Being lazy with class
A history of Haskell

Paul Hudak, Yale University
John Hughes, Chalmers University,
Simon Peyton Jones, Microsoft Research
Phil Wadler, University of Edinburgh

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)

e.g. (\x. x+x) 5
= S (S (K +) I) I 5

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)

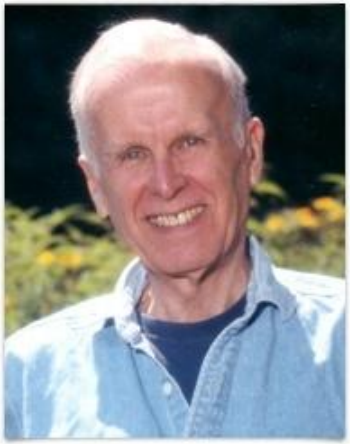
Lambda the Ultimate (Steele, Sussman)

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
con
(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

April 1990 ($2\frac{1}{2}$ yrs later): **Haskell 1.0** report

Timeline

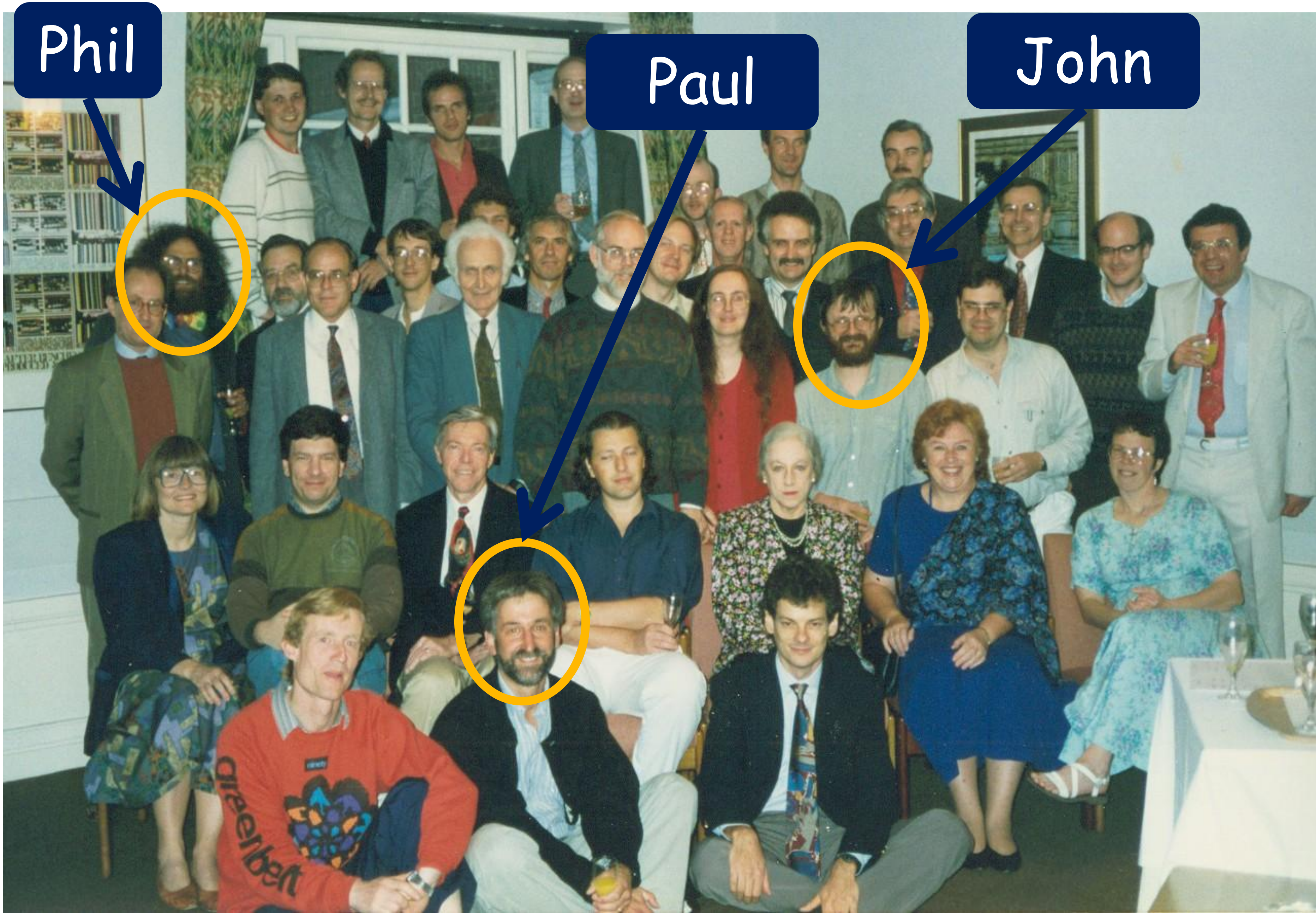
Haskell Curry
1900-1982

- Sept 87: **kick off; choose name**
- Apr 90: **Haskell 1.0**
- Aug 91: **Haskell 1.1 (153pp)**
- May 92: **Haskell 1.2 (SIGPLAN Notices) (164pp)**
- 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)**

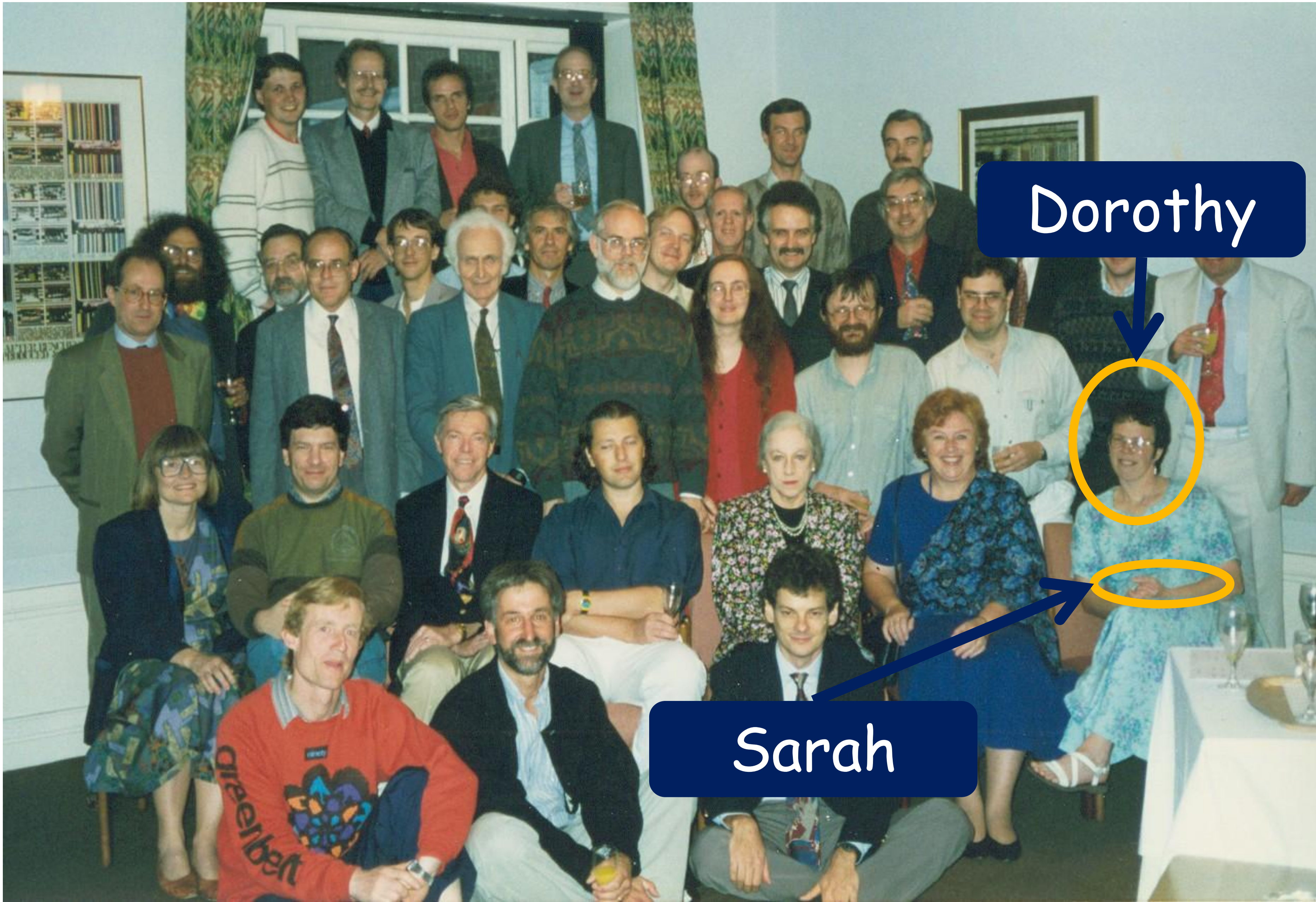
2003-2007 Growth spurt

WG2.8 June 1992

WG2.8 June 1992



WG2.8 June 1992




Haskell the cat (b. 2002)

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

Haskell 98

Haskell 98

- Stable
- Documented
- Consistent across implementations
- Useful for teaching, books

Haskell + extensions

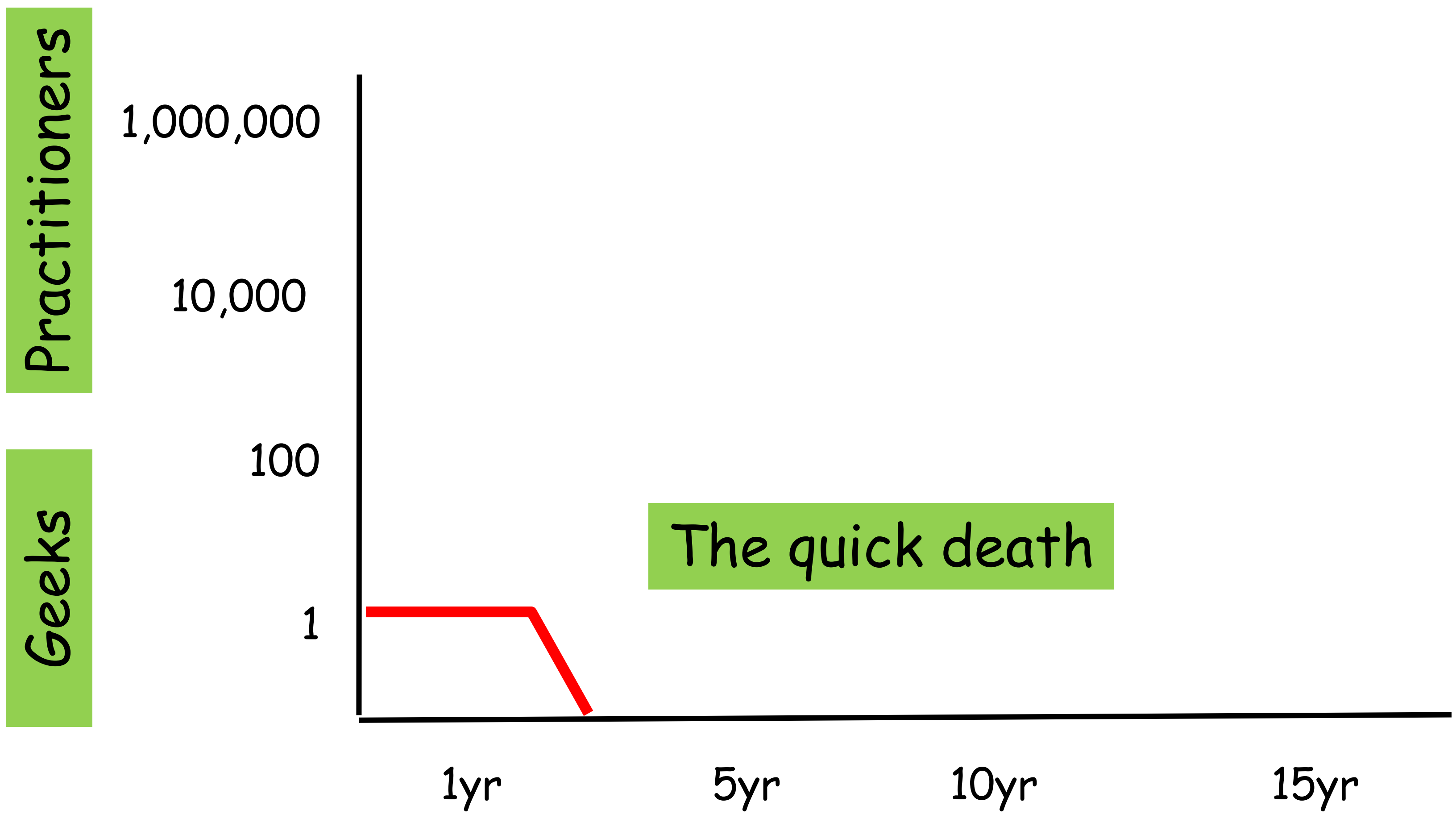
- Dynamic, exciting
- Unstable, undocumented, implementations vary...

Timeline

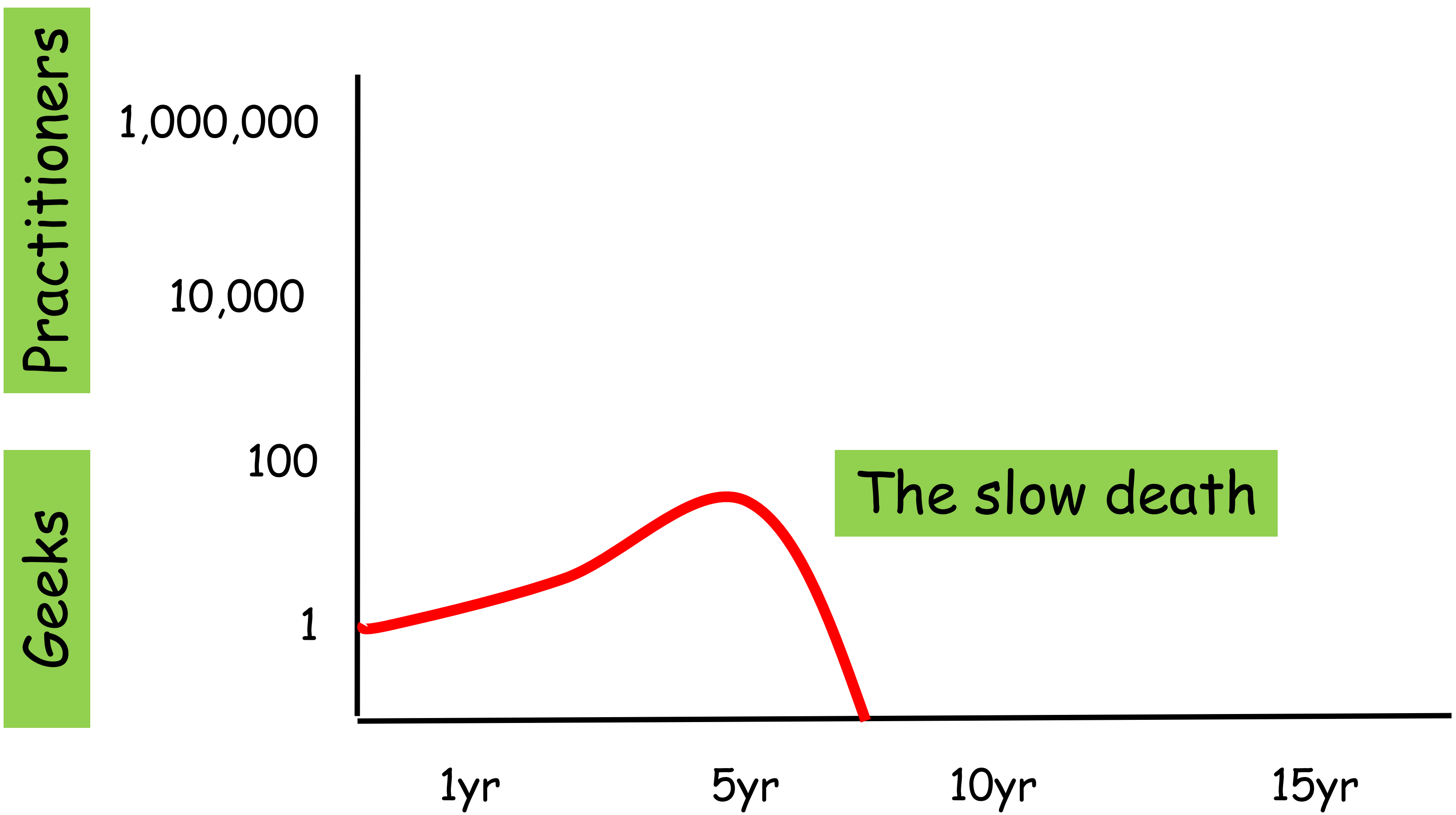
- Sept 87: **kick off**
- Apr 90: **Haskell 1.0**
- Aug 91: **Haskell 1.1 (153pp)**
- May 92: **Haskell 1.2 (SIGPLAN Notices) (164pp)**
- May 96: **Haskell 1.3. Monadic I/O, separate library report**
- Apr 97: **Haskell 1.4 (213pp)**
- Feb 99: **Haskell 98 (240pp)**
- Dec 02: **Haskell 98 revised (260pp)**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

The Book!
(thank you CUP)

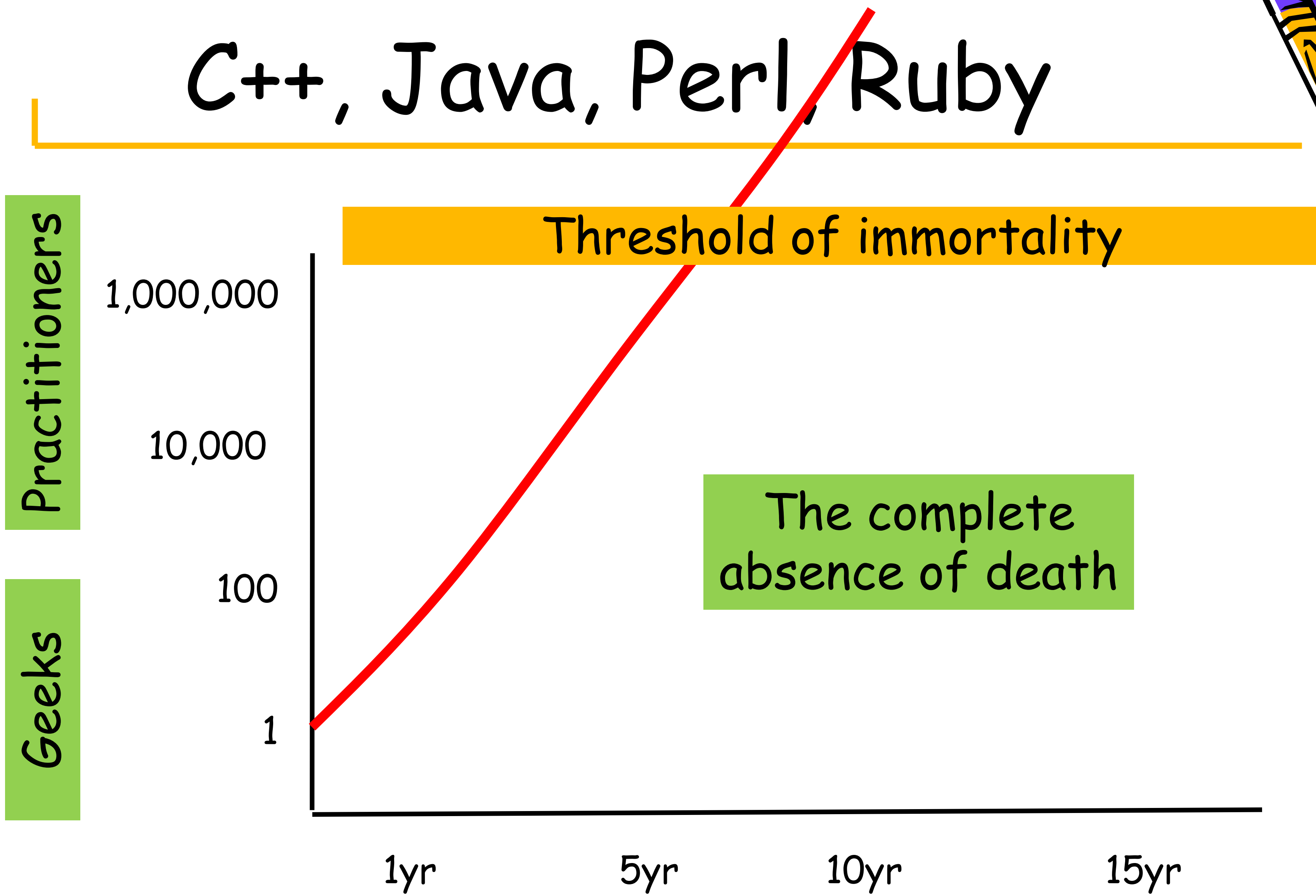
History of most research languages



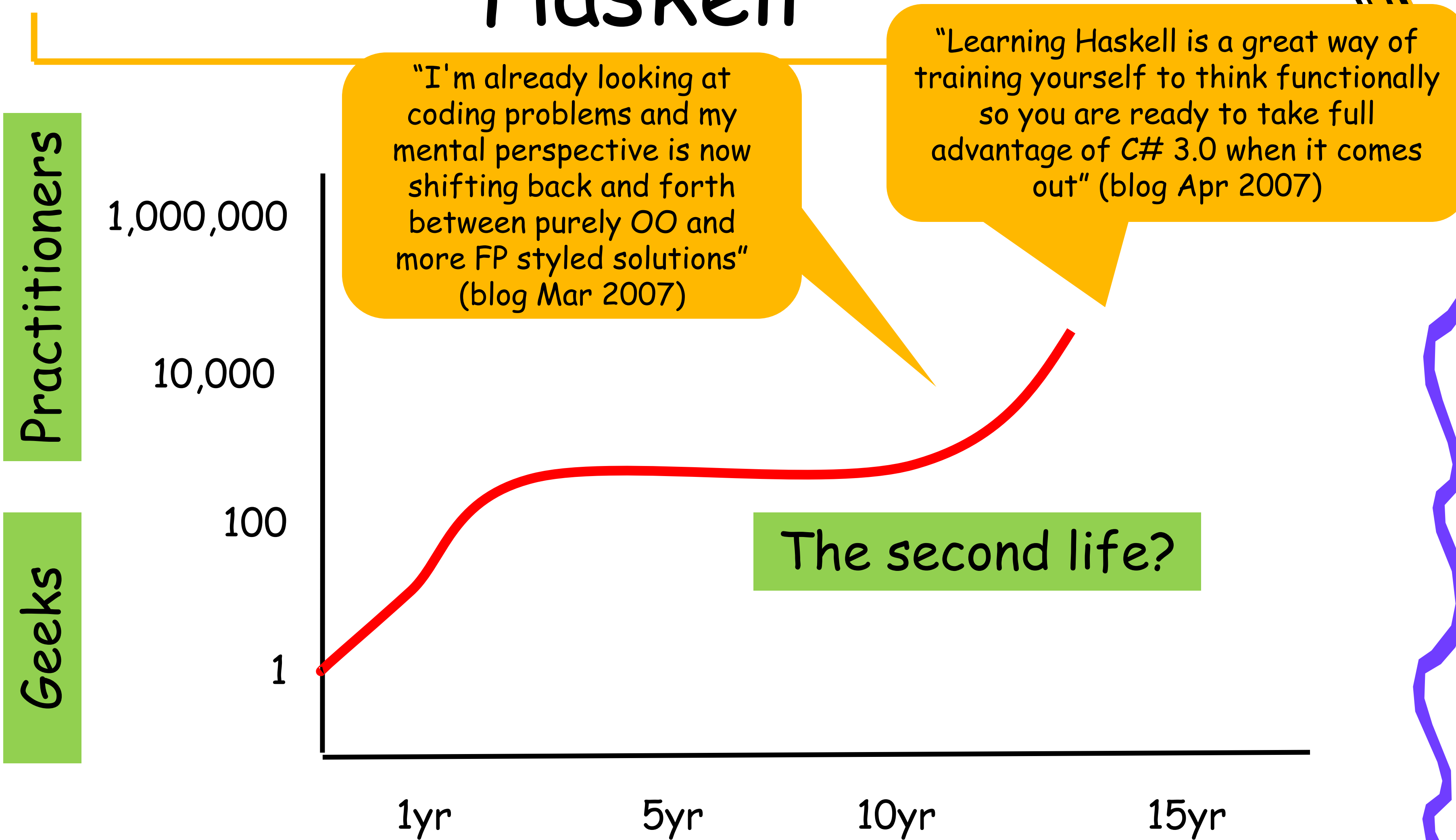
Successful research languages



C++, Java, Perl, Ruby

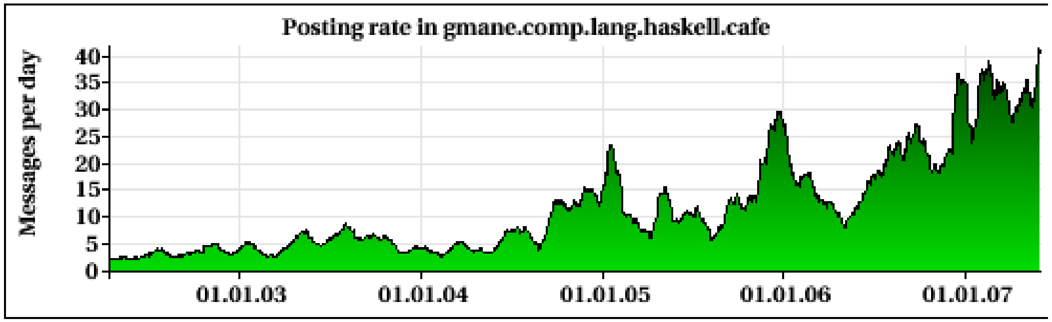


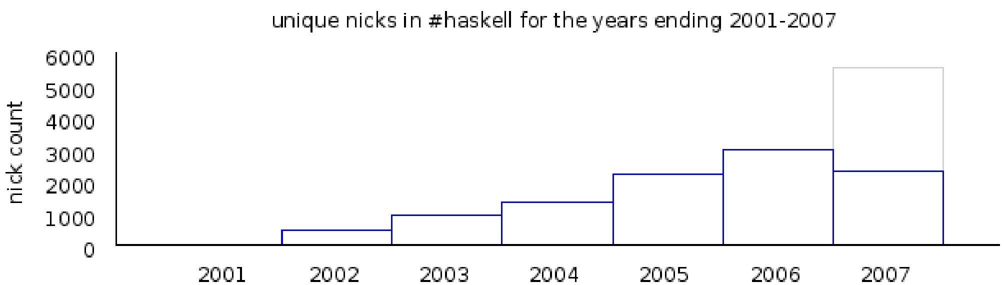
Haskell

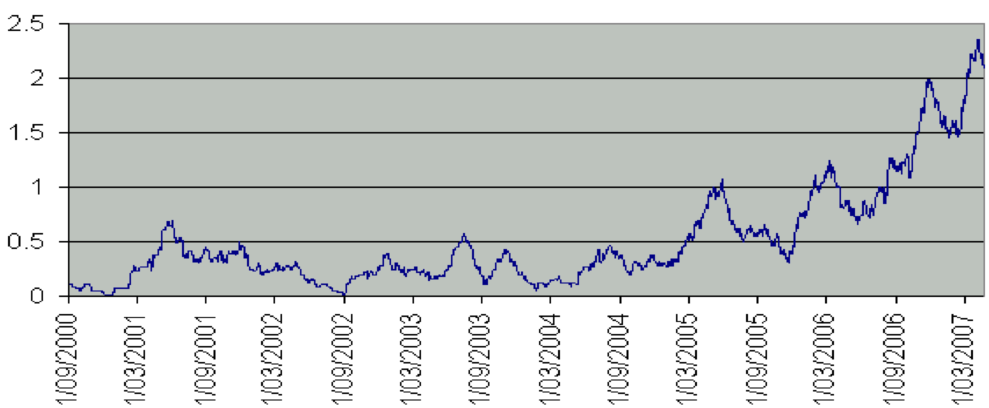


“I'm already looking at coding problems and my mental perspective is now shifting back and forth between purely OO and more FP styled solutions” (blog Mar 2007)
“Learning Haskell is a great way of training yourself to think functionally so you are ready to take full advantage of C# 3.0 when it comes out” (blog Apr 2007)
Practitioners
Geeks
1,000,000
10,000
100
1
The second life?
1yr
5yr
10yr
15yr

2003-7: growth spurt







Haskell Cafe (2000)

Haskell Communities Report (2001)

IRC #Haskell (2001)

Haskell Weekly News (2005)

Google Summer of Code (2006)

GHC bug count climbs

When Haskell is dust, what will it be remembered for?

1. Purity and laziness
2. Type classes
3. Process and community

Purity and laziness

Laziness

- Laziness was Haskell's initial rallying cry
- John Hughes's famous paper "Why functional programming matters"
  - Modular programming needs powerful glue
  - Lazy evaluation enables new forms of modularity; in particular, separating *generation* from *selection*.
  - Non-strict semantics means that unrestricted beta substitution is OK.

But...

- Laziness makes it much, much harder to reason about performance, especially space. Tricky uses of seq for effect `seq :: a -> b -> b`
- Laziness has a real implementation cost
- Laziness can be added to a strict language (although not as easily as you might think)
- And it's not so bad only having βV instead of β

So why wear the hair shirt of laziness?

Laziness keeps you pure

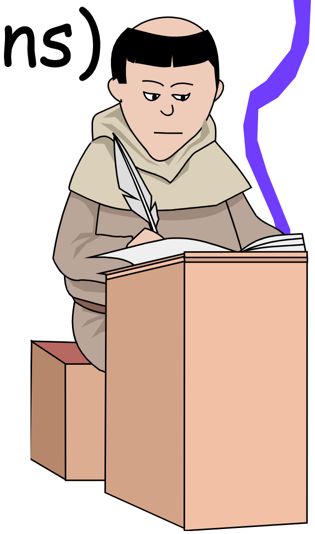
- Every call-by-value language has given into the siren call of side effects
- But in Haskell

```
(print "yes") + (print "no")
```

just does not make sense. Even worse is

```
[print "yes", print "no"]
```

- So effects (I/O, references, exceptions) are just not an option.
- 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.

eg.

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

Connecting I/O operations

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

eg.

```
getChar   >>= (\a ->
getChar   >>= (\b ->
putChar b >>= (\() ->
return (a,b))))
```

The do-notation

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

==

```
do {
  a <- getChar;
  b <- getChar;
  putchar b;
  return (a,b)
}
```

- 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 }
```

e.g. `repeatN 10 (putChar 'x')`

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...

=

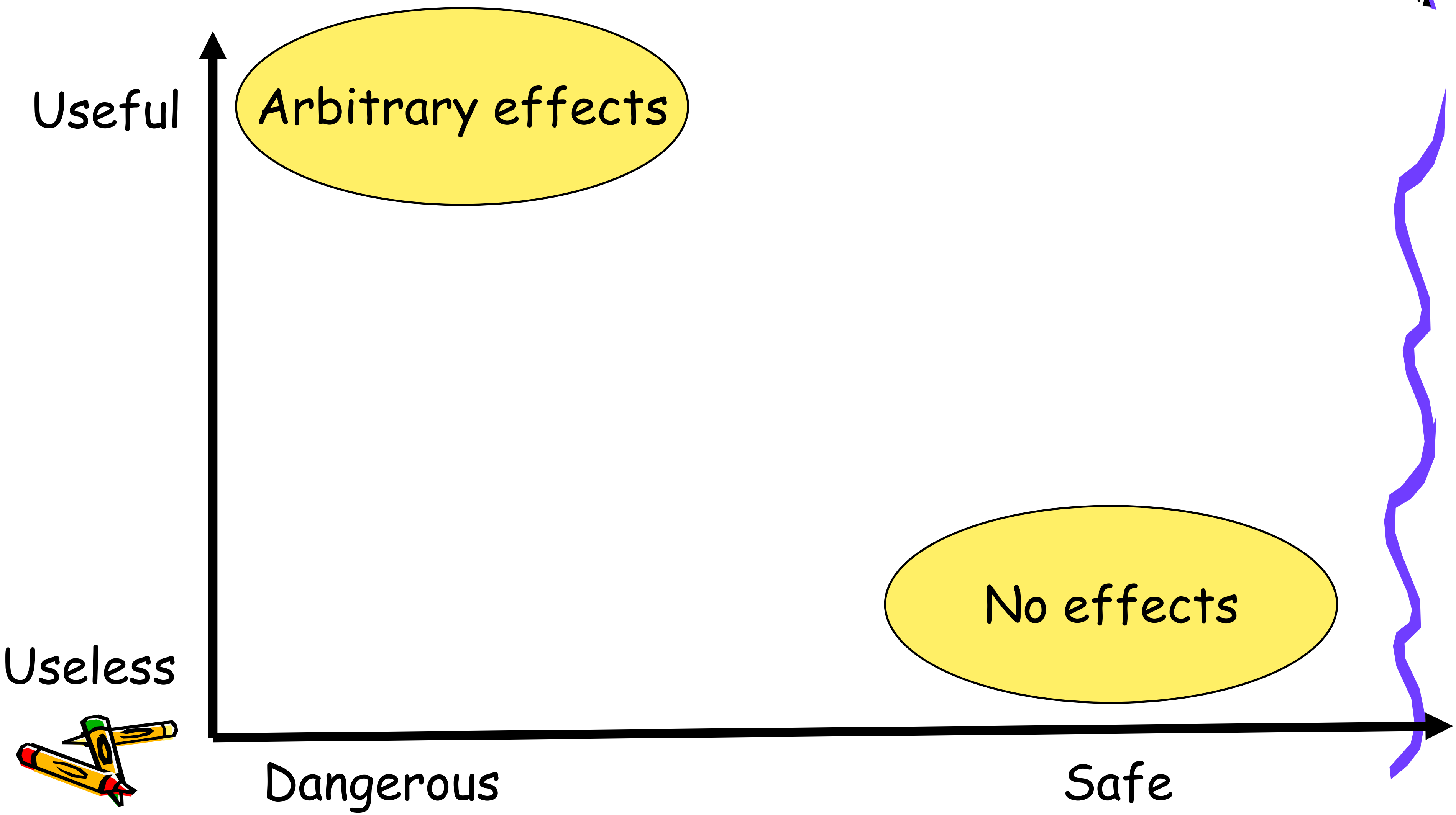
....e....e.....

Fine grain control

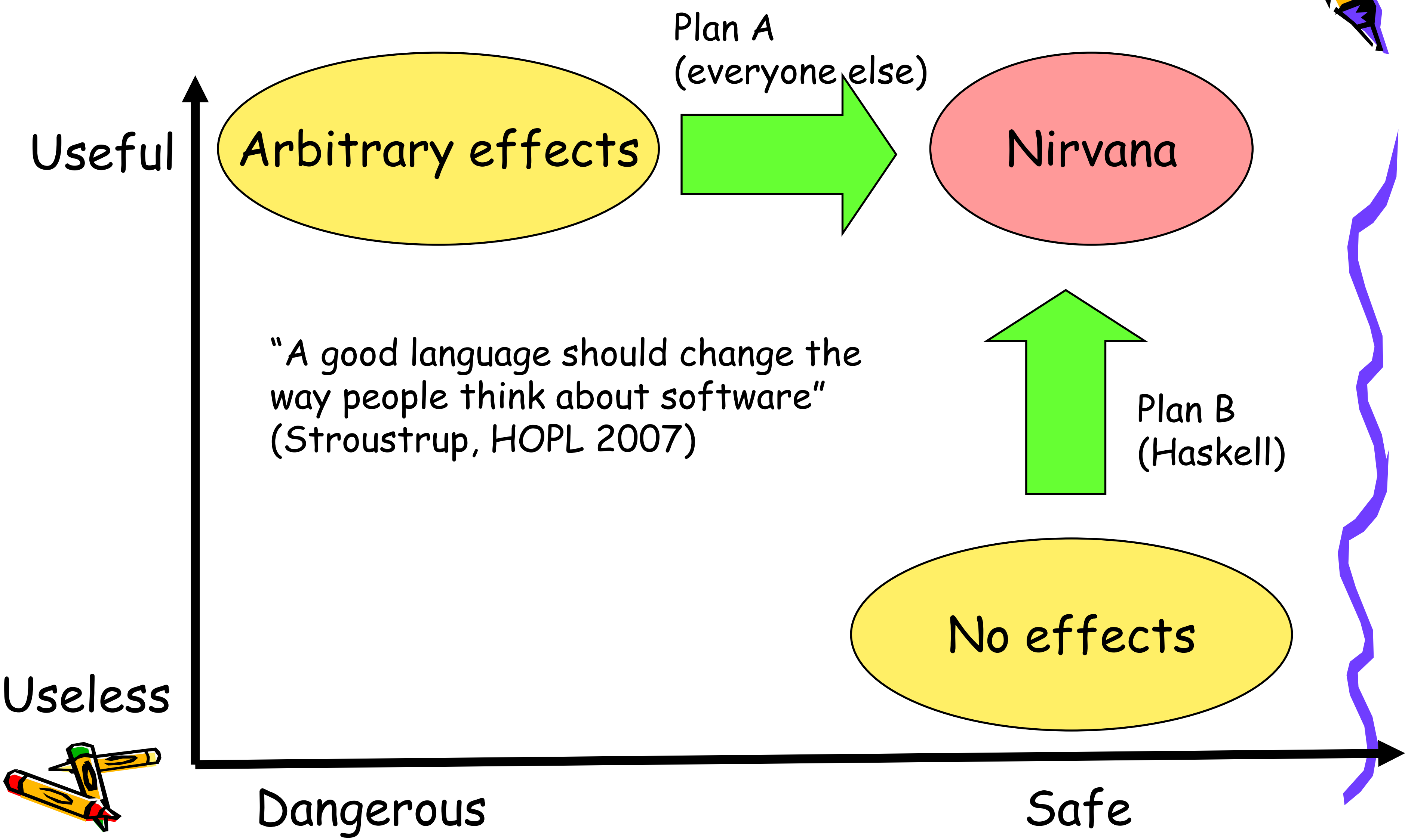
- reverse :: String -> String
  - pure: no side effects
- launchMissiles :: String -> IO [String]
  - impure: international side effects
- transfer :: Acc -> Acc -> Int -> STM ()
  - transactional: limited effects (reading and writing transactional variables

There are lots of useful monads, not only I/O

The central challenge



The challenge of effects



Two basic approaches: Plan A

Arbitrary effects

Default = Any effect
Plan = Add restrictions

- Types play a major role
- Types blur into analyses

Examples

- Regions
- Ownership types
- Vault, Spec#, Cyclone, etc etc

Two basic approaches: Plan B

Default = No effects
Plan = Selectively permit effects

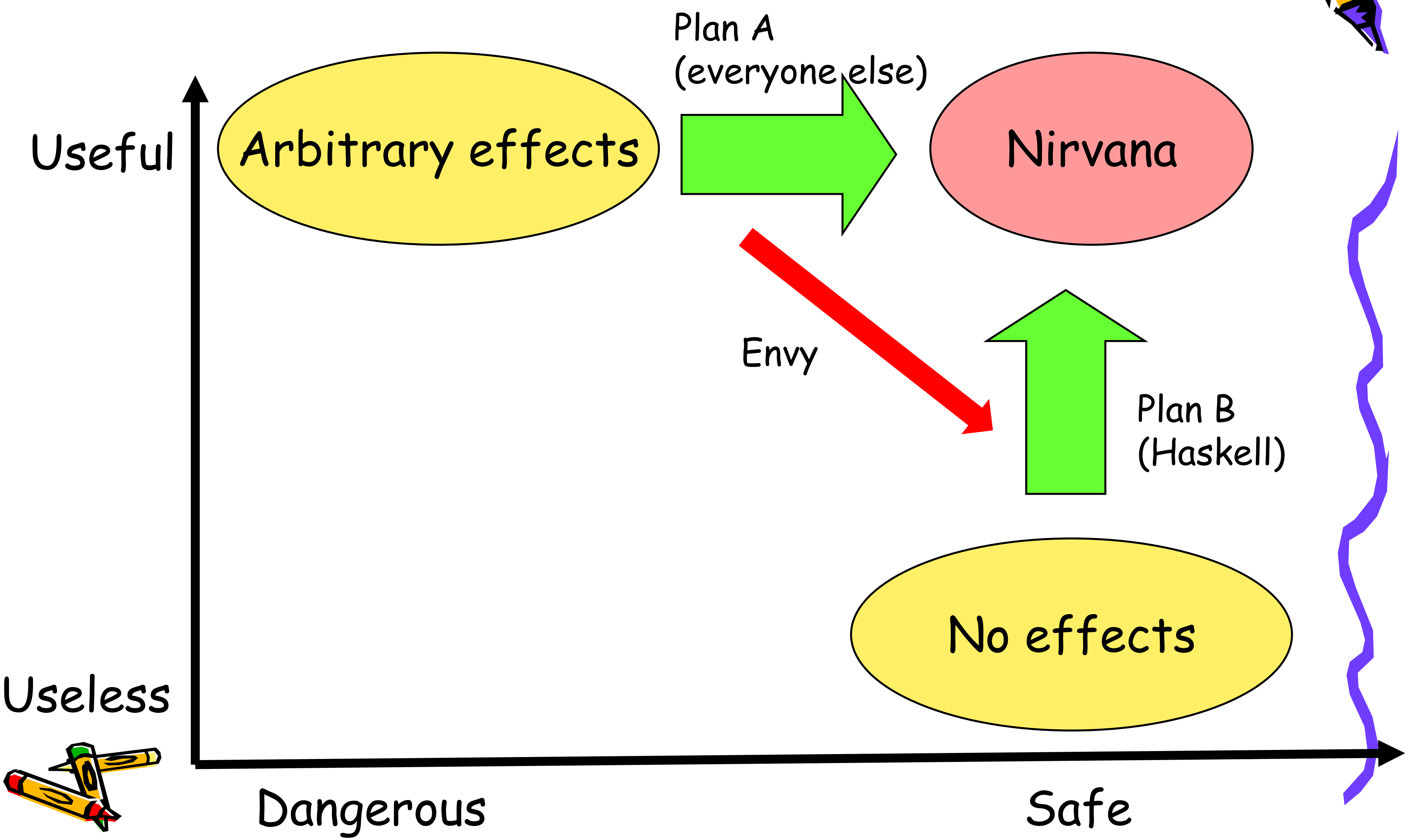
Again, types play a major role

Two main approaches:

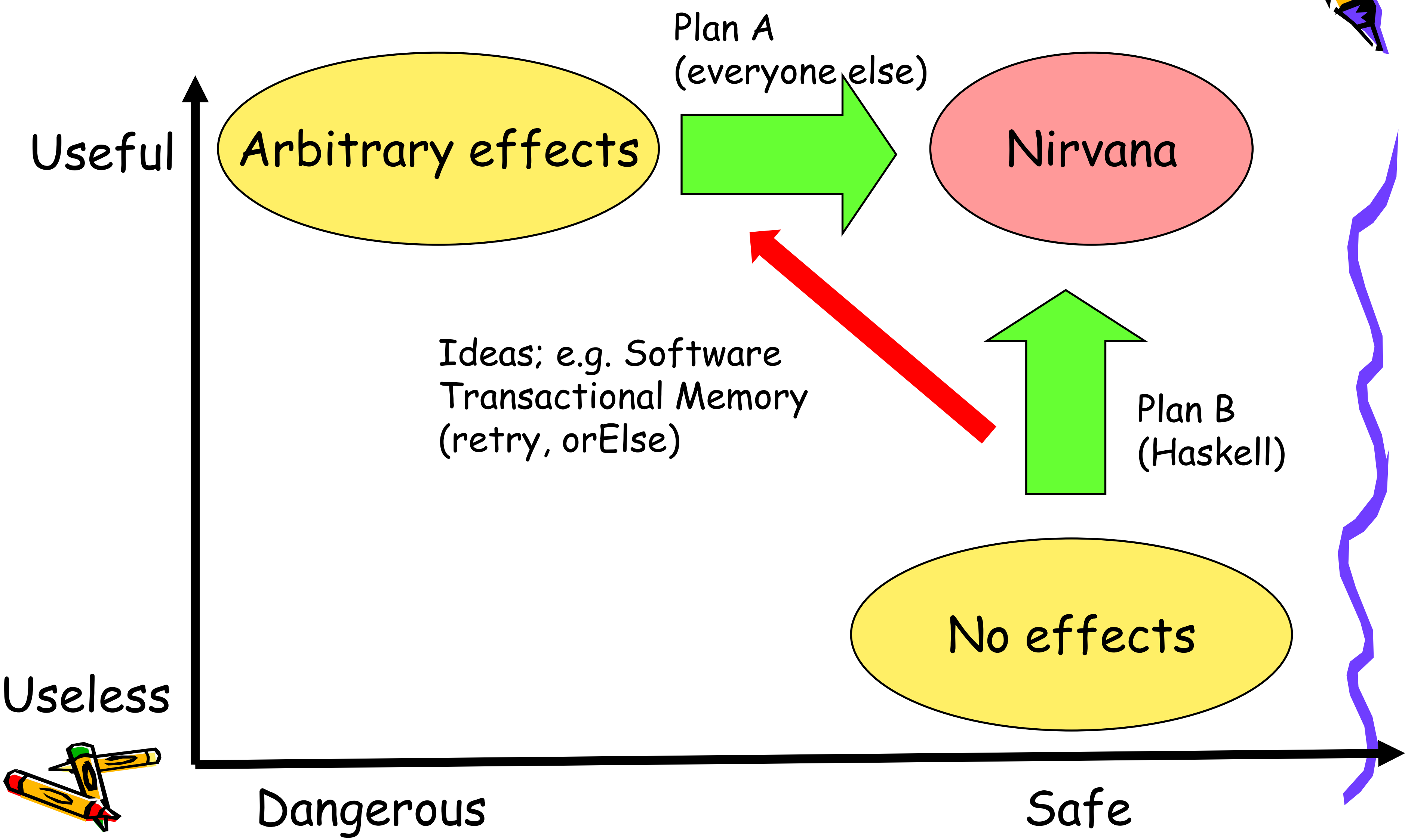
- Domain specific languages (SQL, XQuery, MDX, Google map/reduce)
- Wide-spectrum functional languages + controlled effects (e.g. Haskell)

Value oriented programming

Lots of cross-over



Lots of cross-over



SLPJ conclusions

- One of Haskell's most significant contributions is to take purity seriously, and relentlessly pursue Plan B
- The next ML will be pure, with effects only via monads. The next Haskell will be strict, but still pure.
- Imperative languages will embody growing (and checkable) pure subsets

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
```

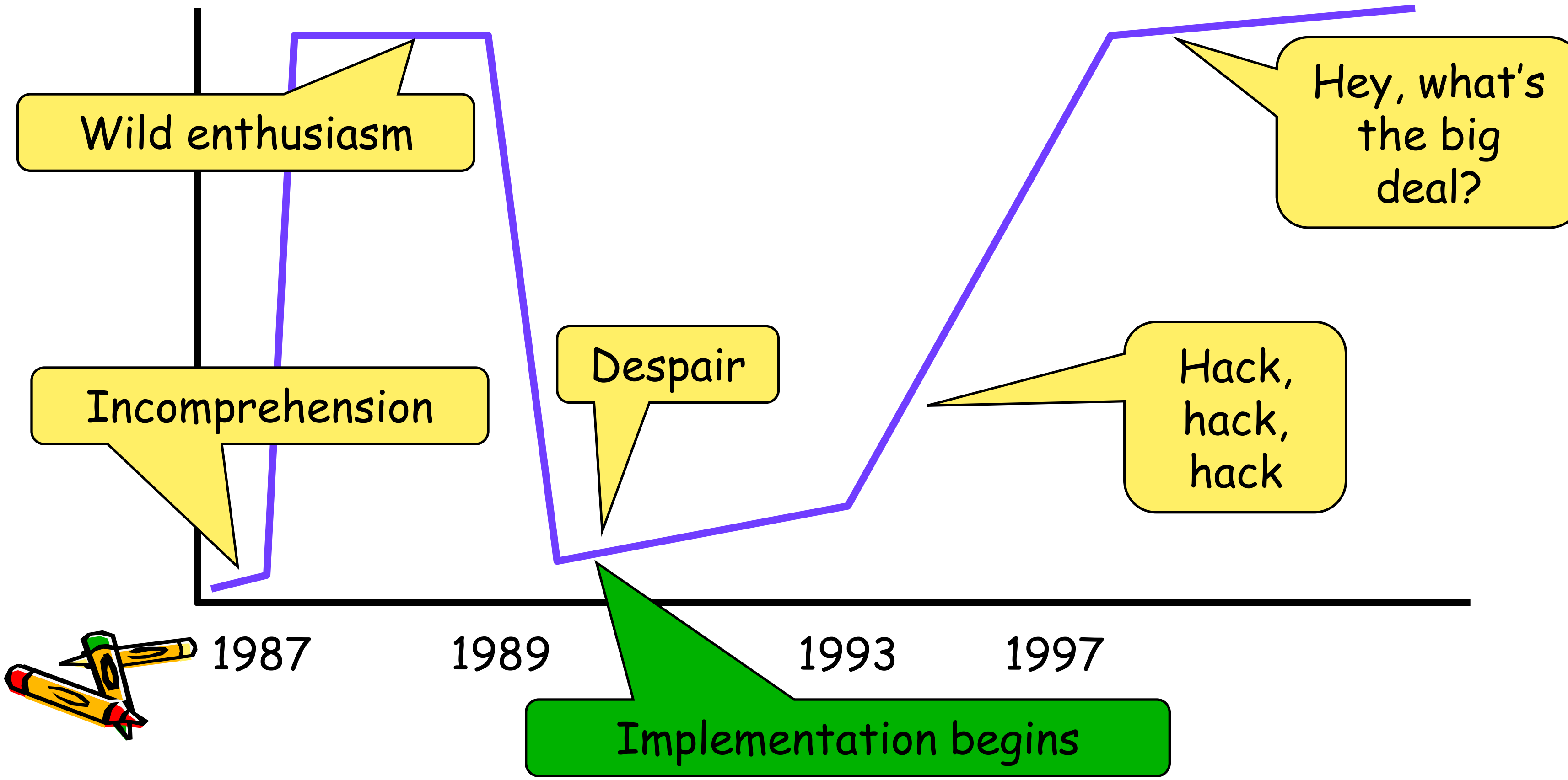
Class witnessed by a "dictionary" of methods

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
```

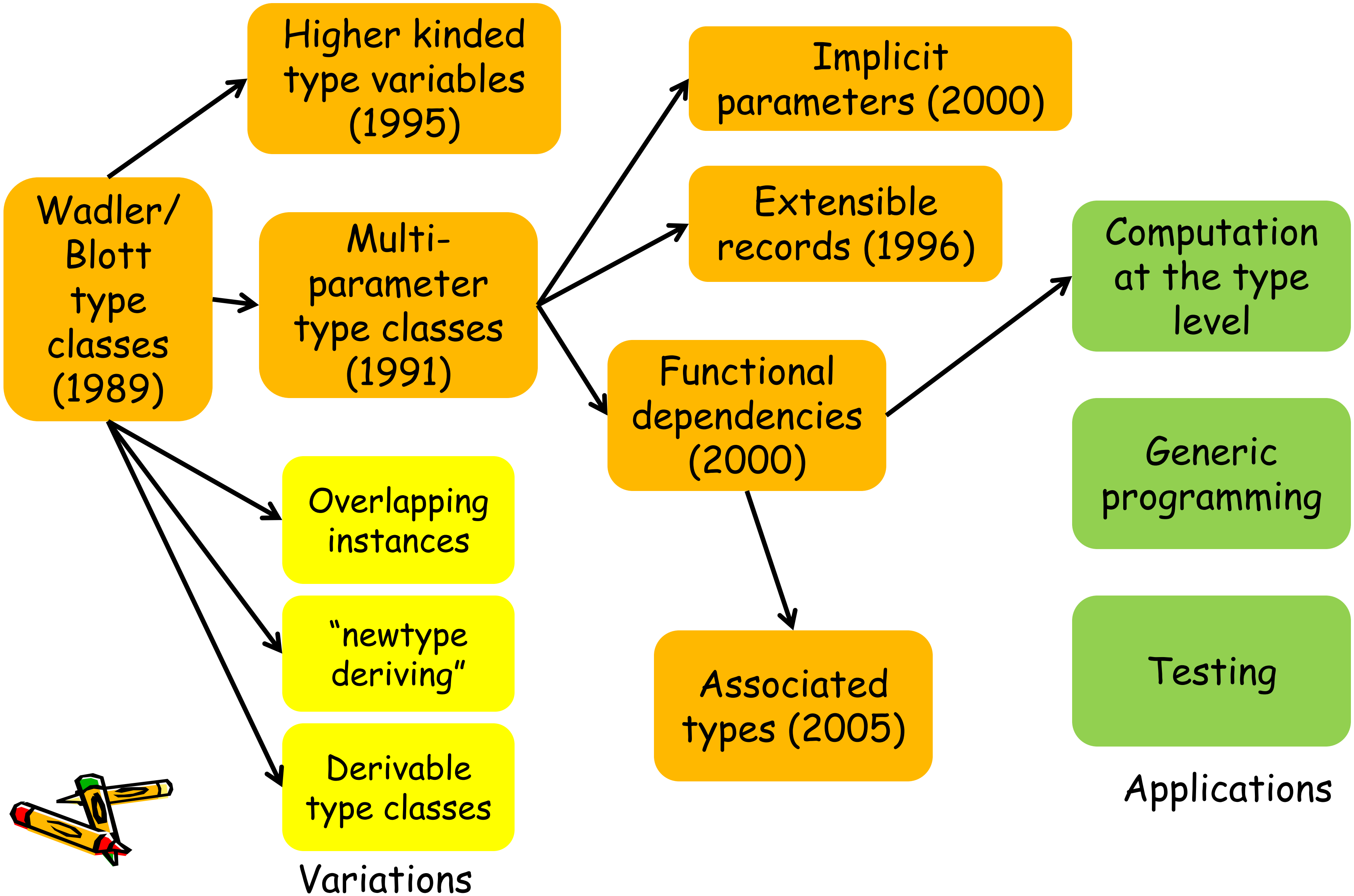
Quickcheck (which is just a Haskell 98 library)

- 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

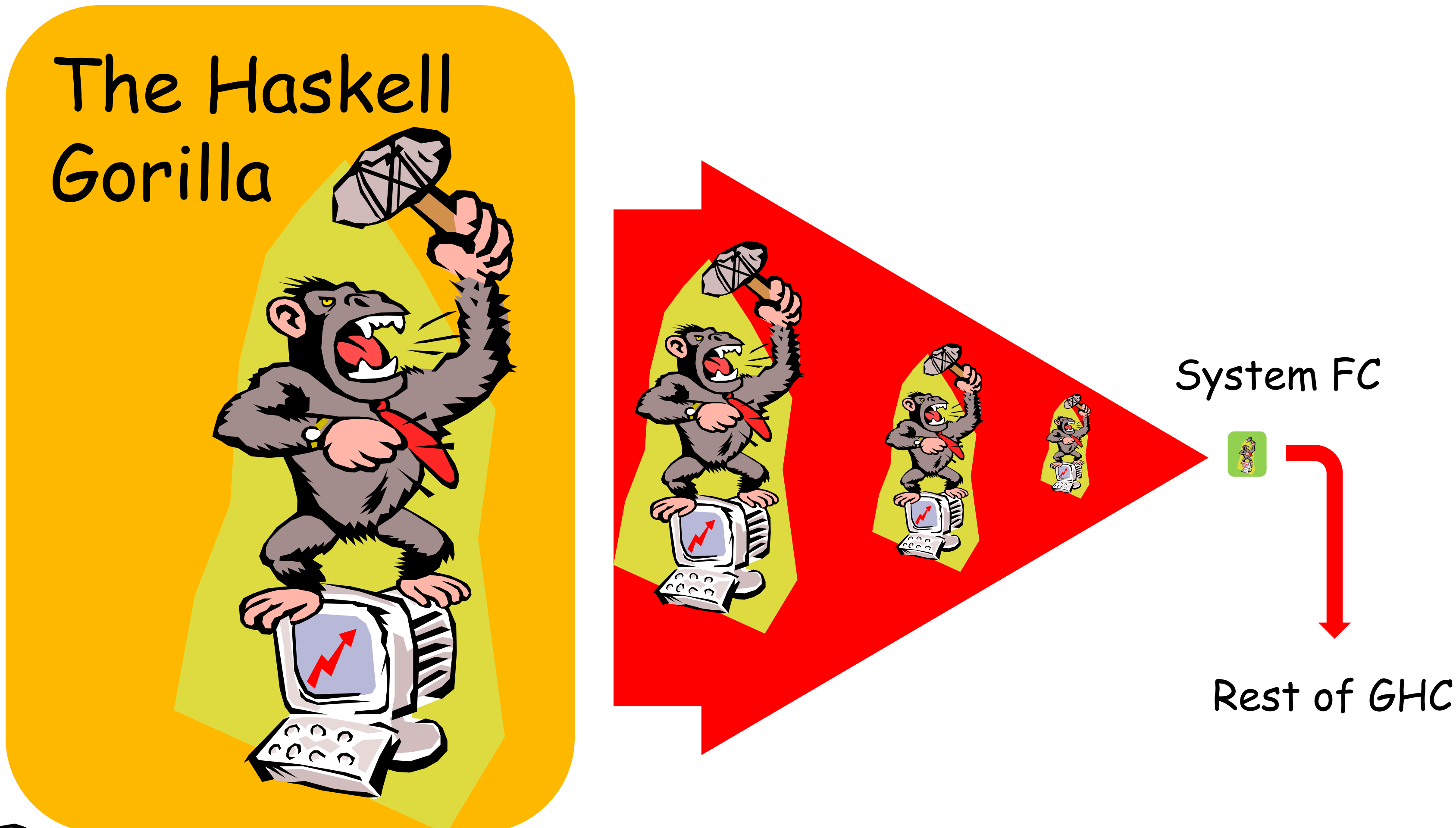
"Deeply held principles"

- 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