do { ...; y <- f 2; x <- f 1; ...}`

- ...but there's less of it!
- ...and actions are first-class values

Our biggest mistake

Using the scary term
“monad”
rather than
“warm fuzzy thing”

Open challenge 1

Open problem: the IO monad has become Haskell's sin-bin. (Whenever we don't understand something, we toss it in the IO monad.)

Festering sore:

```
unsafePerformIO :: IO a -> a
```

Dangerous, indeed type-unsafe, but occasionally indispensable.

Wanted: finer-grain effect partitioning

e.g. IO {read x, write y} Int

Open challenge 2

Which would you prefer?

```
do { a <- f x;
     b <- g y;
     h a b }
```

```
h (f x) (g y)
```

In a commutative monad, it does not matter whether we do `(f x)` first or `(g y)`.

Commutative monads are very common. (Environment, unique supply, random number generation.) For these, monads over-sequentialise.

Wanted: theory and notation for some cool compromise.

Monad summary

- Monads are a beautiful example of a theory-into-practice (more the thought pattern than actual theorems)
- Hidden effects are like hire-purchase: pay nothing now, but it catches up with you in the end
- Enforced purity is like paying up front: painful on Day 1, but usually worth it
- But we made one big mistake...

Extensiblity

- Like OOP, one can add new data types “later”. E.g. QuickCheck works for your new data types (provided you make them instances of **`Arby`**)
- ...but also not like OOP

Type-based dispatch

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

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

```
double :: Num a => a -> a
double x = x+x
```

- No: type classes implement type-based dispatch, not value-based dispatch

Type-based dispatch

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

```
double :: Num a => a -> a
double x = 2*x
```

means

```
double :: Num a -> a -> a
double d x = mul d (fromInteger d 2) x
```

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

Type-based dispatch

So the links to intensional polymorphism are much 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...
```

Haskell

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

C.f. Crary et al λR (ICFP98), Baars et al (ICFP02)

Cool generalisations

- Multi-parameter type classes
- Higher-kinded type variables (a.k.a. constructor classes)
- Overlapping instances
- Functional dependencies (Jones ESOP'00)
- Type classes as logic programs (Neubauer et al POPL'02)

Qualified types

- Type classes are an example of qualified types [Jones thesis]. Main features
  - types of form $\forall\alpha.\mathbf{Q} \Rightarrow \tau$
  - qualifiers Q are witnessed by run-time evidence
- Known examples
  - type classes (evidence = tuple of methods)
  - implicit parameters (evidence = value of implicit param)
  - extensible records (evidence = offset of field in record)
- Another unifying idea: Constraint Handling Rules (Stucky/Sulzmann ICFP'02)