Escape from the ivory tower
The Haskell journey

Simon Peyton Jones, Microsoft Research

1976-80

John and Simon go to university

Early days of microprocessors

4kbytes is a lot of memory

Cambridge University has one (1) computer

The late 1970s, early 1980s

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

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

SK combinators, graph reduction (Turner)

Lambda the Ultimate (Steele, Sussman)

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

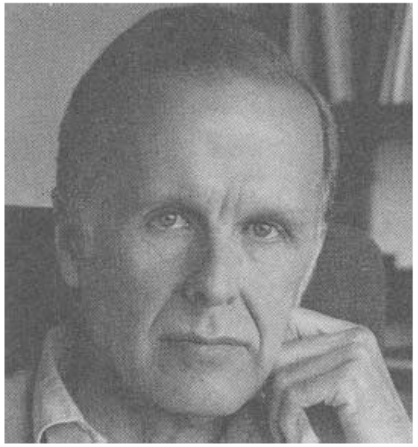
Dataflow architectures (Dennis, Arvind et al)

Lisp machines (Symbolics, LMI)

Backus Turing Award 1977

Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs

John Backus
IBM Research Laboratory, San Jose



Conventional programming languages are growing ever more enormous, but not stronger. Inherent defects at the most basic level cause them to be both fat and weak: their primitive word-at-a-time style of programming inherited from their common ancestor—the von Neumann computer, their close coupling of semantics to state transitions, their division of programming into a world of expressions and a world of statements, their inability to effectively use powerful combining forms for building new programs from existing ones, and their lack of useful mathematical properties for reasoning about programs.

John Backus Dec 1924 – Mar 2007

The Call

nal
g
se,
Turner)

mbinators,
reduction
urner)

Have no truck with the grubby compromises of imperative programming!

Go forth, follow the Path of Purity, and design new languages and new computers and rule the world

"Because we all want to build our own language and VM"
Cameron Purdy

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.

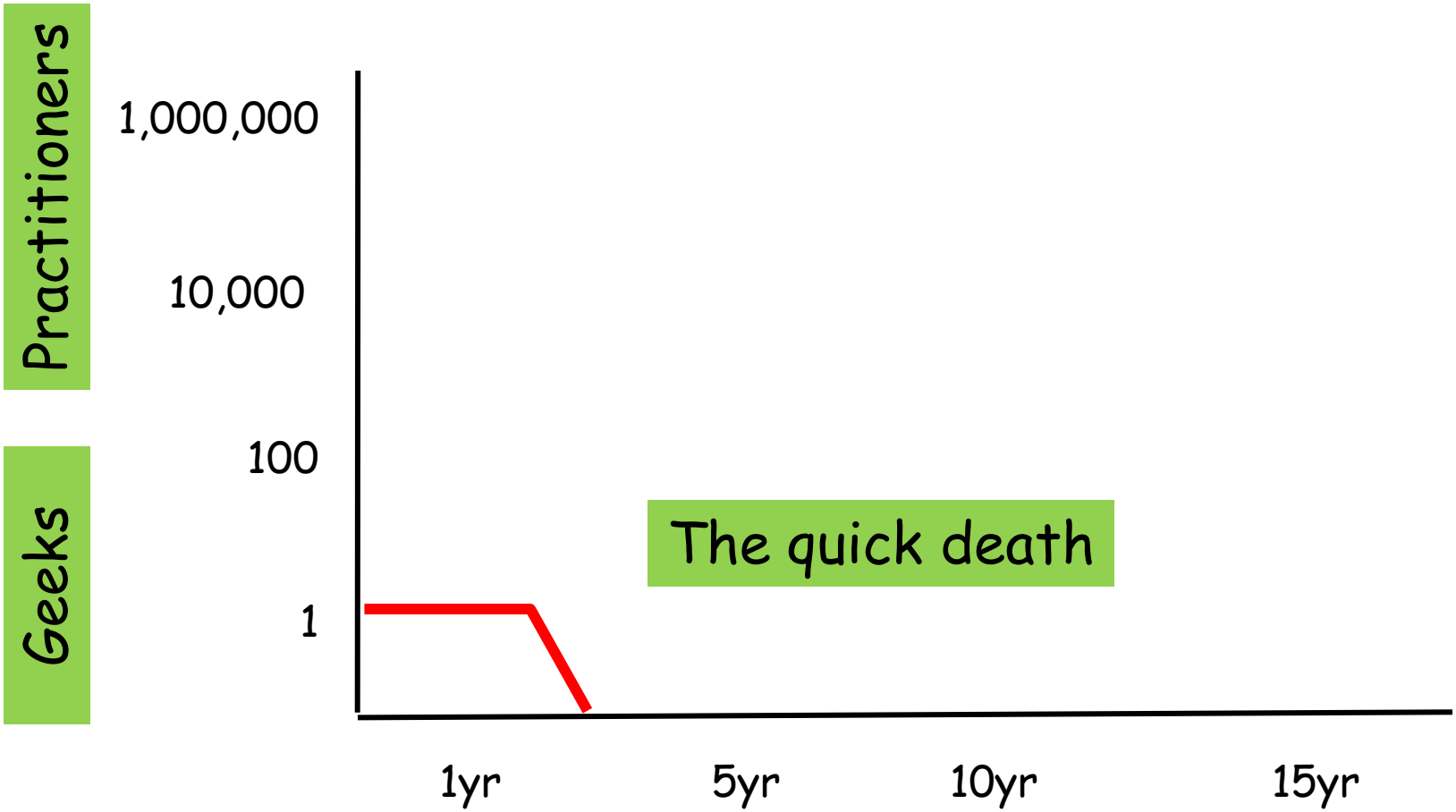
- Suitable for teaching, research, and application
- Formally-described syntax and semantics
- Freely available
- Embody the apparent consensus of ideas
- Reduce unnecessary diversity

Absolutely no clue how much work we were taking on

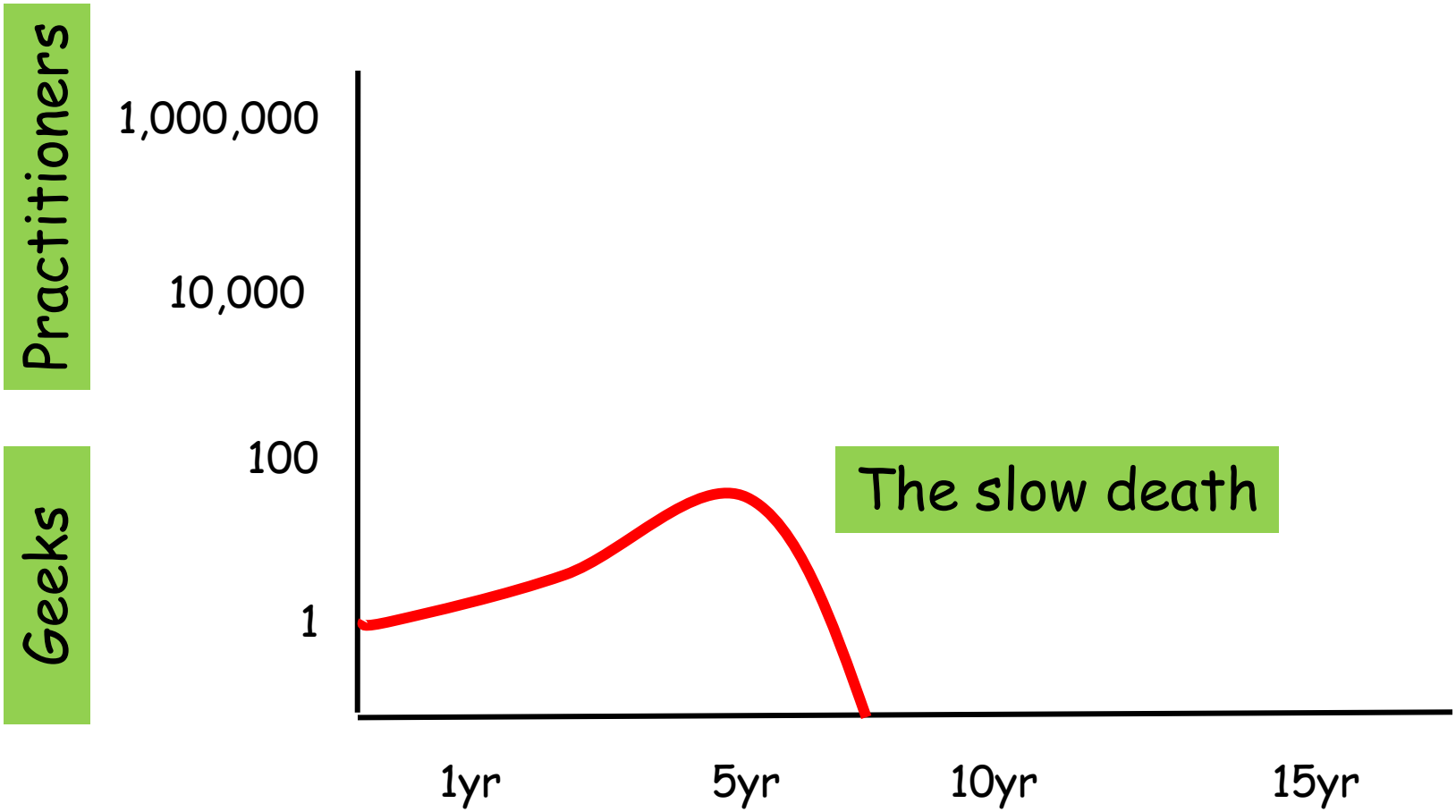
Led to...a succession of face-to-face meetings

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

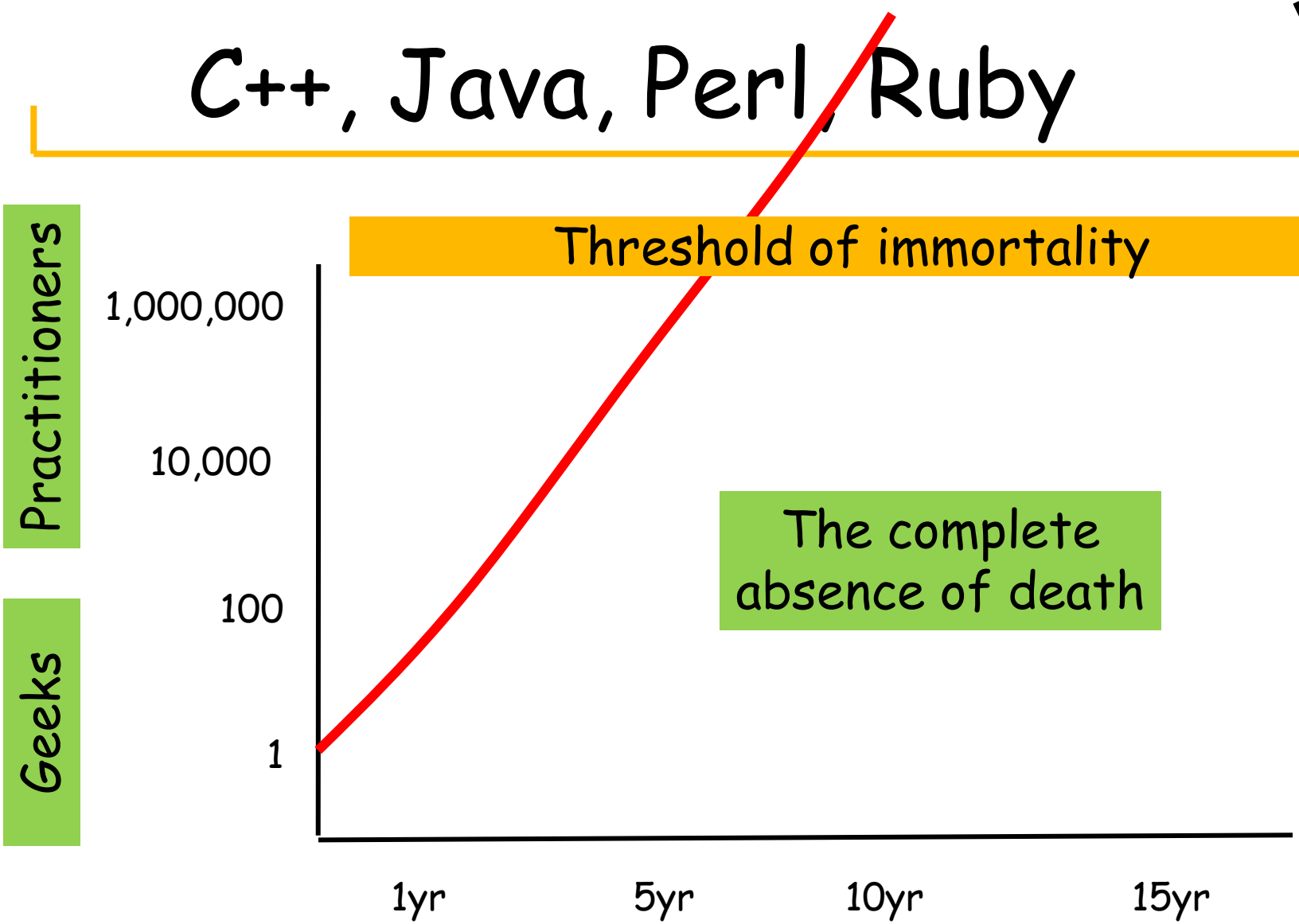
History of most research languages



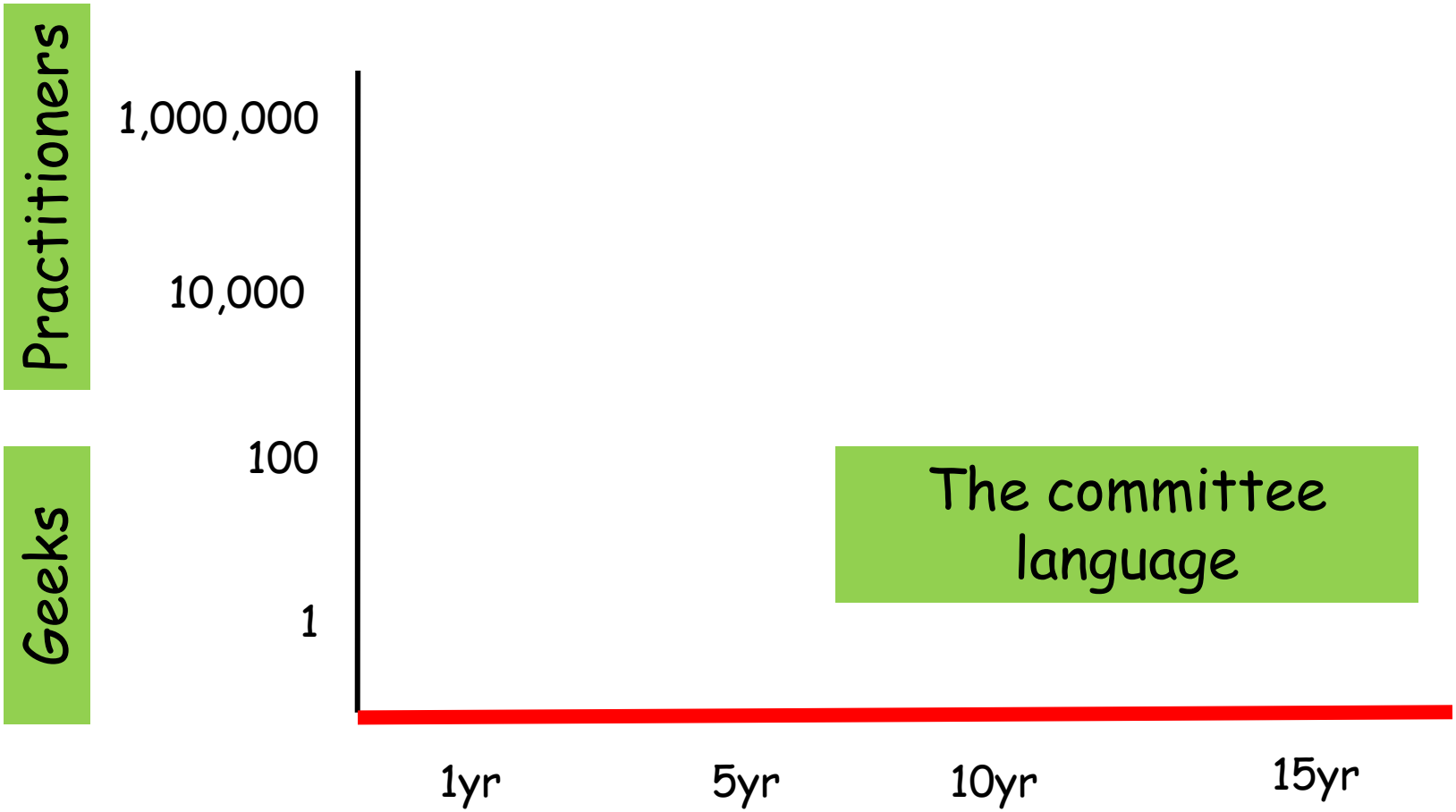
Successful research languages



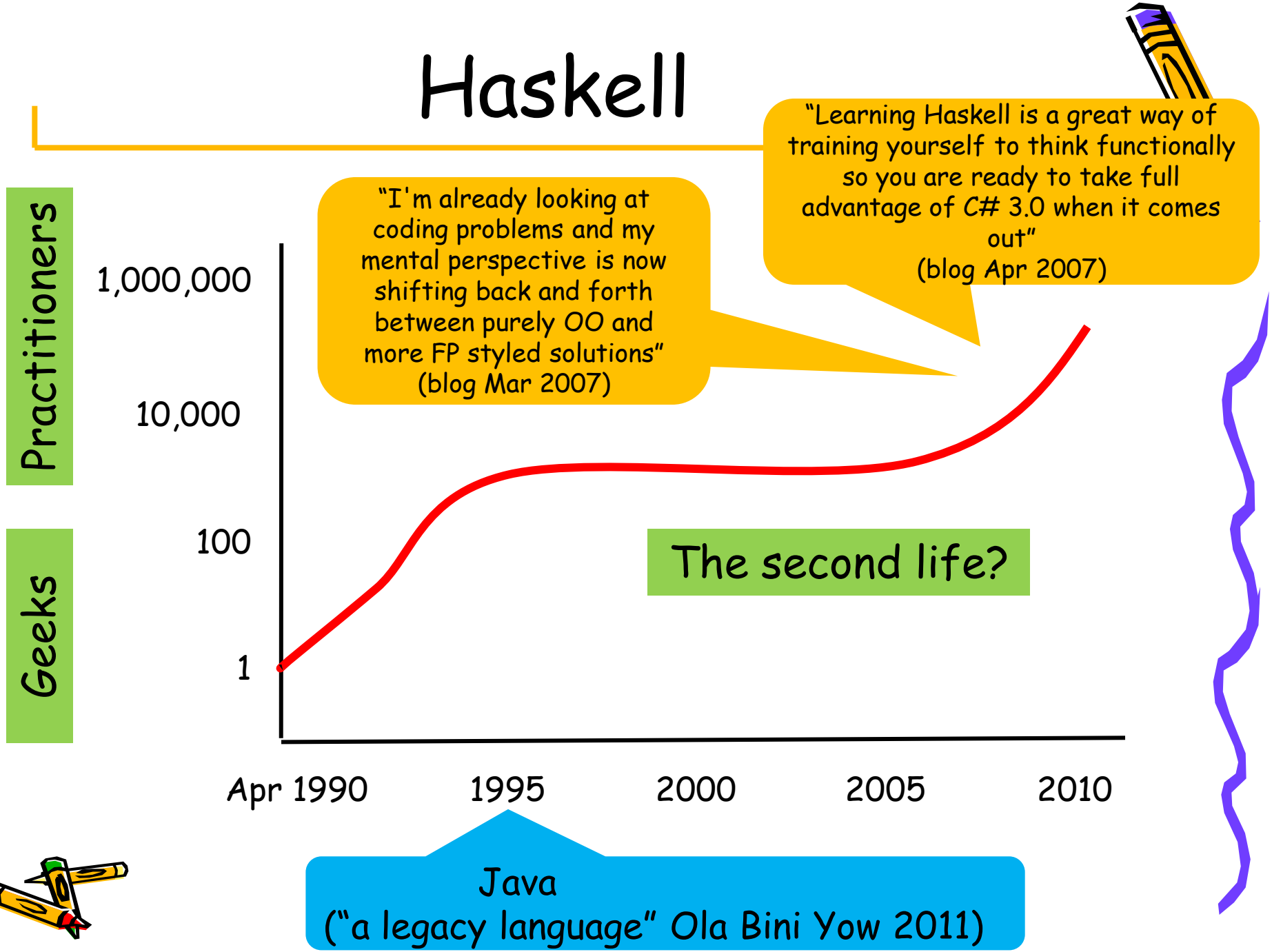
C++, Java, Perl, Ruby



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?
Apr 1990
1995
2000
2005
2010
Java
("a legacy language" Ola Bini Yow 2011)

Haskell is 21; so is Michael

Michael (b 1990)

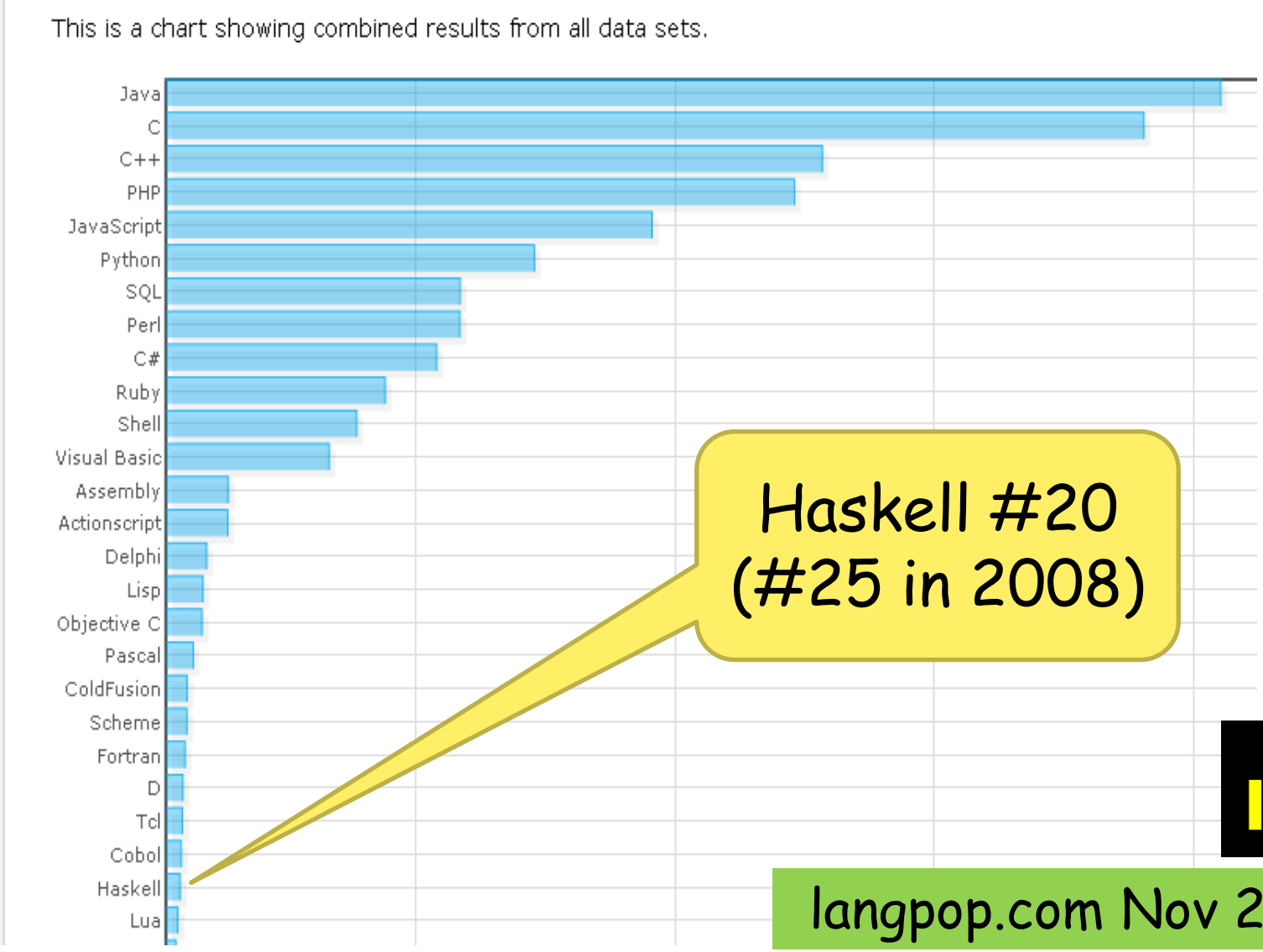

Haskell the cat (b. 2002)

WG2.8 June 1992

WG2.8 June 1992

WG2.8 June 1992

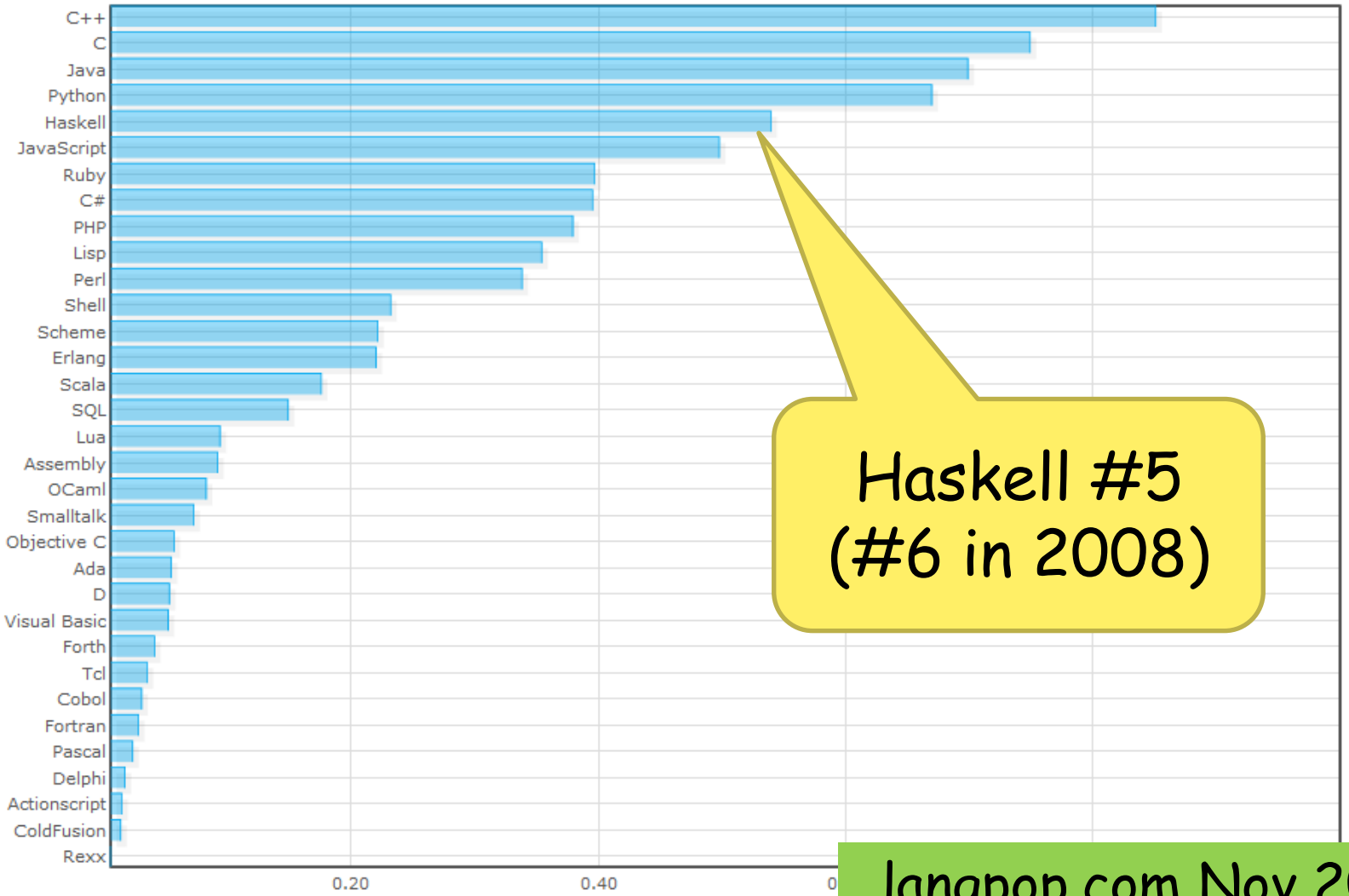
Language popularity
how much language X is used



Haskell #20
(#25 in 2008)

langpop.com Nov 2011

Language popularity
how much language X is talked about



Haskell #5
(#6 in 2008)

langpop.com Nov 2011

Incredibly supportive community

Firefox

haskell.org/haskellwiki/Haskell

Log in / create account

Go Search

View source History

The Haskell Programming Language

Haskell is an advanced purely-functional programming language. An open-source product of more than twenty years of cutting-edge research, it allows rapid development of robust, concise, correct software. With strong support for integration with other languages, built-in concurrency and parallelism, debuggers, profilers, rich libraries and an active community, Haskell makes it easier to produce flexible, maintainable, high-quality software.

Learn Haskell

- What is Haskell?
- Try Haskell in your browser
- Learning resources
- Books & tutorials
- Library documentation

Use Haskell

- Download Haskell
- Language specification
- Hackage library database
- Applications and libraries
- Hoogle and Hayoo API search

Join the Community

- Haskell on Reddit, Stack Overflow
- Mailing lists, IRC channels
- Wiki (how to contribute)
- Communities and Activities Reports
- Haskell in industry, research and education.
- Planet Haskell, The Monad.Reader

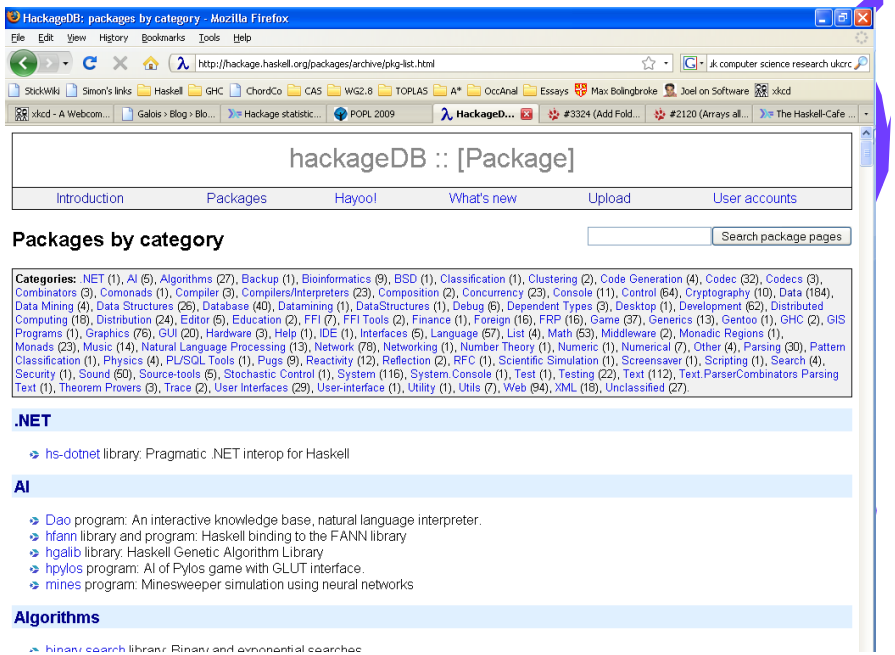
Find: taste Next Previous Highlight all Match case

Mobilising the community

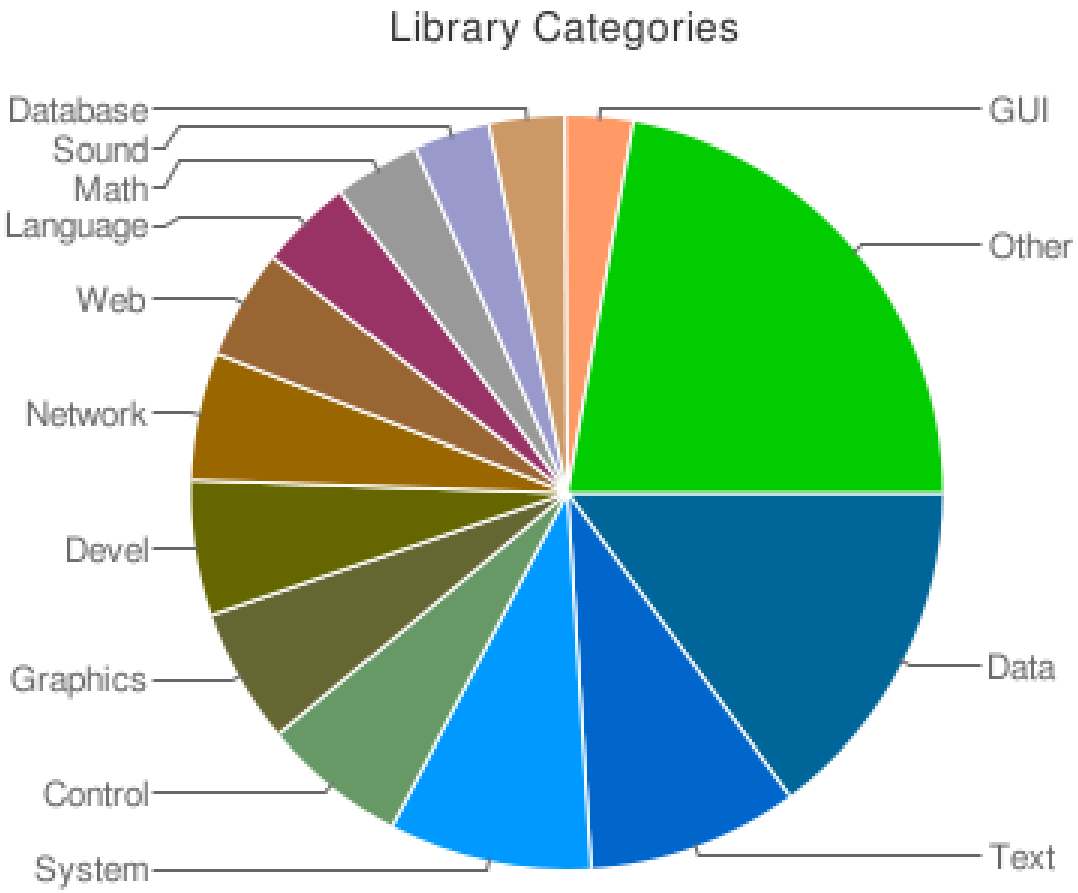
- Package = unit of distribution
- **Cabal**: simple tool to install package and all its dependencies

```
bash$ cabal install pressburger
```

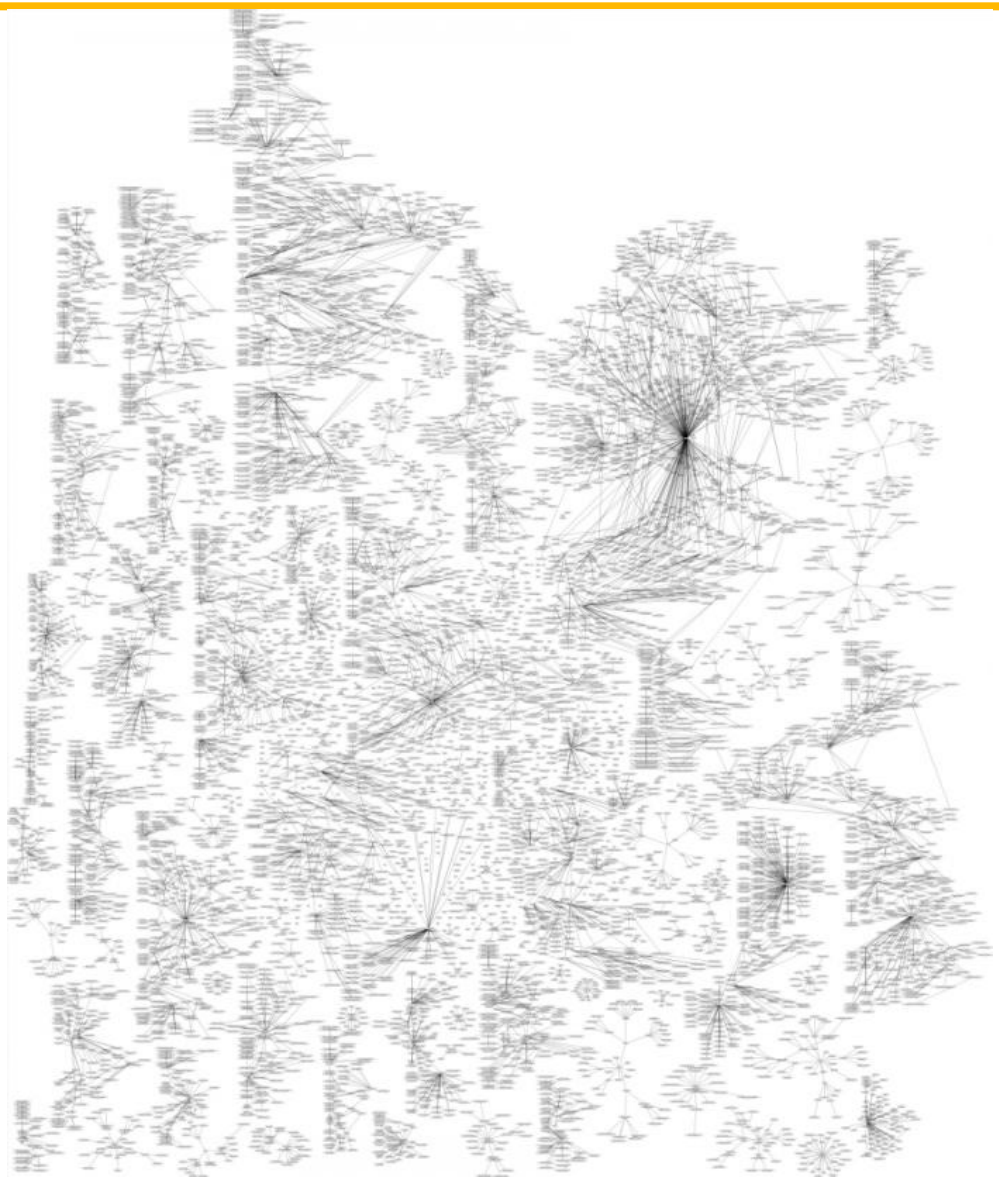
- **Hackage**: central repository of packages, with open upload policy



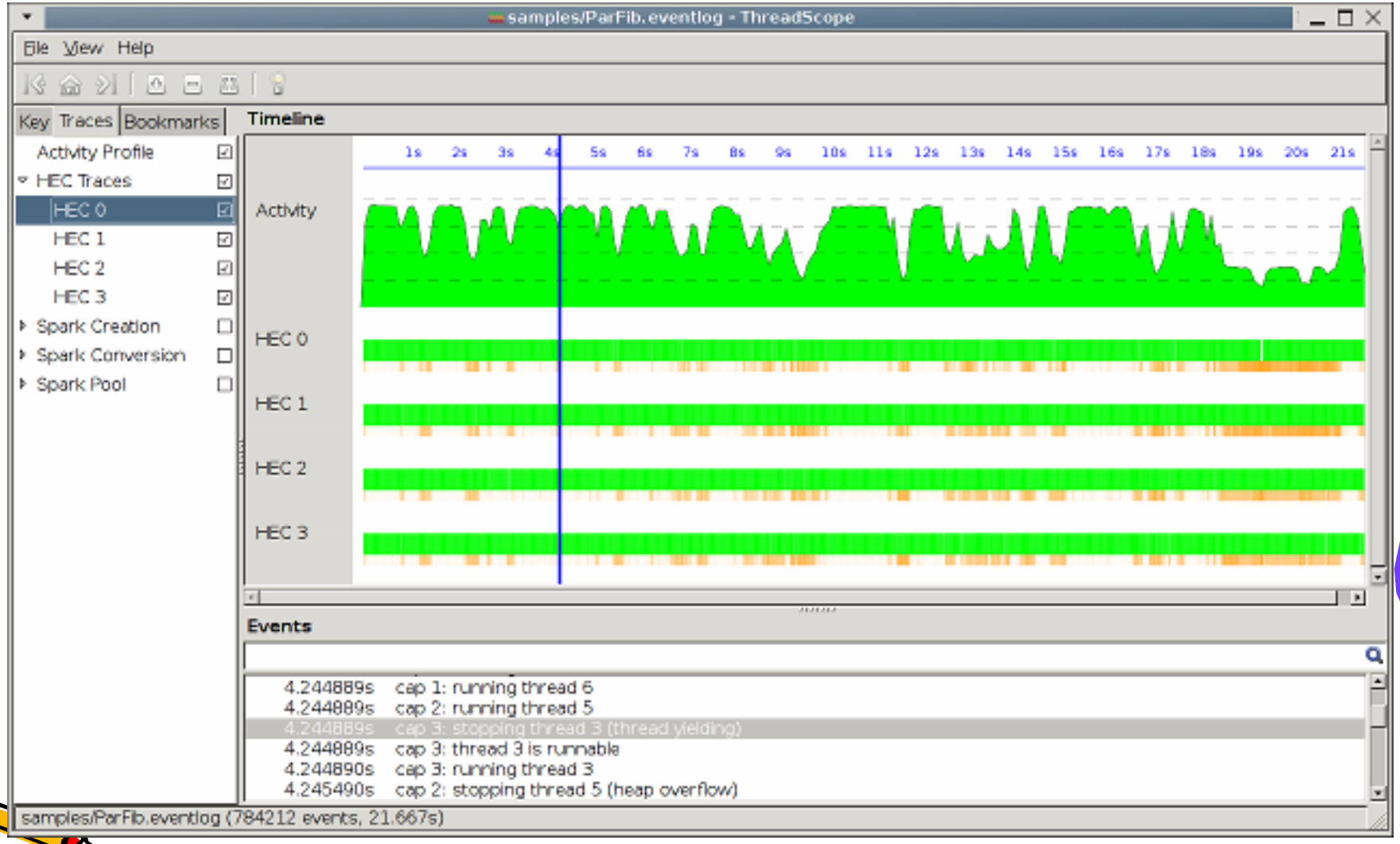
Now over 3,5000 packages on Hackage



The packages on Hackage



Tools: eg parallel profiler



The Glasgow Haskell Compiler

- GHC today
  - First release 1991: 13k lines, 110 modules, sequential
  - Now: 125k lines, 380 modules, parallel
- >> 100k users
- 100% open source (BSD)
- Still in furious development: > 200 commits/month

Commercial users

- High assurance systems (Galois, Mitre, NICTA)
- Controls systems (Eaton)
- Banks (lots)
- Electricity supply contracts (RWE), risk analysis (iba CG)
- Web frameworks/servers (HAppS, JanRain)
- Games (Joyride)
- Social networks (Peerium)

http://haskell.org/haskellwiki/Haskell_in_industry

After 21 years, Haskell has a vibrant, growing ecosystem, and is **still** in a ferment of new developments.

Why?

1. Keep faith with deep, simple principles
2. Killer apps:
   - domain specific languages
   - concurrent and parallel programming
3. Avoid success at all costs

"This is so simple I've wasted my entire life"
Steve Vinoski

Avoiding success

- A user base that is
  - Smallish: enough users to drive innovation, not so many as to stifle it
  - Tolerant. Very tolerant.
  - Innovative and slightly geeky: Haskell users react to new features like hyenas react to red meat
  - Extremely friendly

  makes Haskell nimble.
- Avoided the Dead Hand of standardisation committees

Deep, simple principles

Source language

Haskell

Dozens of types

100+ constructors

Intermediate language

System FC
3 types,
15 constructors

Rest of GHC

Deep simple 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
  | Type     Type
  | Coercion Coercion
  | Tick     Note Expr
data Bind   = NonRec Var Expr | Rec [(Var,Expr)]
data AltCon = DEFAULT | LitAlt Lit | DataAlt DataCon
```

System FC

$$
\begin{aligned}
e ::= \; & x \mid k \mid \tau \mid \gamma \\
& \mid e_1 \; e_2 \mid \backslash(x{:}\tau).e \\
& \mid \text{let bind in } e \\
& \mid \text{case } e \text{ of alts} \\
& \mid e \triangleright \gamma
\end{aligned}
$$

Everything has to translate into this tiny language
Fantastic language design sanity check

What deep, simple principles?

1. Purity and laziness
2. Types; especially type classes

Laziness and Purity

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 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

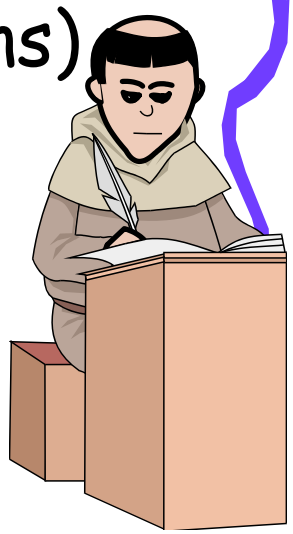
- Every call-by-value language has given into the siren call of side effects
- But in Haskell

```
(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



Laziness keeps you pure

- Even ... into

Comprehending Monads

Philip Wadler
University of Glasgow

Imperative functional programming

Simon L Peyton Jones Philip Wadler

Dept of Computing Science, University of Glasgow
Email: {simonpj,wadler}@dcs.glagsow.ac.uk

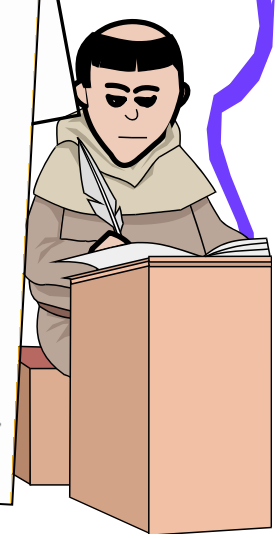
October 1992

This paper appears in ACM Symposium on Principles Of Programming Languages (POPL), Charleston, Jan 1993, pp71-84. This copy corrects a few minor typographical errors in the published version.

Abstract

We present a new model, based on monads, for perform-

I/O are constructed by gluing together smaller programs that do so (Section 2). Combined with higher-order functions and lazy evaluation, this gives a



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'
```

Connecting I/O operations

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

eg. Read two characters,
print the second, return both

```
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')`

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

Our biggest mistake

Using the scary term
"monad"
rather than
"warm fuzzy thing"

What have we achieved?

- The ability to mix imperative and purely-functional programming, without ruining either: the types keep them separate
- All laws of pure functional programming remain unconditionally true

Purity by default

effects are a little inconvenient

But why is purity good?

Purity pays: understanding

```
X1.insert( Y )
X2.delete( Y )
```

What does this program do?

- Would it matter if we swapped the order of these two calls?
- What if X1=X2?
- I wonder what *else* X1.insert does?

Lots of heroic work on static analysis, but hampered by unnecessary effects

Purity pays: verification

Spec#

```
void Insert( int index, object value )
  requires (0 <= index && index <= Count)
  ensures Forall{ int i in 0:index; old(this[i]) == this[i]
}
{ ... }
```

Pre-condition

Post-condition

- The pre and post-conditions are written in... a functional language
- Also: object invariants
  But: invariants temporarily broken
  Hence: "expose" statements

Purity pays: testing

A property of sets

$s \cup s = s$

```
propUnion :: Set a -> Bool
propUnion s  =  union s s  ==  s
```

In an imperative or OO language, you must

- set up the state of the object, and the external state it reads or writes
- make the call
- inspect the state of the object, and the external state
- perhaps copy part of the object or global state, so that you can use it in the postcondition

Purity pays: maintenance

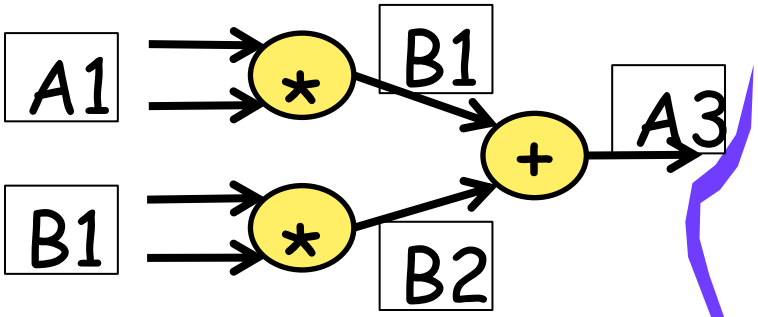
- The **type** of a function tells you a LOT about it

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

- Large-scale data representation changes in a multi-100kloc code base can be done reliably:
  - change the representation
  - compile until no type errors
  - works

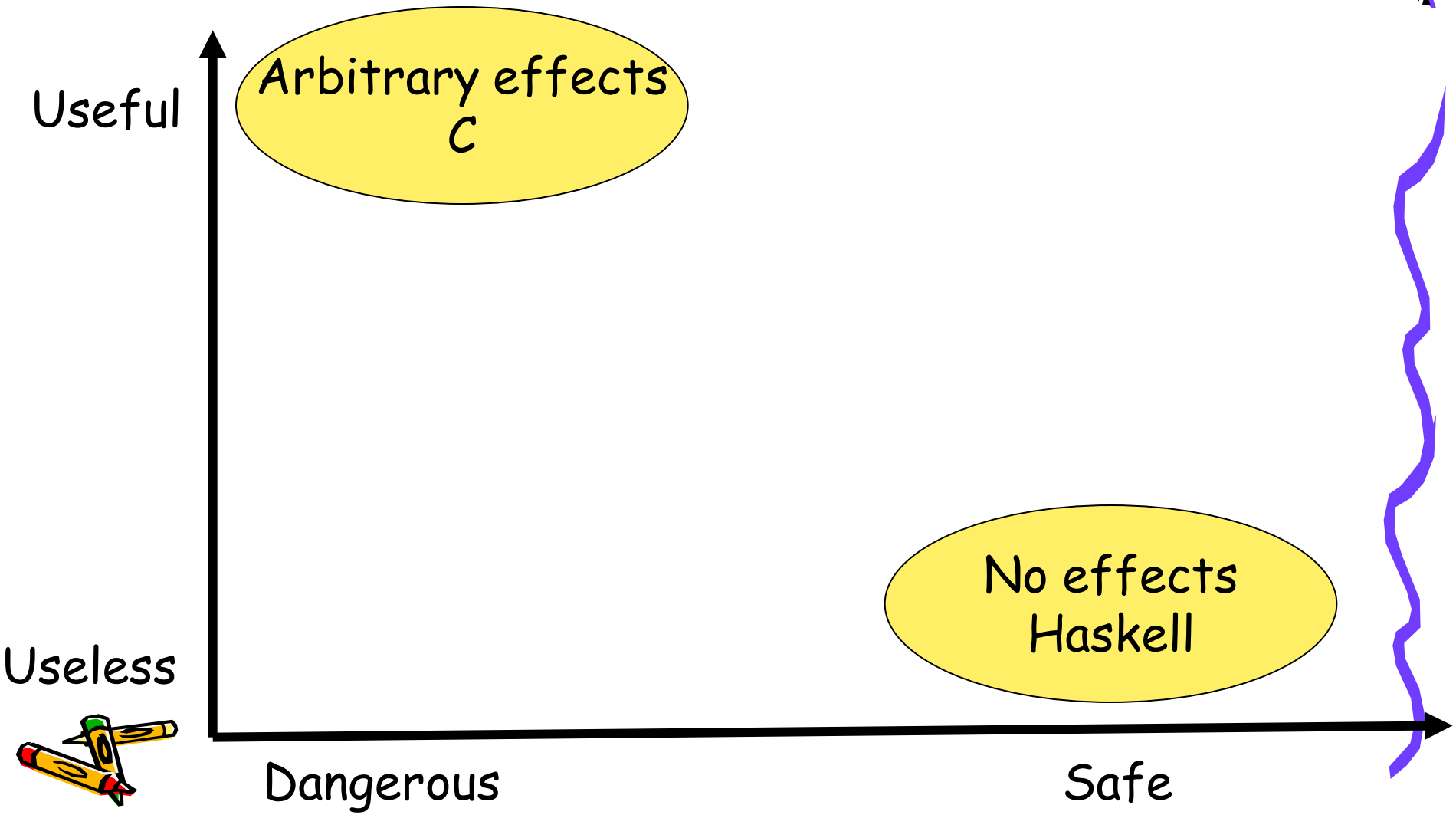
Purity pays: parallelism

- Pure programs are "naturally parallel"
- No mutable state means no locks, no race hazards

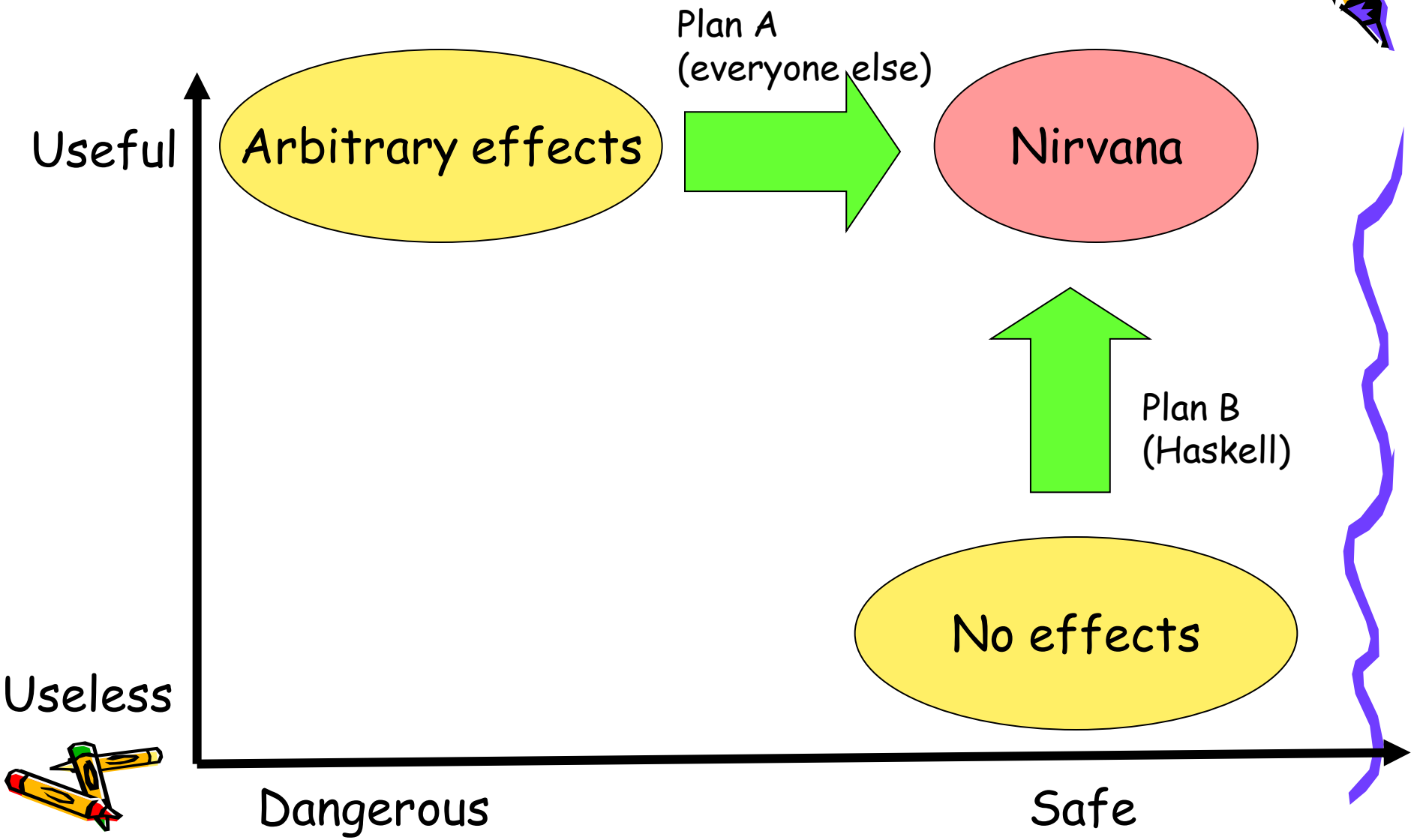


- Results totally unaffected by parallelism (1 processor or zillions)
- Examples
  - Google's map/reduce
  - SQL on clusters
  - PLINQ

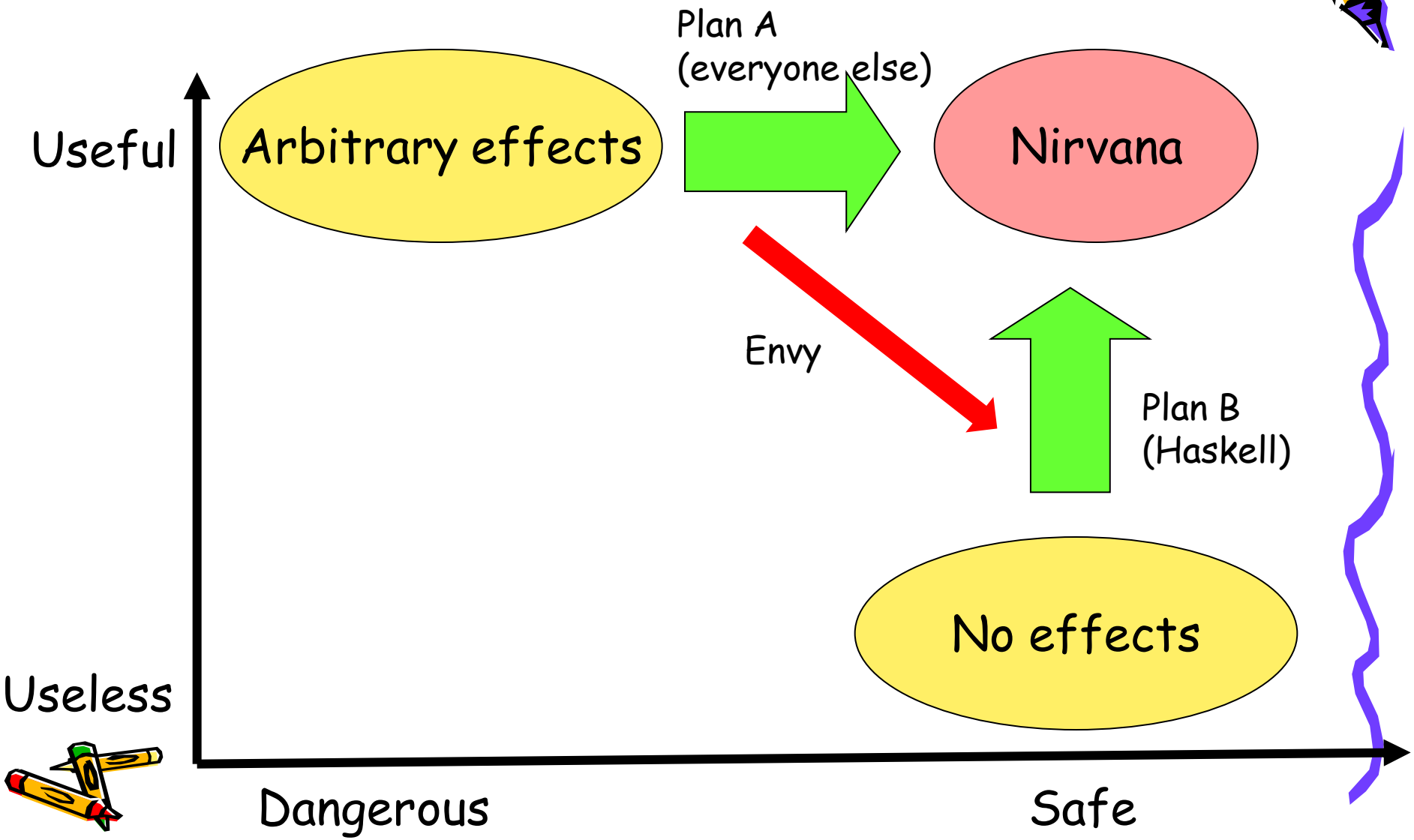
The challenge of effects



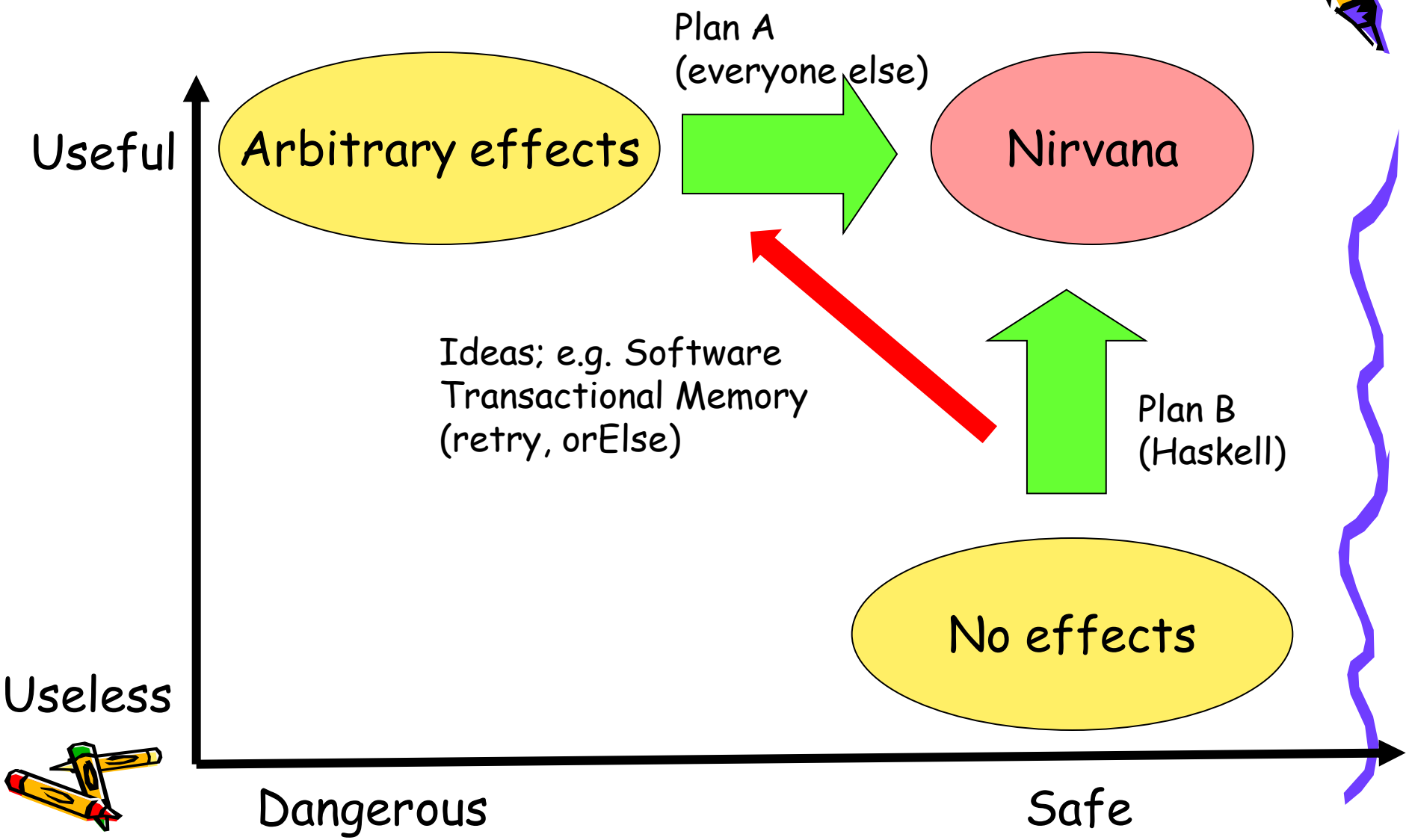
The challenge of effects



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**
- Purely functional programming feels very very different: you have to "rewire your brain"
- But it's not "just another approach": ultimately, there is no alternative.

Types and type classes

Starting point: ML

- Parametric polymorphism

```
append :: [a] -> [a]
```

- Types are inferred

```
append []       ys = ys
append (x:xs) ys = x : append xs ys
```

- Algebraic data types

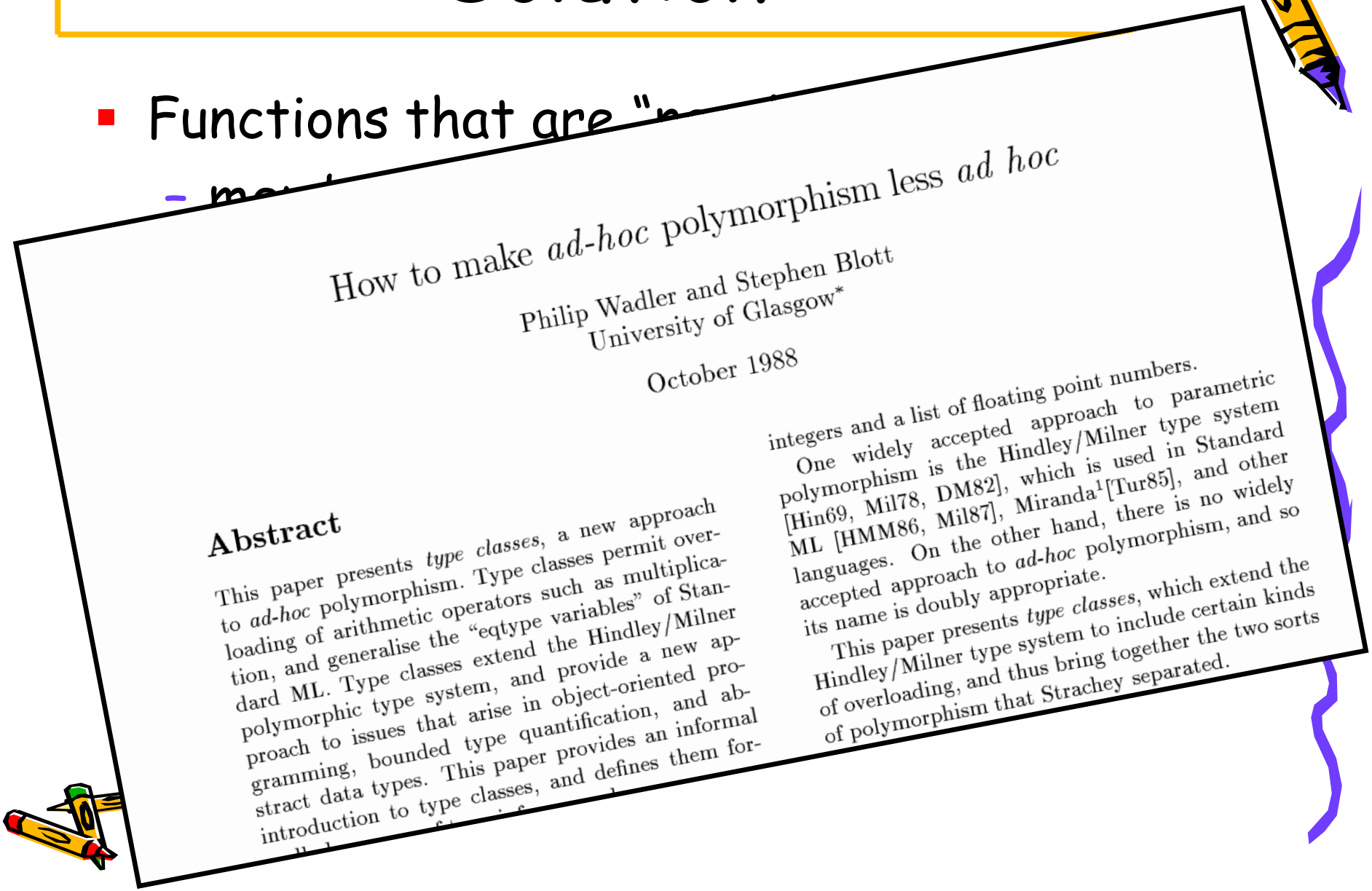
```
data Tree a
     = Leaf a
     | Branch (Tree a) (Tree a)
```

Problem

- Functions that are "nearly polymorphic"
  - member :: a -> [a] -> Bool
  - sort :: [a] -> [a]
  - square :: a -> a
  - show :: a -> String
  - serialise :: a -> BitString
  - hash :: a -> Int
- Usual solution: "bake them in" as a runtime service

Solution

- Functions that are "n
  - m



How to make *ad-hoc* polymorphism less *ad hoc*

Philip Wadler and Stephen Blott
University of Glasgow*

October 1988

Abstract

This paper presents *type classes*, a new approach to *ad-hoc* polymorphism. Type classes permit overloading of arithmetic operators such as multiplication, and generalise the "eqtype variables" of Standard ML. Type classes extend the Hindley/Milner polymorphic type system, and provide a new approach to issues that arise in object-oriented programming, bounded type quantification, and abstract data types. This paper provides an informal introduction to type classes, and defines them for-

integers and a list of floating point numbers.

One widely accepted approach to parametric polymorphism is the Hindley/Milner type system [Hin69, Mil78, DM82], which is used in Standard ML [HMM86, Mil87], Miranda[1][Tur85], and other languages. On the other hand, there is no widely accepted approach to *ad-hoc* polymorphism, and so its name is doubly appropriate.

This paper presents *type classes*, which extend the Hindley/Milner type system to include certain kinds of overloading, and thus bring together the two sorts of polymorphism that Strachey separated.

Type classes

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

```
square :: a -> a
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
```

Declaring classes

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

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

instance Num Int where
   (+) = plusInt
   (*) = mulInt
   ...etc...
```

Haskell **class** is like a Java **interface**

Allows 'square' to be applied to an Int

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

Unlike OOP…

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

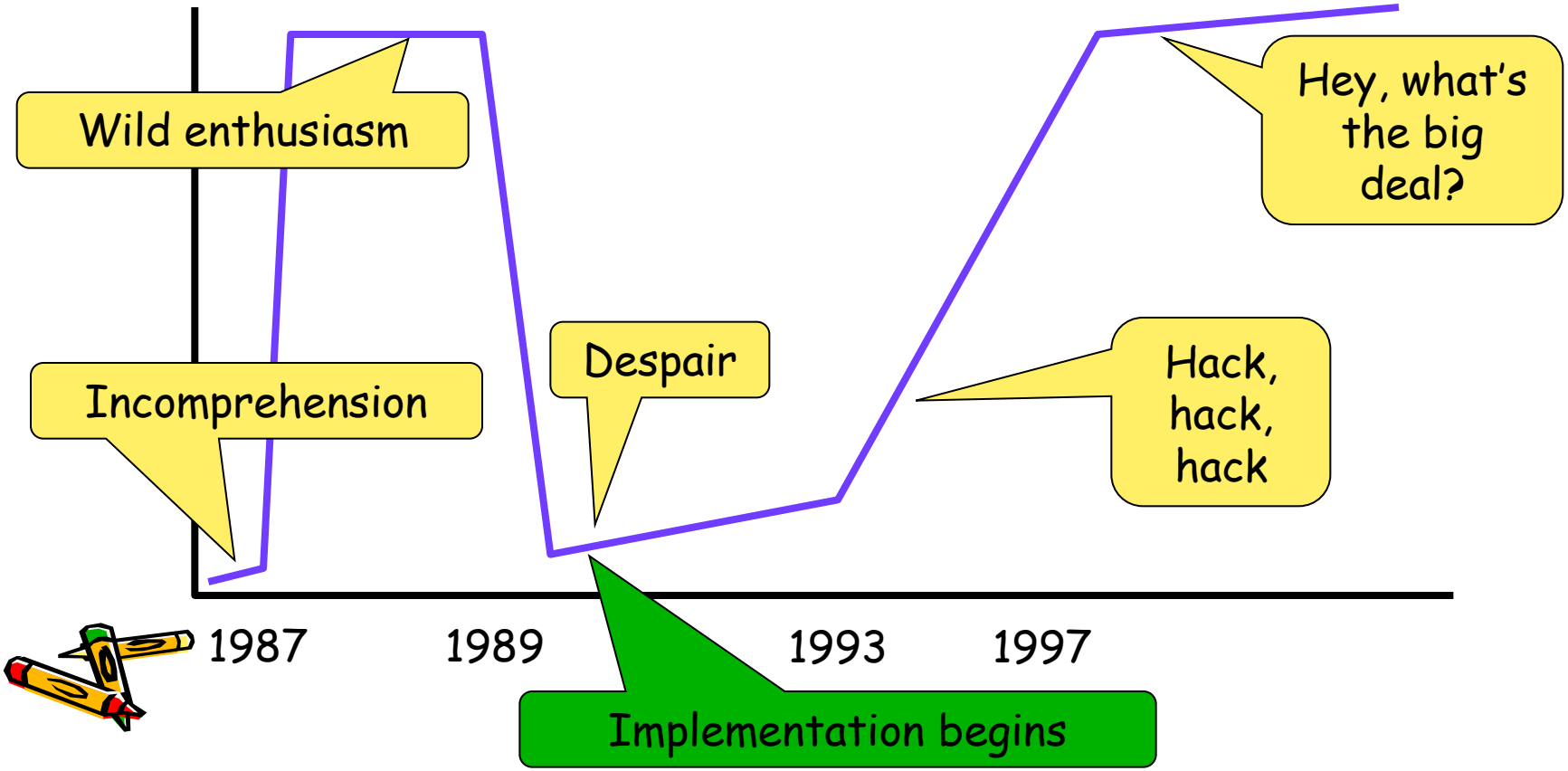
```
readSq :: (Read a, Num a) => String -> a
readSq s = square (read s)
```

```
readSq dr dn s = square dn (read dr s)
```

- Unlike OOP:
  - The vtables are passed **in**
  - The value of type 'a' is returned **out**
- This ability turns out to be a Big Deal

Type classes over time

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



Type classes have proved extraordinarily convenient in practice

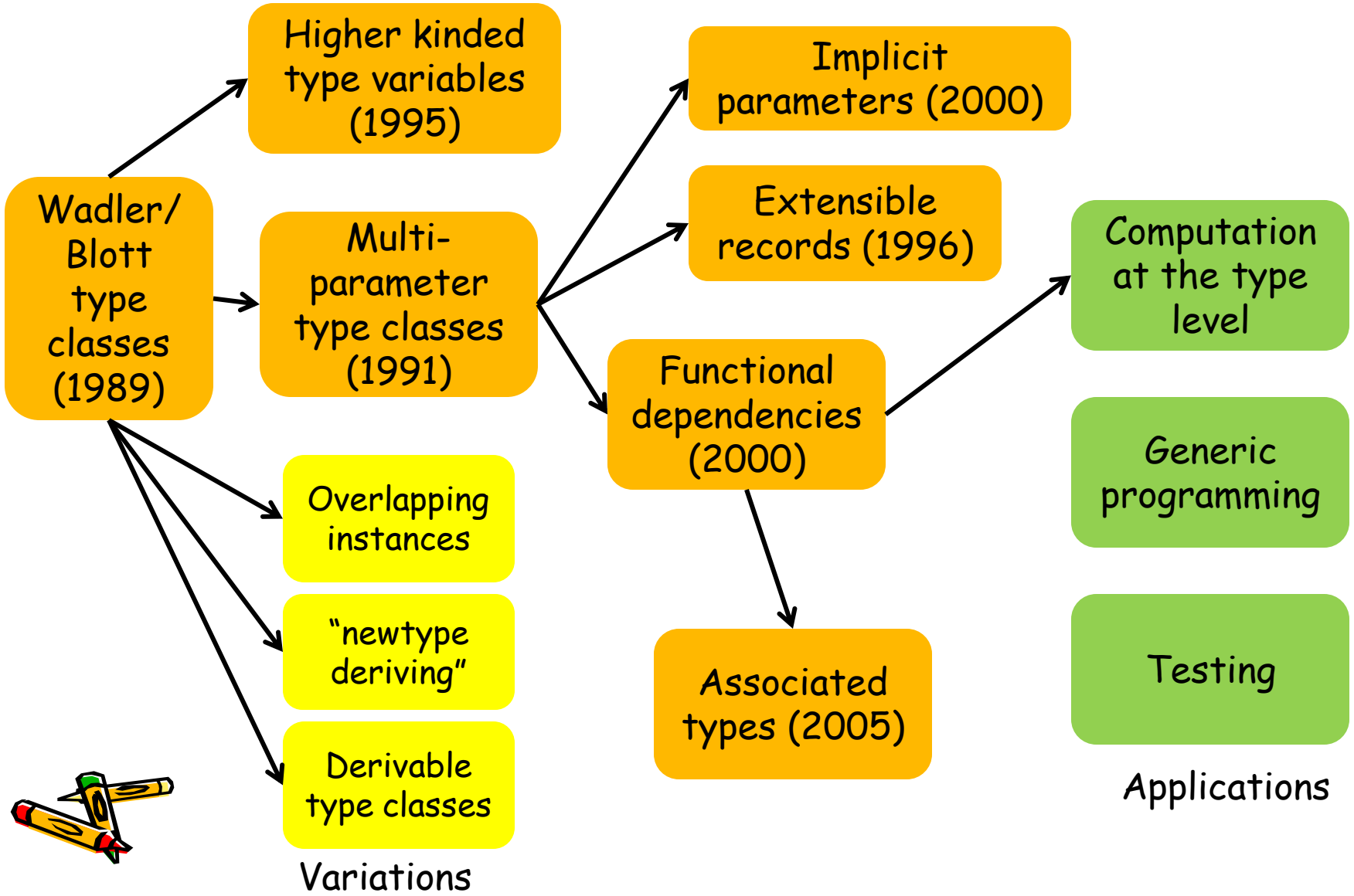
- Equality, ordering, serialisation
- Numerical operations. Even numeric constants are overloaded
- Monadic operations

```
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....

Note the higher-kinded type variable, m

Type-class fertility



Sexy types

Haskell has become a laboratory and playground for advanced type systems

- 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`

Sexy types

Haskell has become a laboratory and playground for advanced type systems

- Generalised Algebraic Data Types (GADTs)

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

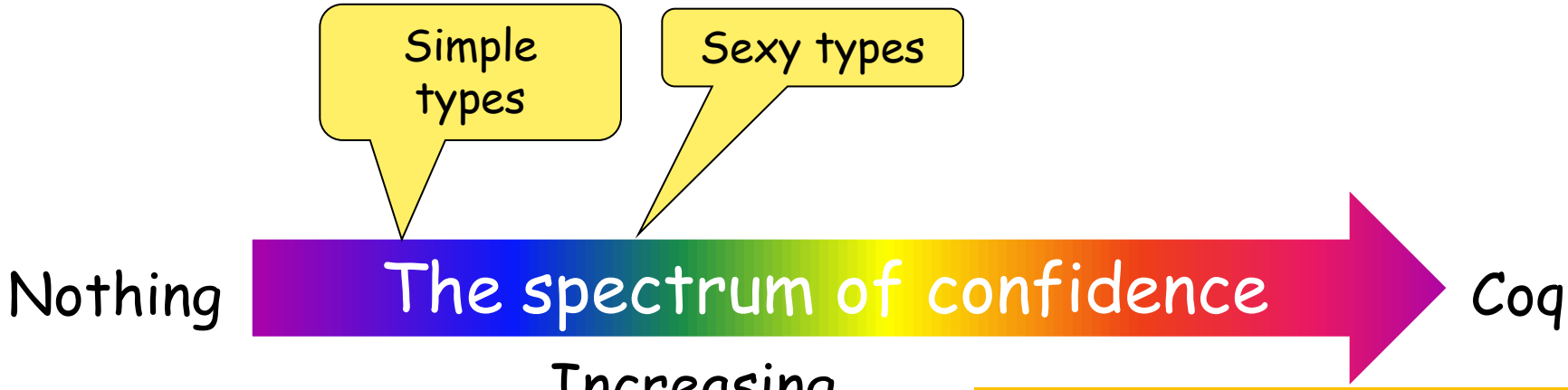
- Type families and associated types

```
class Collection c where
  type Elem c
  insert :: Elem c -> c -> c
```

- Polymorphic kinds
- .....and on and on

Building on success

- Static typing is by far the most successful program verification technology in use today
  - Comprehensible to Joe Programmer
  - Checked on every compilation

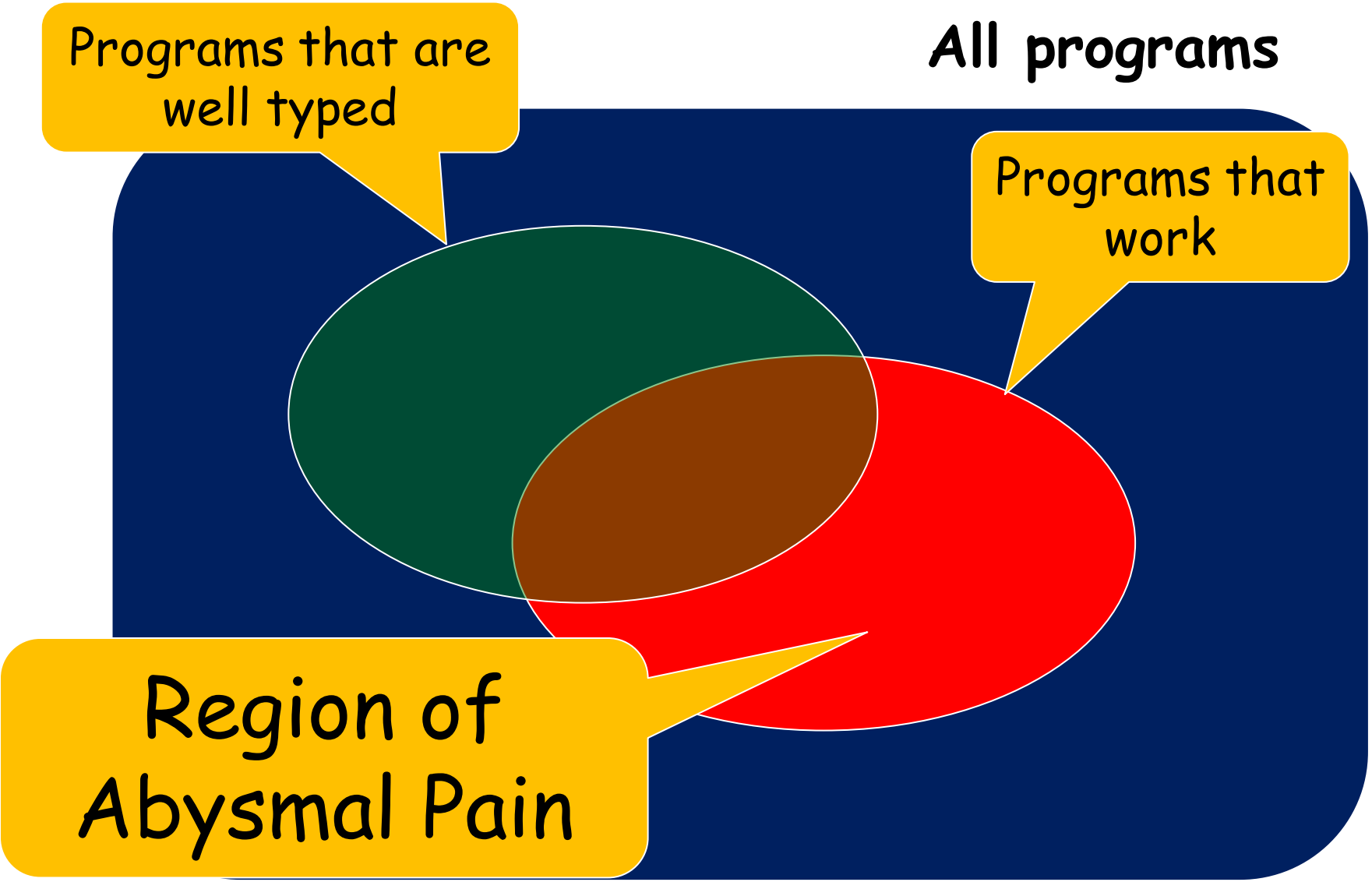


Hammer (cheap, easy to use, limited effectivenes)

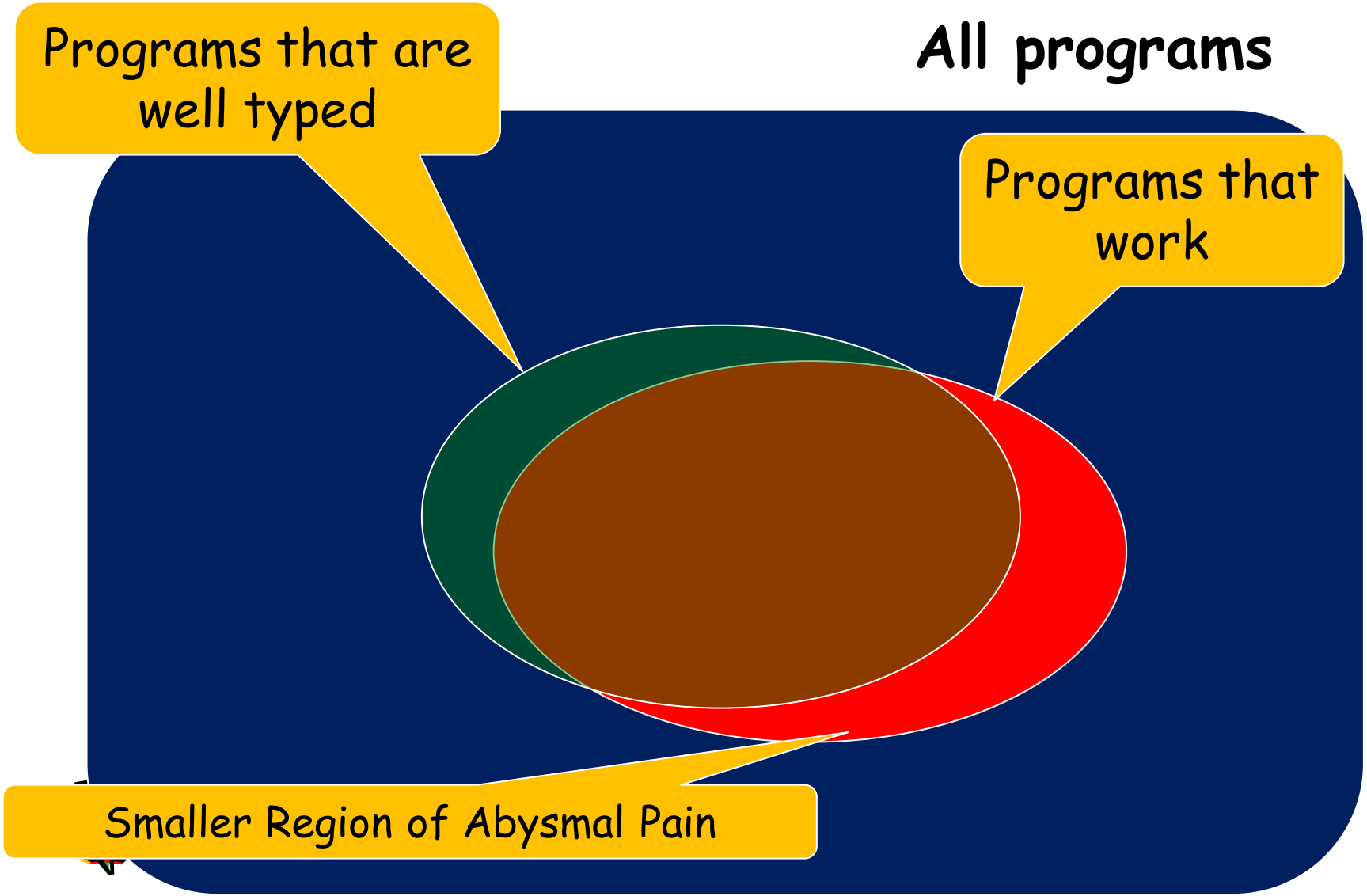
Increasing confidence that the program does what you want

Tactical nuclear weapon (expensive, needs a trained user, but very effective indeed)

Bad type systems



Sexy type systems



Plan for World Domination

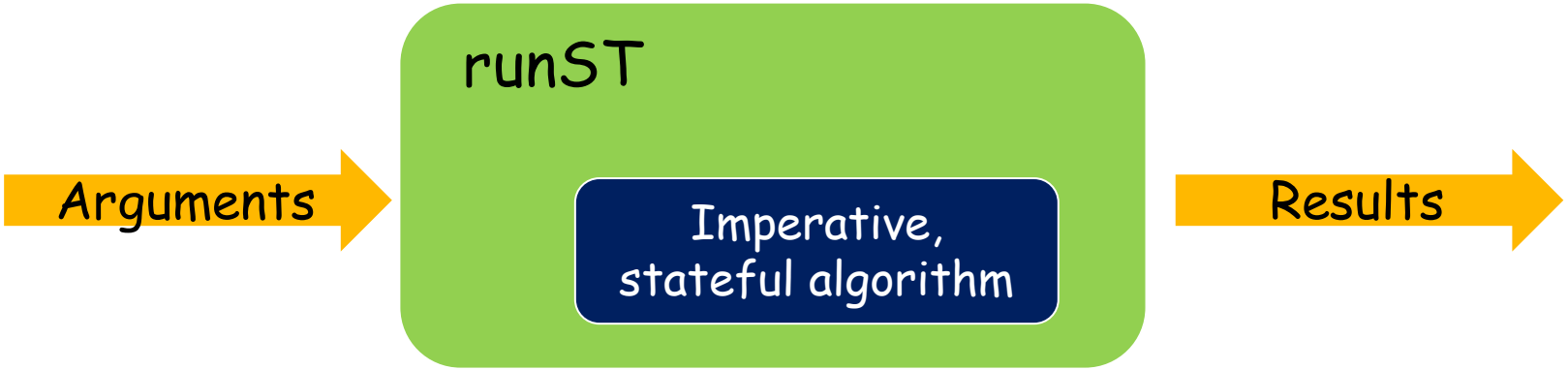
- Build on the demonstrated success of static types
- ...by making the type system more expressive
- ...so that more good programs are accepted (and more bad ones rejected)
- ...without losing the Joyful Properties (comprehensible to programmers)

Encapsulating it all

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

Stateful computation

Pure result



A pure function

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)

Closing thoughts

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

Fun

- Haskell is rich enough to be very useful for real applications
- 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
- You can play too....

Escape from the ivory tower

- You will be a better Java programmer if you learn Haskell
- The ideas are more important than the language: Haskell aspires to infect your **brain** more than your **hard drive**
- The ideas really **are** important IMHO
  - Purity (or at least controlling effects)
  - Types (for big, long-lived software)

  Haskell is a laboratory where you can see these ideas in distilled form

  (But take care: addiction is easy and irreversible)

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