Esape from the ivory tower
The Haskell journey

Simon Peyton Jones, Microsoft Research

May 2017

1976-80

John Hughes,
Maths, Churchill
(first)

John and Simon go to university

Simon Peyton Jones,
Maths, Trinity
(failed)

Early days of microprocessors

4kbytes is a lot of memory

Cambridge University has one (1) computer

and…

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)

SKIM: Lisp & FP 1980

SKIM - The S, K, I Reduction Machine

T.J.W. Clarke, P.J.S. Gladstone, C.D. MacLean, A.C. Norman

Trinity College, Cambridge

Abstract

SKIM is a computer built to explore pure functional programming, combinators as a machine language and the use of hardware to provide direct support for a high level language. Its design stresses simplicity and aims at providing minicomputer performance (in its particular application areas) for microcomputer costs. This paper discusses the high level reduction language that SKIM supports, the way in which this language is compiled into combinators and the hardware and microcode that then evaluate programs.

1. Introduction

In [1] Turner shows how combinators can be used as an intermediate representation for applicative programs. He compares (software) interpretation of combinator forms with more traditional schemes based on lambda calculus, and demonstrates that his new method is both elegant and efficient, at least when normal order evaluation is required. SKIM is an investigation of how Turner's ideas translate into hardware. It views his combinators as machine code, and the fixed program that obeys them as microcode. In section 2 we will present the particular applicative language we use, and comment on the need for special computers to support such languages. Section 3 reviews Turner's programming style which fits in very smoothly with the mathematical flavour of symbolic algebra. Also, since in an algebra system even small amounts of arithmetic may involve calling fairly expensive subroutines, the initial design for Small did not feel obliged to allow for compilation into efficient machine code. As a user-level language for driving large packages it can afford an interpretive implementation. This results in a language which demands proper treatment of functional objects (the Funarg facility, so often missing or restricted in full sized LISP systems), call-by-need (otherwise known as lazy evaluation) and an error-handling scheme compatible with the semantics of the rest of the language.

Figure 1 gives a few simple examples of Small functions and so illustrates how it compares with the direct use of lambda calculus or LISP. It is easy to demonstrate the positive features of a language such as Small, such as its pattern-matching test for decomposing structures, its capability for recursive definitions of data as well as program and its lazy evaluation. When these points have been covered there remain various real worries as to how practical Small could be for the development of large programs. Here we will ignore most of these - for instance those concerning the relationship between pure language and file stores - and just discuss the two concerns that we have considered most pressing. We pose each in the form of direct questions:

SLPJ: Lisp & FP 1982

AN INVESTIGATION OF THE RELATIVE EFFICIENCIES OF

COMBINATORS AND LAMBDA EXPRESSIONS

by

Simon L Peyton Jones
Beale Electronic Systems Ltd
Whitehall, Wraysbury, UK.

ABSTRACT

In 'A New Implementation Technique for Applicative Languages' [Tu79a] Turner uses combinators to implement lambda expressions. This paper describes an experimental investigation of the efficiency of Turner's technique compared with more traditional reducers.

OVERVIEW

The basis for comparison of the two systems is discussed in Section 1. This is followed by some implementation considerations in Section 2, while the main results are presented in Section 3. Section 4 presents some discussion of the results and related issues, and conclusions are drawn in Section 5.

1 BASIS FOR COMPARISON

1.1 Background

Functional languages are characterised by the absence of side effects and imperative commands. They are the focus of considerable current programming errors are less likely, and programs are more amenable to formal verification.

(ii) The absence of side effects means that expressions can be concurrently evaluated by several cooperating processors. This suggests functional languages as a base for highly parallel computing.

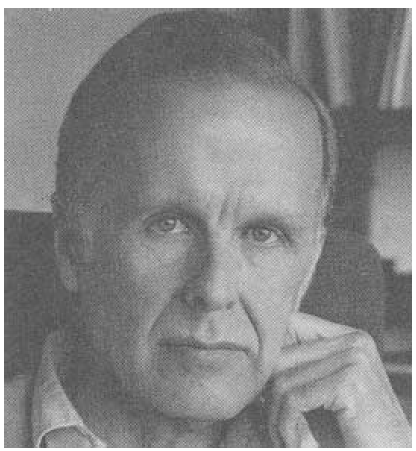
The two main techniques for efficiently implementing functional semantics are data flow and reduction. This paper concentrates exclusively on the implementation of reduction techniques.

The cannonical reduction architecture is the lambda calculus, which has an extensive literature (eg [Ch41], [St77b]). However, some old results derived by Curry and Feys [Cu58] have been used by Turner [Tu79] to implement a reduction machine for the combinator calculus. The combinator calculus has the same semantics as the lambda calculus, but has a rather different implementation. Thus the two calculi can be thought of as two machine codes for a functional .igh-level

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

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

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

Dataflow architectures (Arvind et al)

SK combinators, graph reduction (Turner)

Backus

Can programming be liberated from the von Neumann style?

The Call

Functi
recursi
comp
(ML, S

Datafl
(

rs,
ion
)

Have no truck with the grubby compromises of imperative programming!

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

Result

Chaos

Many bright young things

Many conferences
(birth of FPCA, LFP)

Many languages
(Sasl, 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.

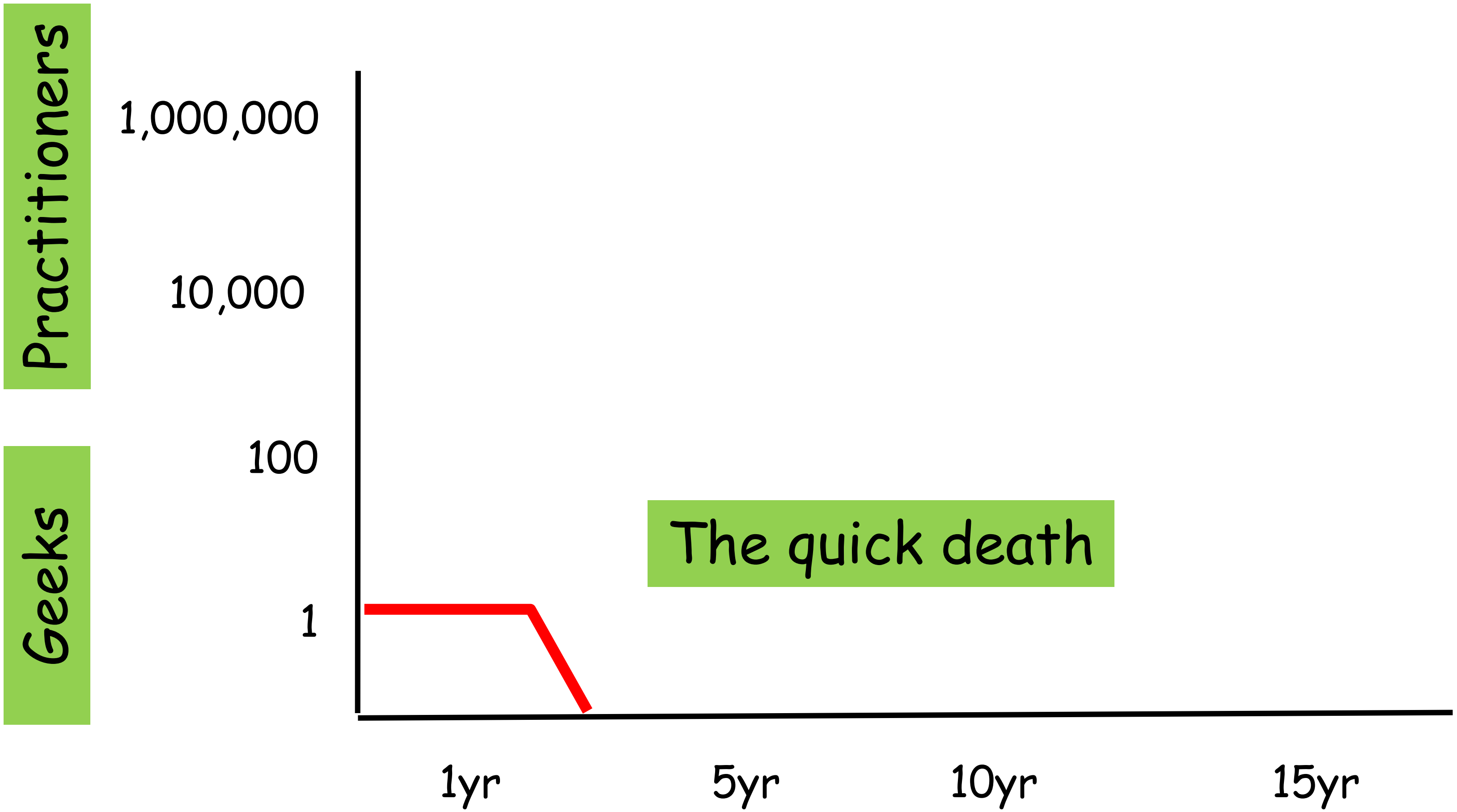
- Suitable for teaching, research, and application
- Formally-described syntax and semantics
- Freely available
- Embody the apparent consensus of ideas
- Reduce unnecessary diversity

Absolutely no clue how much work we were taking on

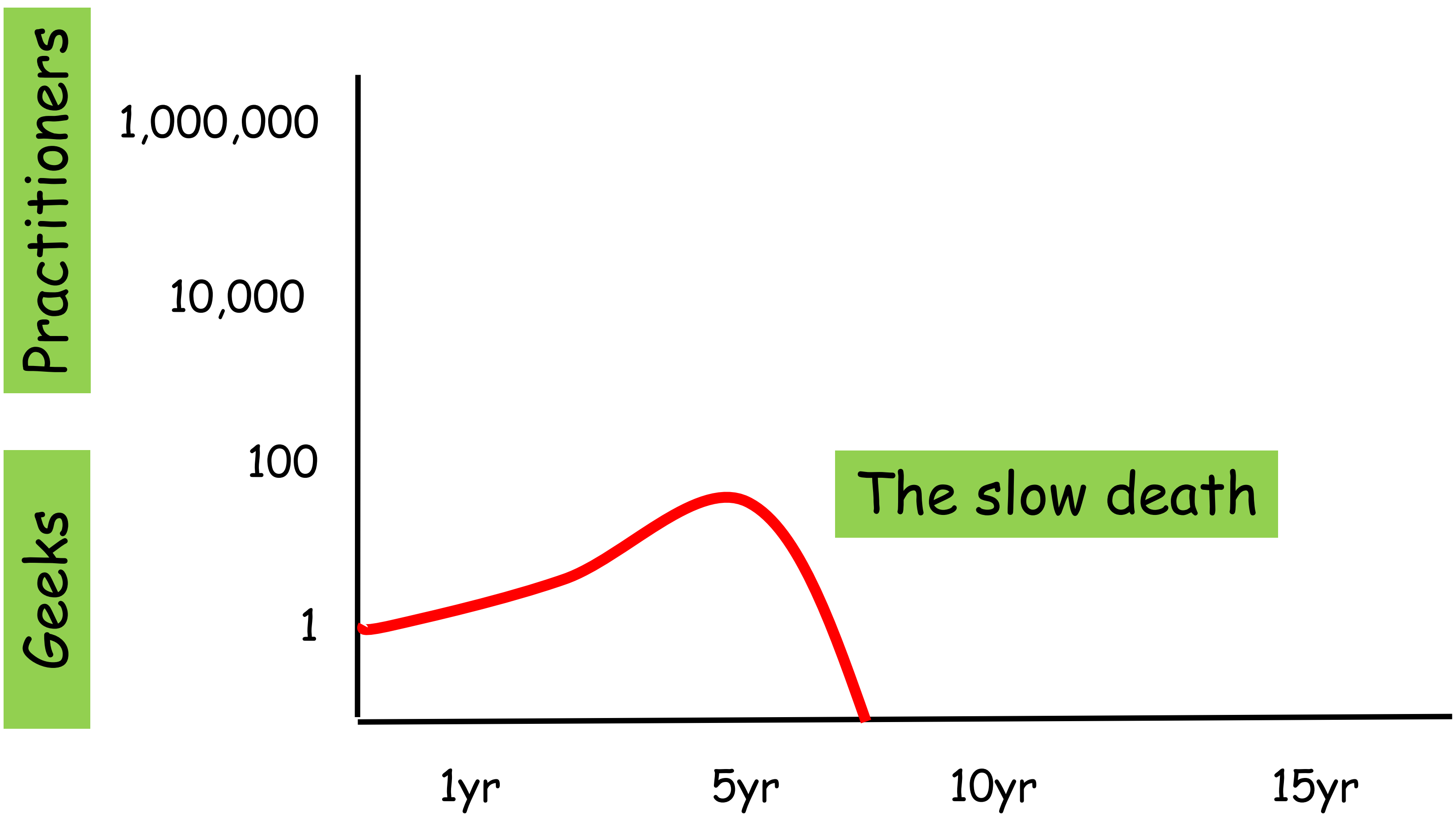
Led to...a succession of face-to-face meetings

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

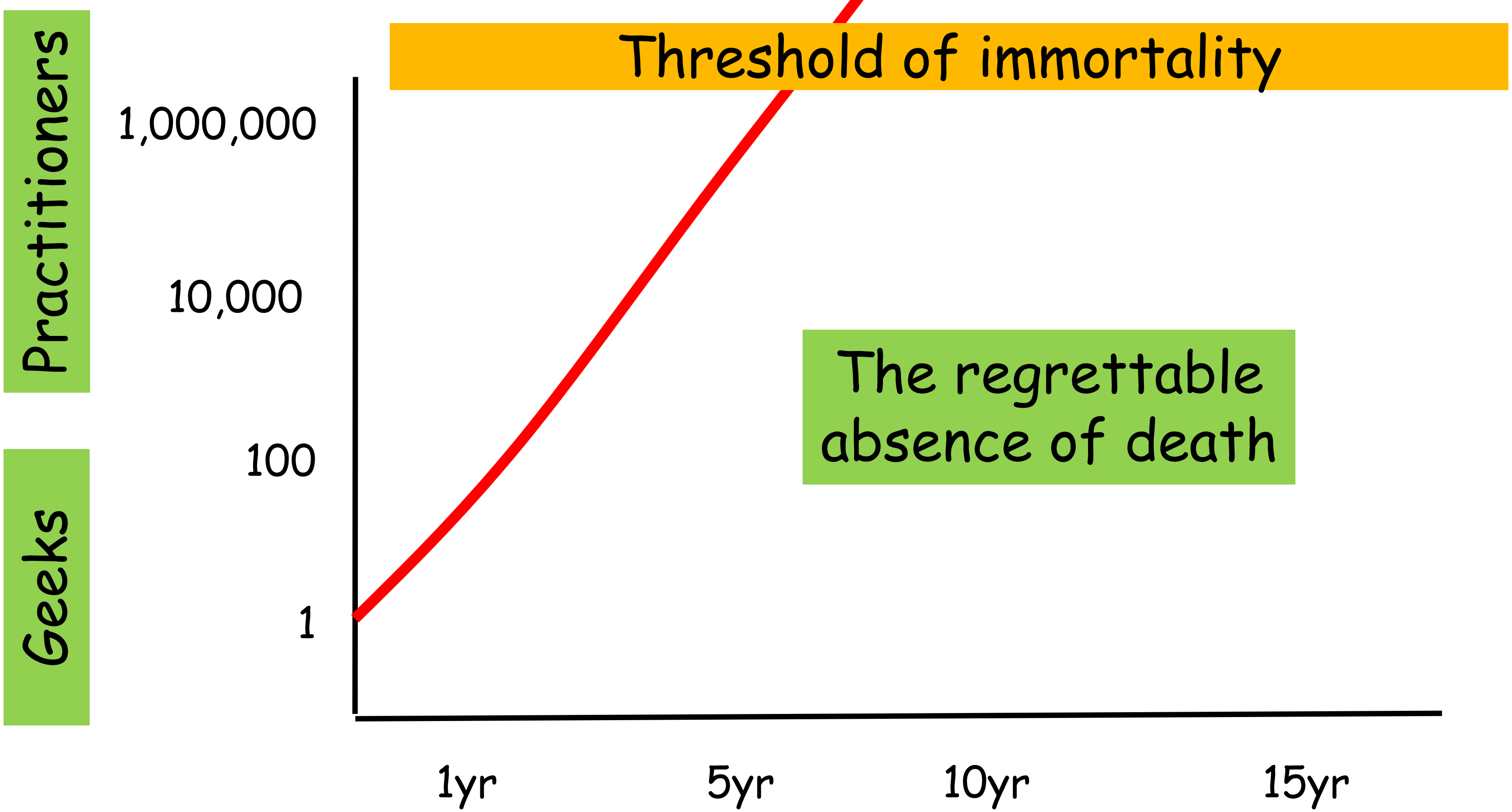
History of most research languages



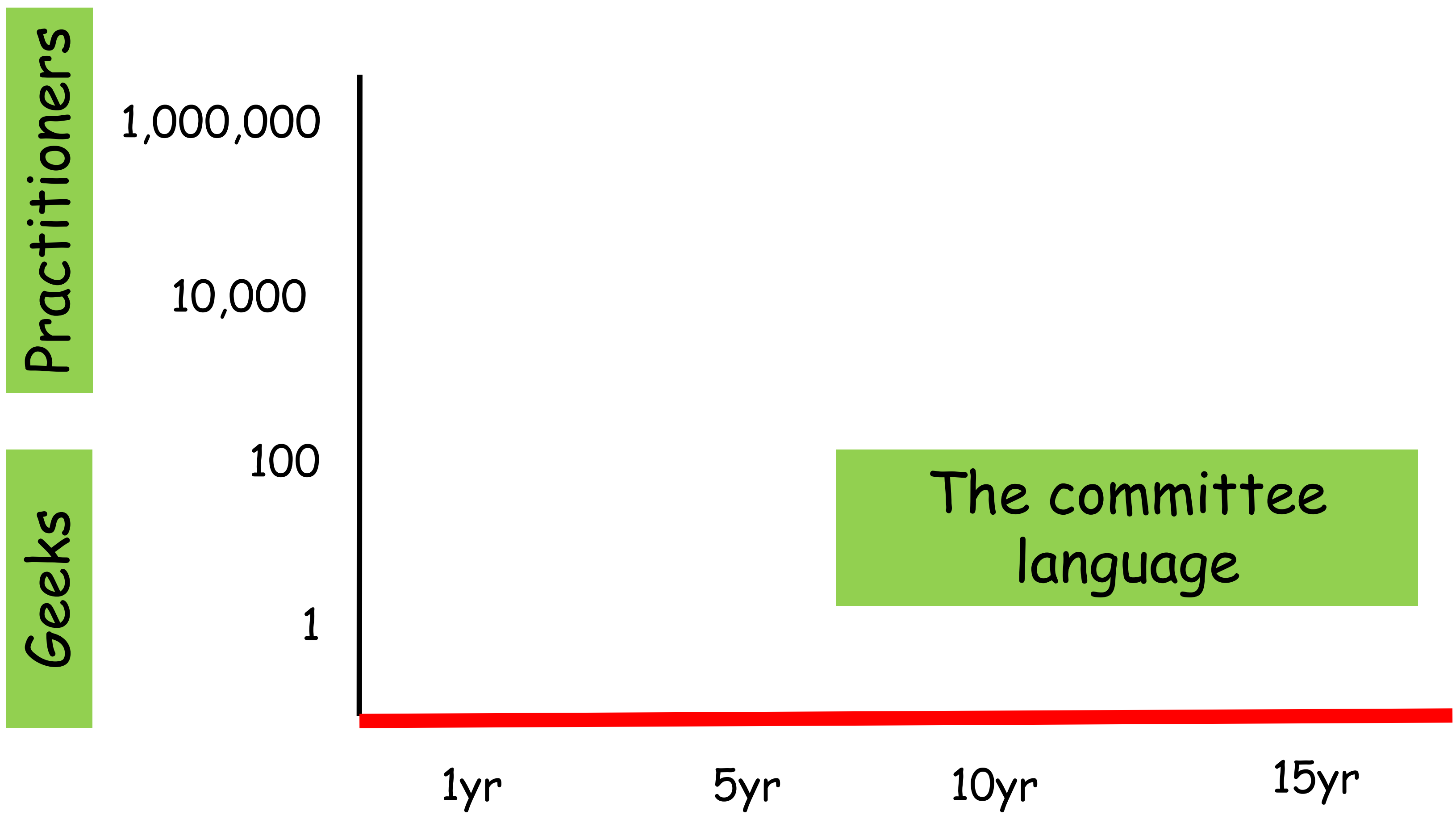
Successful research languages



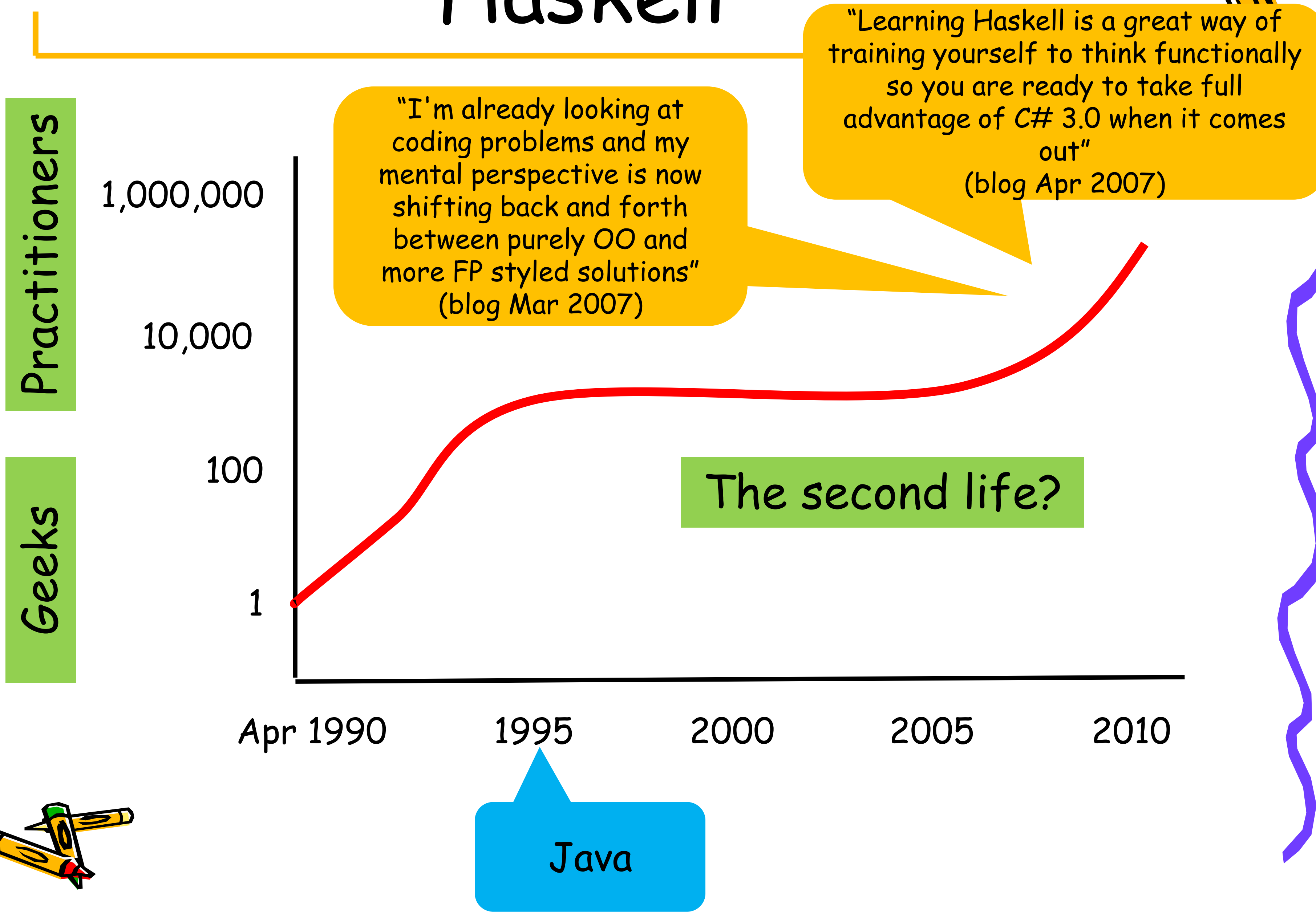
C++, Java, Perl, Ruby



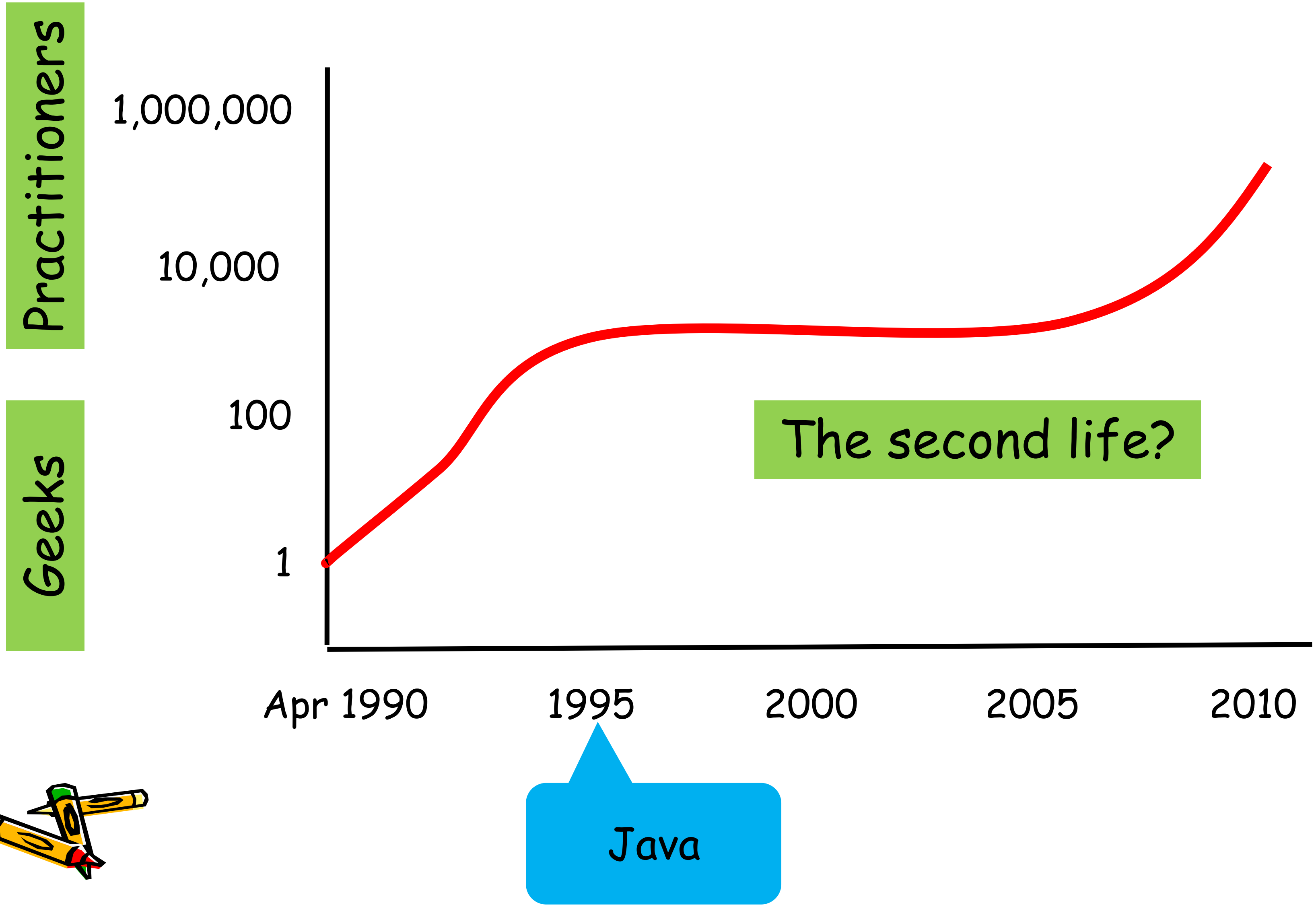
Committee languages



Haskell

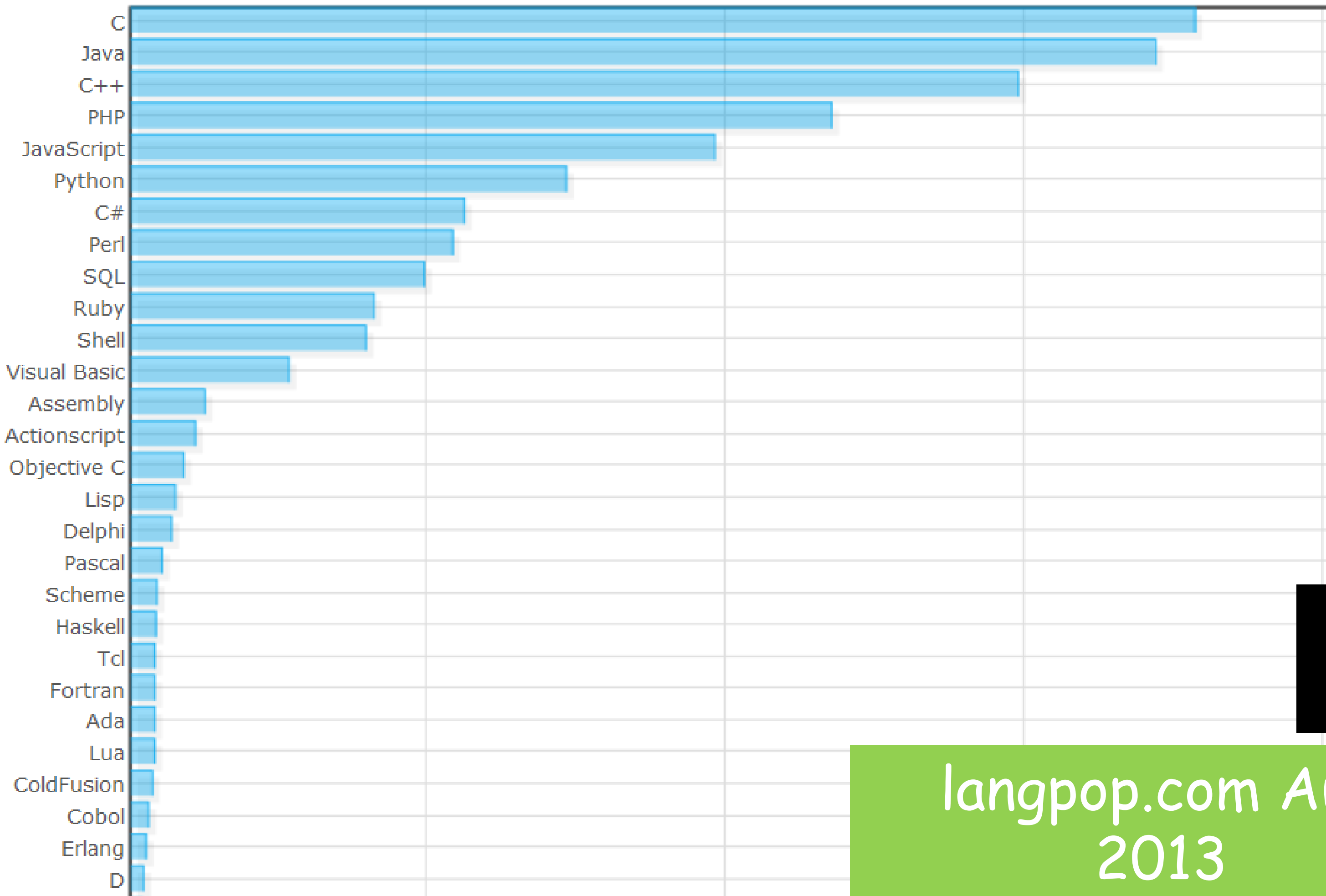

Practitioners
Geeks
1,000,000
10,000
100
1
Apr 1990
1995
2000
2005
2010
"I'm already looking at coding problems and my mental perspective is now shifting back and forth between purely OO and more FP styled solutions"
(blog Mar 2007)
"Learning Haskell is a great way of training yourself to think functionally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

Haskell


Practitioners
Geeks
1,000,000
10,000
100
1
The second life?
Apr 1990
1995
2000
2005
2010
Java

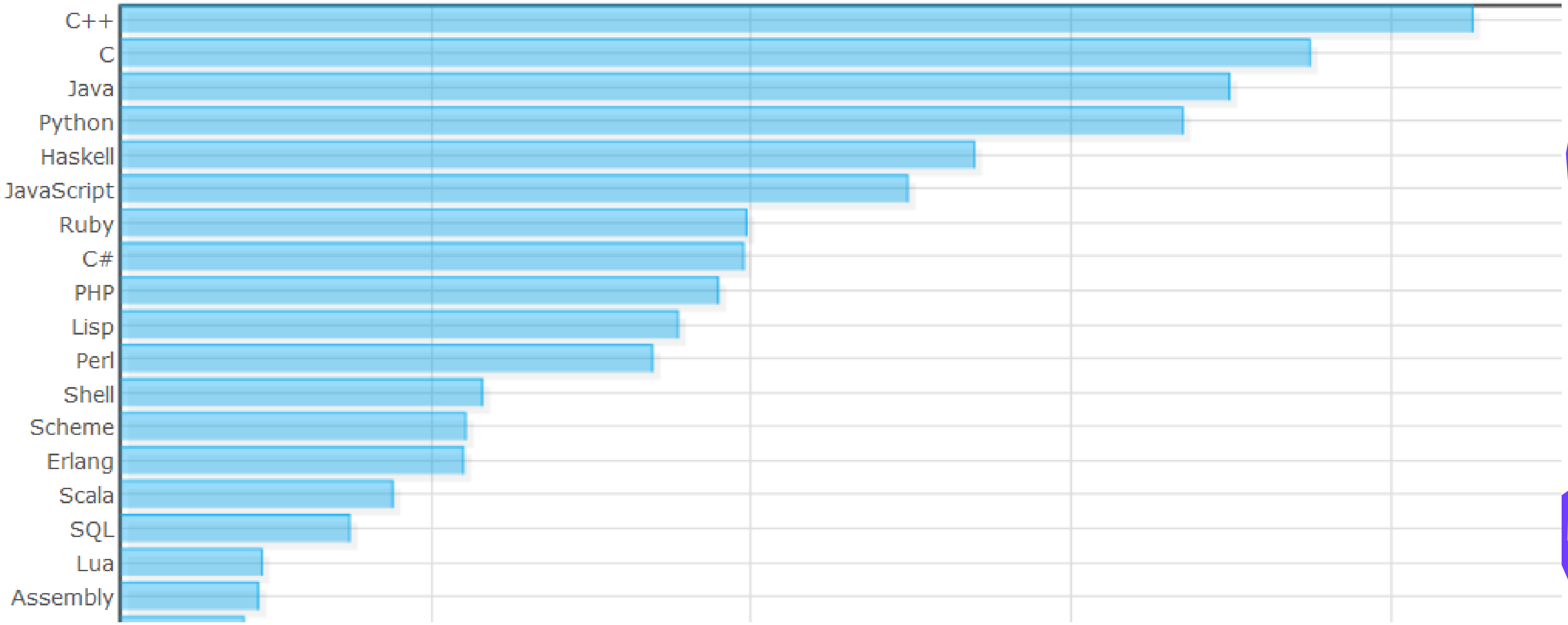
Language popularity
how much language X is used

This is a chart showing combined results from all data sets, listed individually below.



langpop.com Aug 2013

Language popularity
how much language X is talked about



langpop.com Aug 2013

THIS ALL STARTED
A VERY LONG TIME AGO

Born April 1990, Haskell is 27

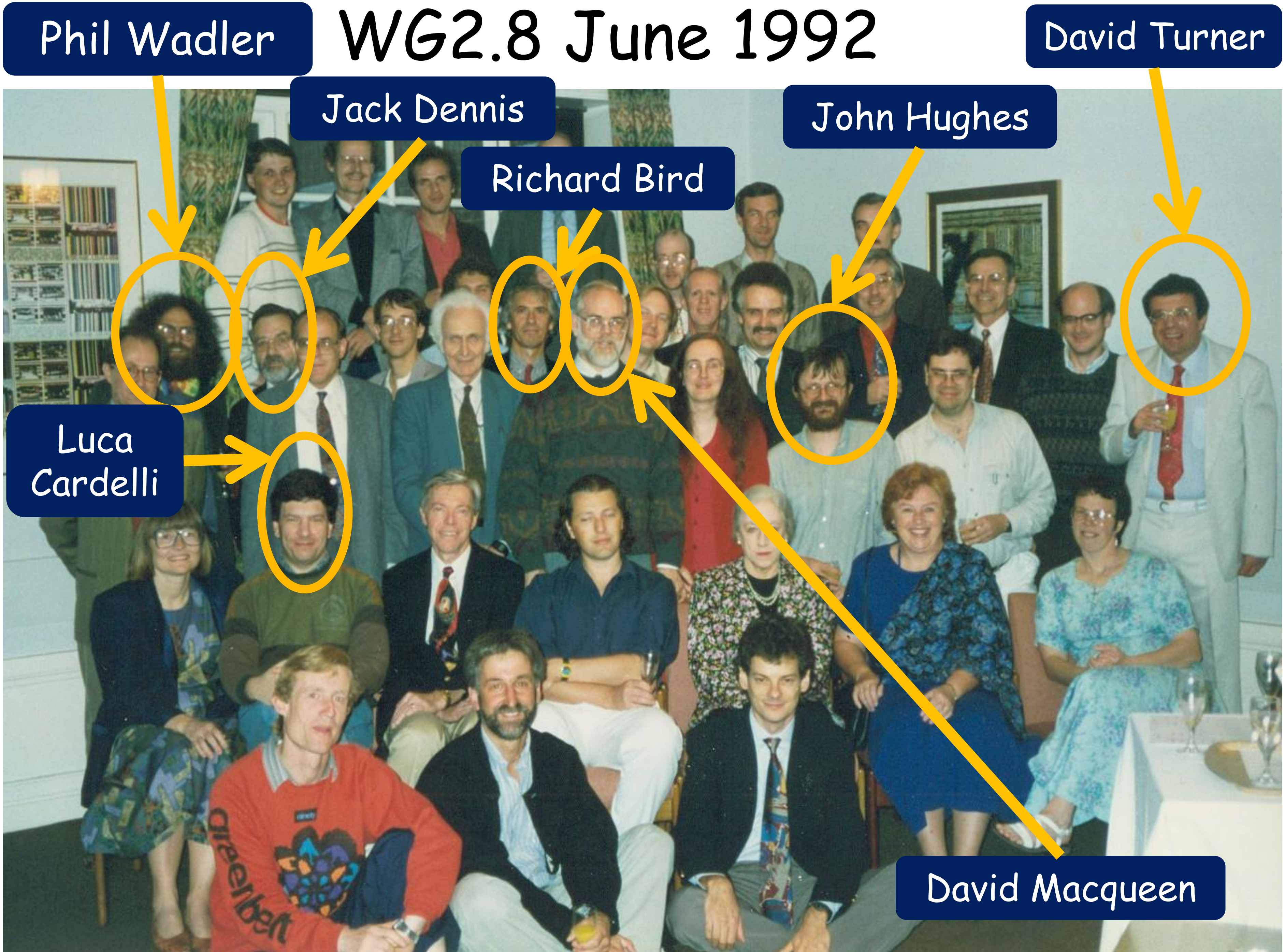
And so is Michael

Meg (b 1995) Michael (b May 1990) Sarah (b 1993)

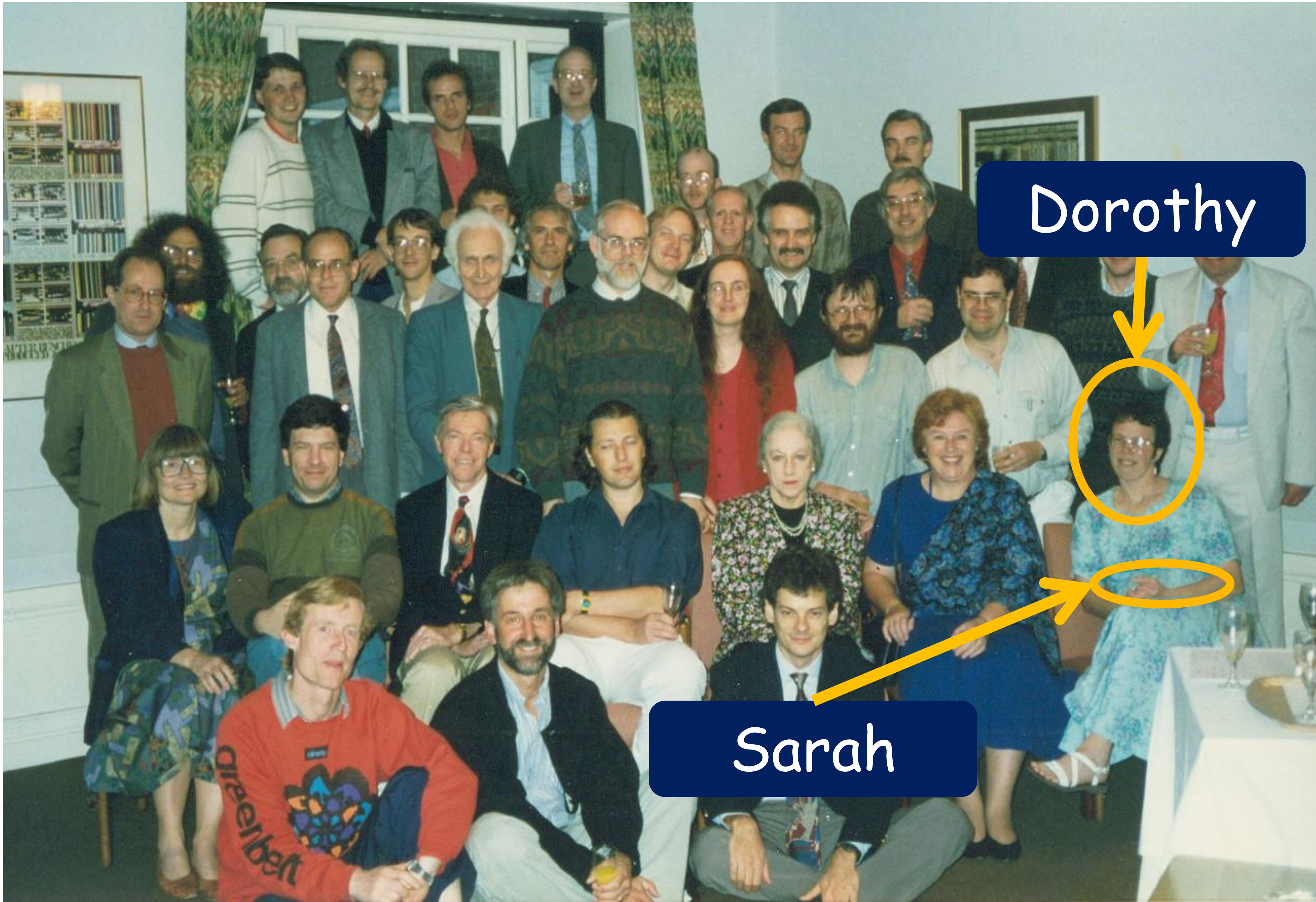

Haskell the cat (b. 2002)

WG2.8 June 1992

WG2.8 June 1992



WG2.8 June 1992



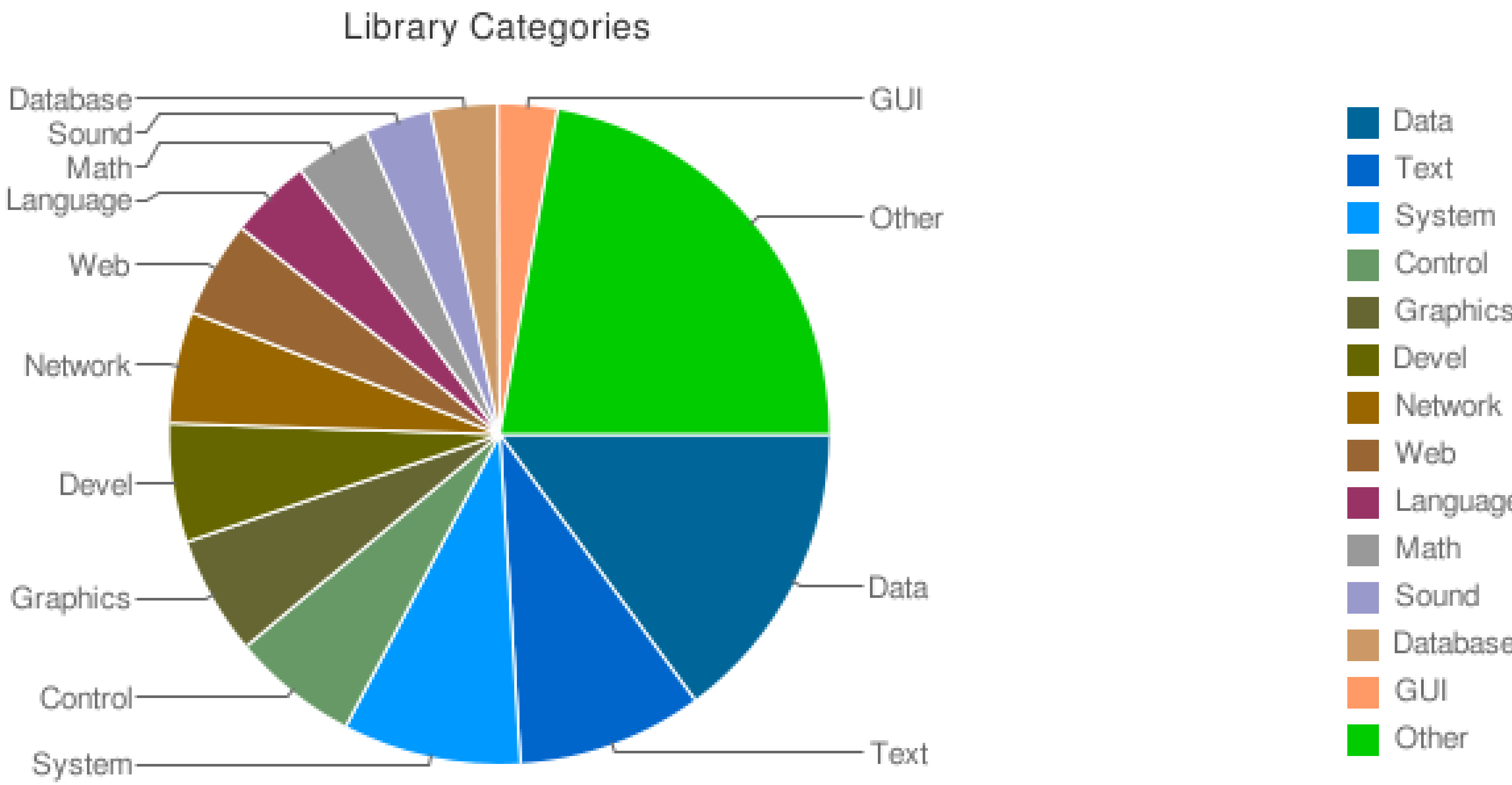
THIS ALL STARTED
A VERY LONG TIME AGO

BUT IT IS STILL GOING
STRONG

The Glasgow Haskell Compiler

- GHC today
  - First release 1991: 13k lines, 110 modules, sequential
  - Now: 150k lines, 380 modules, parallel
- >> 100k users
- 100% open source (BSD)
- Still in furious development: > 200 commits/month

Now over 11,000 packages on Hackage



Incredibly supportive community

Haskell Weekly News

- *Anonymous*: I'd love to explain to you how to write hello world in Haskell, but first let me introduce you to basic category theory
- *neutrino*: in many ways, Haskell is like this primodial soup out of which other languages end up springing
- *hobophobe*: So, I can only conclude that Haskell is a memetic virus, and monads are the eggs it lays out in innocent programming forums to entice others to become infected

- GuySteele: Some people prefer not to commingle the functional, lambda-calculus part of a language with the parts that do side effects.

  It seems they believe in the separation of Church and State.

My favourite

- *Berengal*: I was squashing a bug, got frustrated, and typed "fix error" in ghci...

After 26 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

Avoiding success

- A user base that makes Haskell nimble:
  - Smallish: enough users to drive innovation, not so many as to stifle it
  - Tolerant of bugs in GHC. Very tolerant.
  - Innovative and slightly geeky: Haskell users react to new features like hyenas react to red meat
  - Extremely friendly
- Avoided the Dead Hand of standardisation committees

What deep, simple principles?

1. A tiny core language
2. Purity and laziness
3. Types; especially type classes

GHC

Module	Lines (1992)	Lines (2011)	Increase
Compiler			
Main	997	11,150	11.2
Parser	1,055	4,098	3.9
Renamer	2,828	4,630	1.6
Type checking	3,352	24,097	7.2
Desugaring	1,381	7,091	5.1
Core tranformations	1,631	9,480	5.8
STG transformations	814	840	1
Data-Parallel Haskell	—	3,718	—
Code generation	2913	11,003	3.8
Native code generation	—	14,138	—
LLVM code generation	—	2,266	—
GHCi	—	7,474	—
Haskell abstract syntax	2,546	3,700	1.5
Core language	1,075	4,798	4.5
STG language	517	693	1.3
C-- (was Abstract C)	1,416	7,591	5.4
Identifier representations	1,831	3,120	1.7
Type representations	1,628	3,808	2.3
Prelude definitions	3,111	2,692	0.9
Utilities	1,989	7,878	3.96
Profiling	191	367	1.92
Compiler Total	28,275	139,955	4.9
Runtime System			
All C and C-- code	43,865	48,450	1.10

Figure 1: Lines of code in GHC, past and present

Deep, simple principles

Source language

Intermediate language

Haskell

Dozens of types

100+ constructors

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
data Bind   = NonRec Var Expr | Rec [(Var,Expr)]
data AltCon = DEFAULT | LitAlt Lit | DataAlt DataCon
```

System FC

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

Everything has to translate into this tiny language

Statically typed (very unusual)

Fantastic language design sanity check

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 John did not mention the most important reason

Laziness keeps you pure

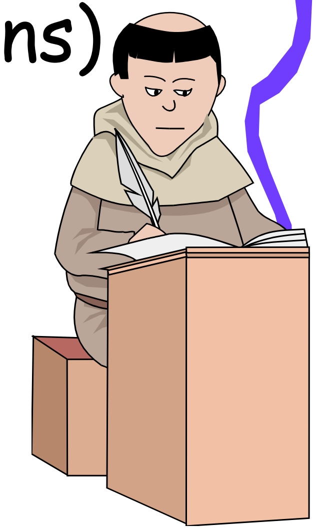
- Every call-by-value language has given into the siren call of side effects
- But in Haskell

```
f (print "yes") (print "no")
```

just does not make sense. Even worse is

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

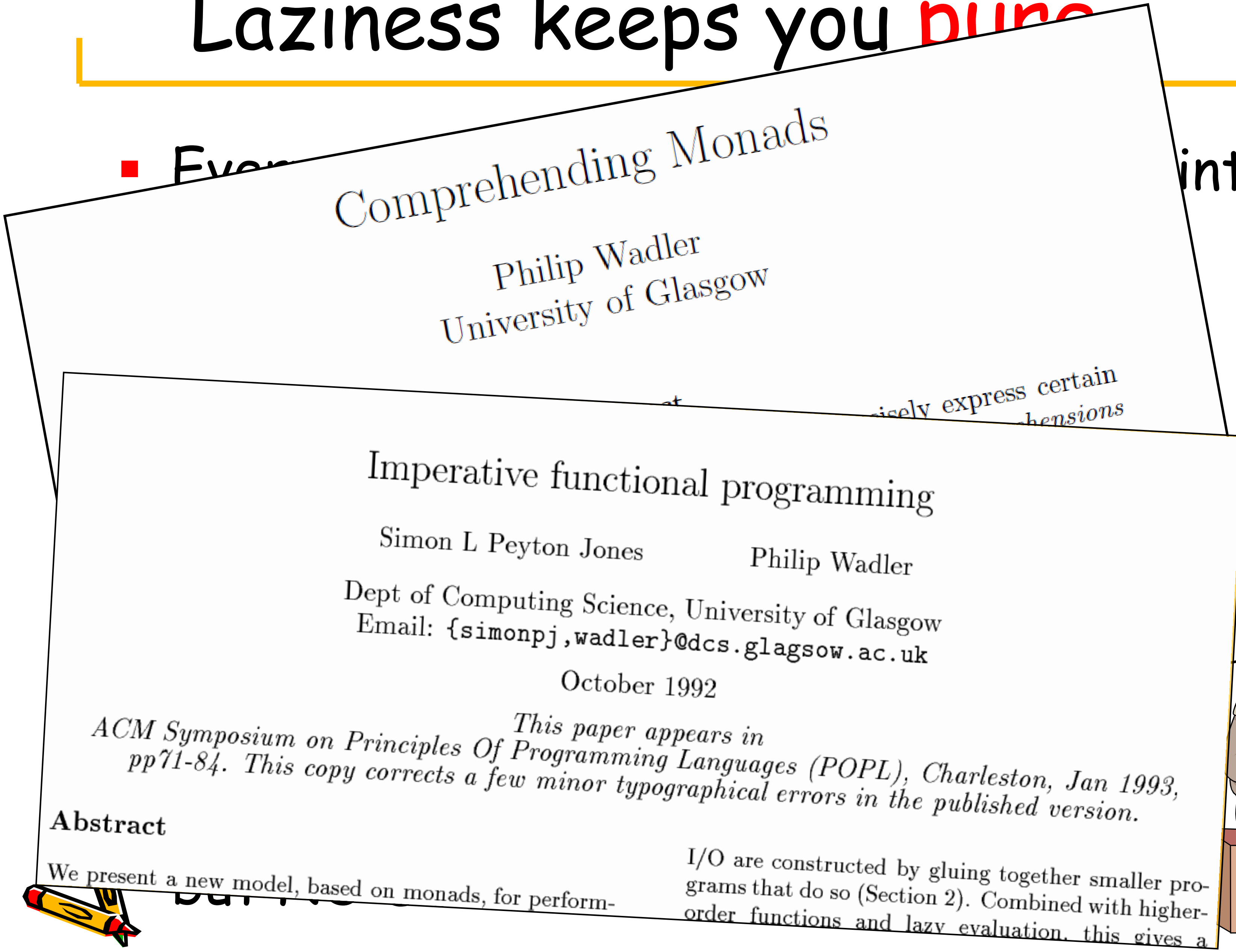
- So effects (I/O, references, exceptions) are just not an option.
- Result: **prolonged embarrassment**. Stream-based I/O, continuation I/O... but NO DEALS WIH THE DEVIL



Enter Phil Wadler

Laziness keeps you pure

- Ever… into



Comprehending Monads

Philip Wadler
University of Glasgow

…cisely express certain …hensions

Imperative functional programming

Simon L Peyton Jones Philip Wadler

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

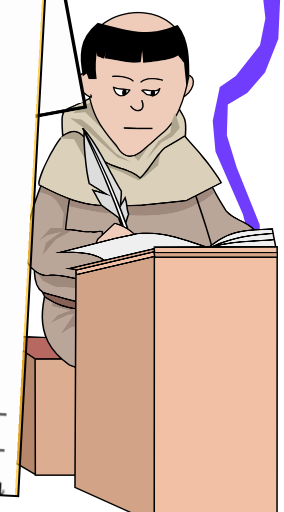
October 1992

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



- But …

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.

```
toUpper :: Char -> Char
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))))
```

What have we achieved?

- The ability to mix imperative and purely-functional programming, without ruining either: the types keep them separate
- Benefits for
  - understanding
  - maintenance
  - testing
  - parallelism

Purity by default

effects are a little inconvenient

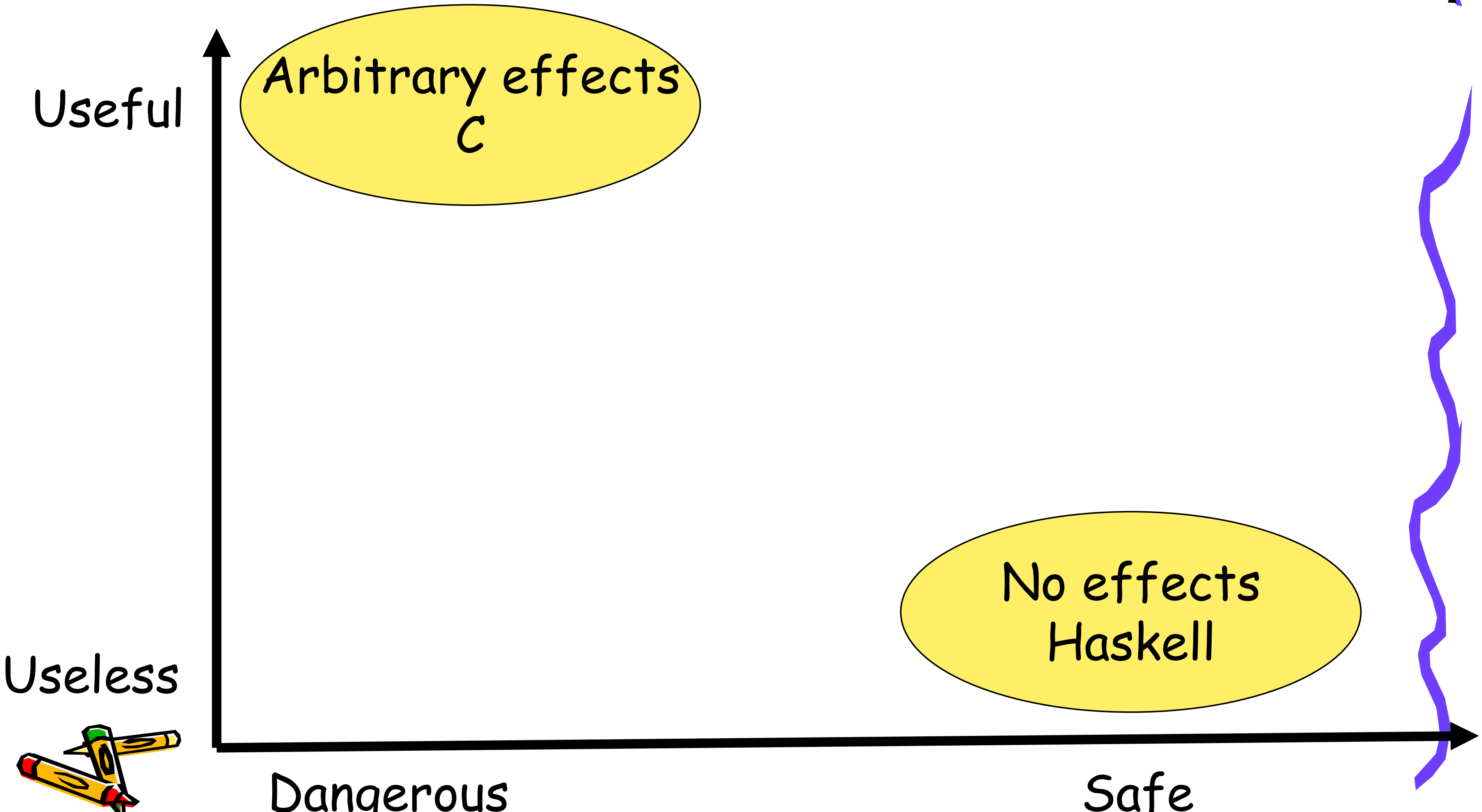
Our biggest mistake

Using the scary term "monad"

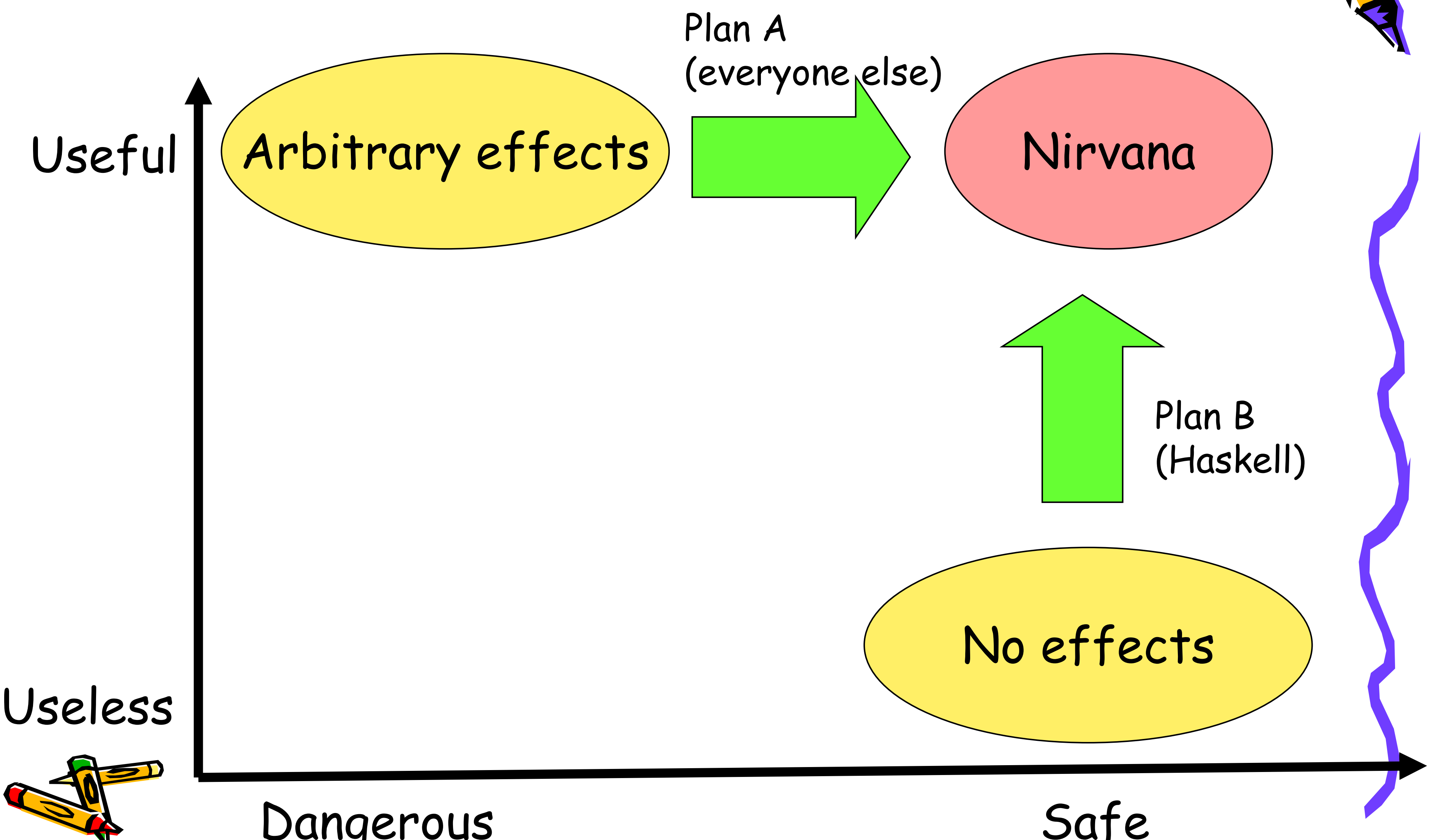
rather than

"warm fuzzy thing"

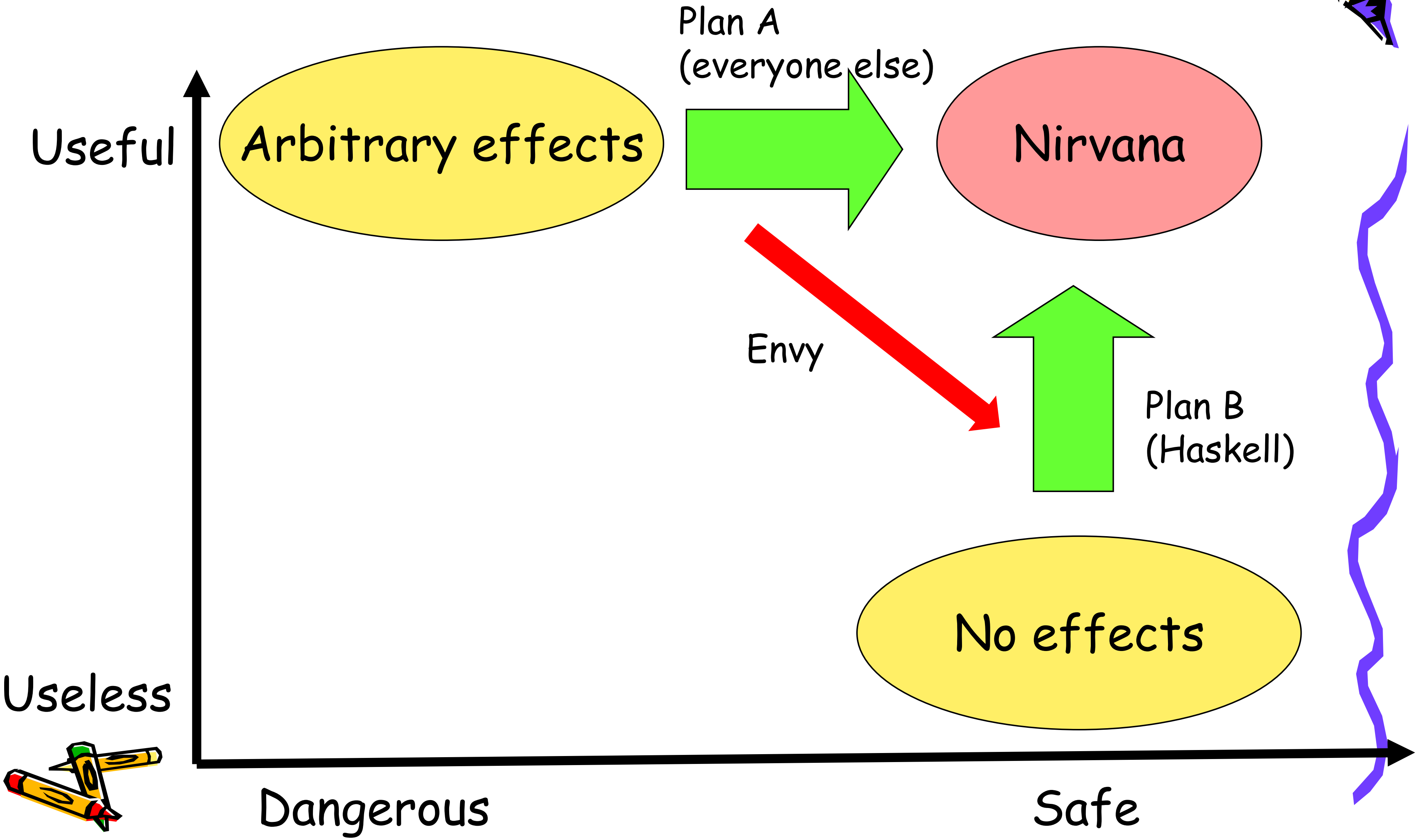
The challenge of effects



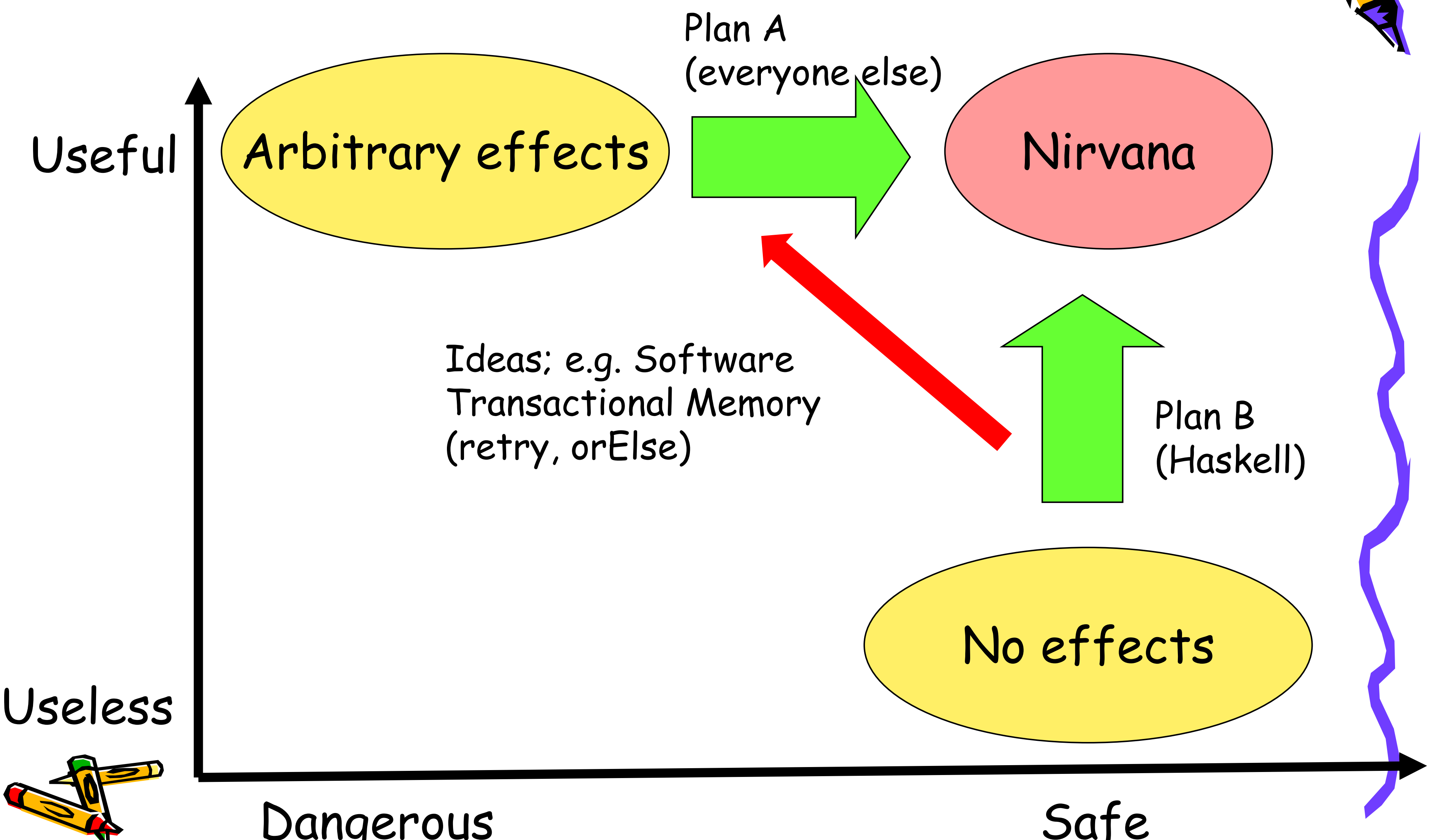
The challenge of effects



Lots of cross-over



Lots of cross-over



SLPJ conclusions

- One of Haskell's most significant contributions is to **relentlessly pursue purity** and see where takes us
- 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

Haskell committee hated this, but had no idea what else to do

Enter Phil Wadler (again)

The birth of type classes

==

From: Philip Lee Wadler <plw@cs.glasgow.ac.uk>
Date: Sat, 27 Feb 88 15:33:30 GMT
To: bob@lfcs.ed.ac.uk, fplangc@cs.ucl.ac.uk, mads@lfcs.ed.ac.uk,
plw@cs.glasgow.ac.uk
Subject: Overloading in Haskell
Sender: fplangc-request@cs.ucl.ac.uk

Proposal: Overloading in Haskell
Phil Wadler
24 February 1988

Overloading was a topic that sparked much discussion at the Yale meeting. It seemed clear that if the language was to be usable, we would at least need overloading of operations such as "+" and "*". The overall philosophy of the language suggested that we should do this in as general a way as possible, rather than just as a special case for a few operators. There appeared to be no easy "off-the-shelf" solution available for us to use.

A worrying point was exemplified by the definition

```
square x  =  x * x
```

Since "*" applies to values of both type "int" and type "float", shouldn't "square" apply to both as well? Clearly this was desirable, but we could see no easy way to achieve it. (The simplest method leads to a potential blow-up when the original source with overloading is translated to a core language with overloading removed.)

Another source of discussion was the "polymorphic equality" operator. The "polymorphic equality" operation found in Standard ML and Miranda is, from some perspectives, an odd beast. Standard ML requires an extension to the type system, "equality types", to guarantee, for example, that two functions are never compared for equality. Further, polymorphic equality is not "lambda definable"---it must be defined as a new primitive. This poses problems for some implementations, such as

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... **...the compiler generates this**

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

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

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

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

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

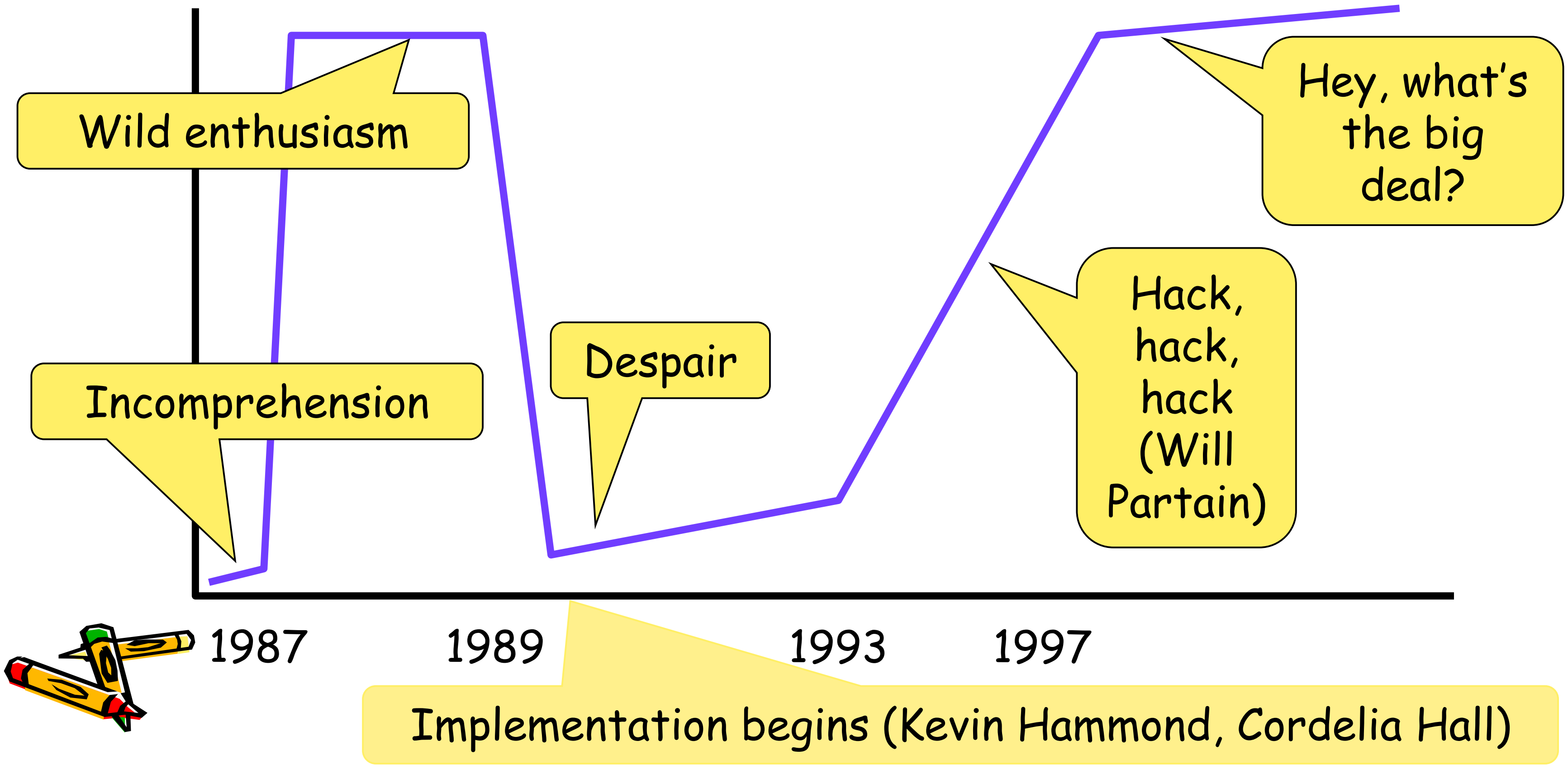
The class decl translates to:

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

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

Type classes over time

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



Will Partin, Jim Mattson, Cordelia Hall, Kevin Hammond

Date: Tue, 14 Feb 1995 09:28:06 +0000
From: Jim Mattson <mattson@dcs.gla.ac.uk>

> I've successfully made GHC 0.23 under Solaris 2.3
> using the .hc files
> and two quick hacks to the C code. Yet my attempts to rebuild to
> produce a native code generator have been stymied.

Poor wee soul. I hate to see you suffer like this. Don't do anything. I will devote the day to intense self-flagellation. By the time you wake up, there will either be a Solaris binary for GHC 0.24, or one less Research Assistant on the Aqua project.

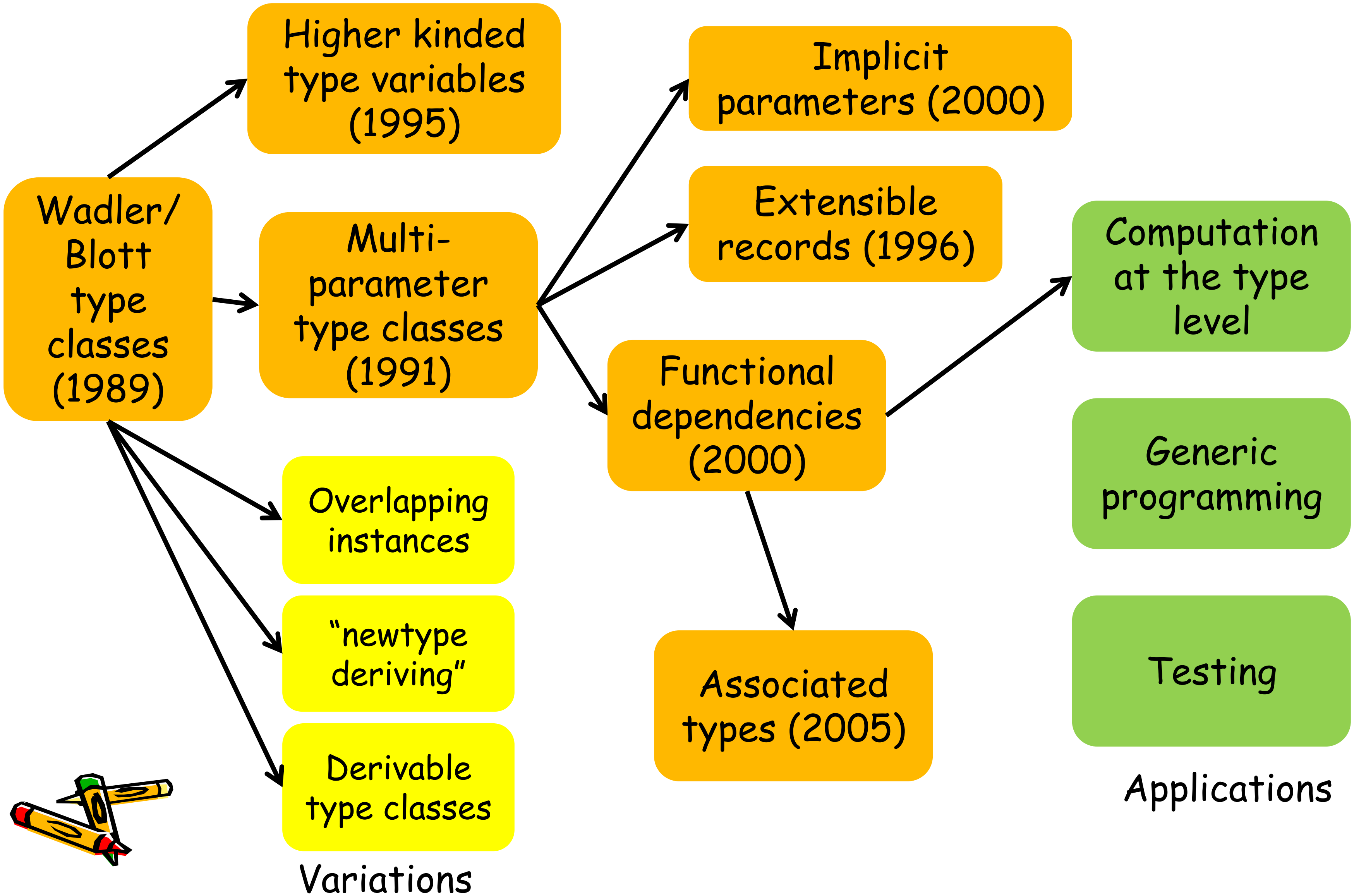
Type classes have proved extraordinarily convenient in practice

- 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....

Type-class fertility



Beyond type classes

Haskell has become a laboratory and playground for advanced type systems

- Higher kinded type variables

```
data T f a = T a (f (T k f))
    -- f :: * -> *
```

- Allows new forms of abstraction

```
f ::[a] -> [a]
f :: Monad m => m a -> m a
f :: (Profunctor p, Monad m) => p (m a) (m a)
```

Beyond type classes

Haskell has become a laboratory and playground for advanced type systems

- Polymorphic recursion
- Kind polymorphism

```
data S f a = S (f a)
  -- S :: ∀k. (k->*) -> k -> Type
```

- Polymorphic functions as function arguments (higher ranked types)

```
f :: (forall a. [a]->[a]) -> ...
```

- Existential types

```
data T = exists a. Show a => MkT a
```

Beyond type classes: 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
```

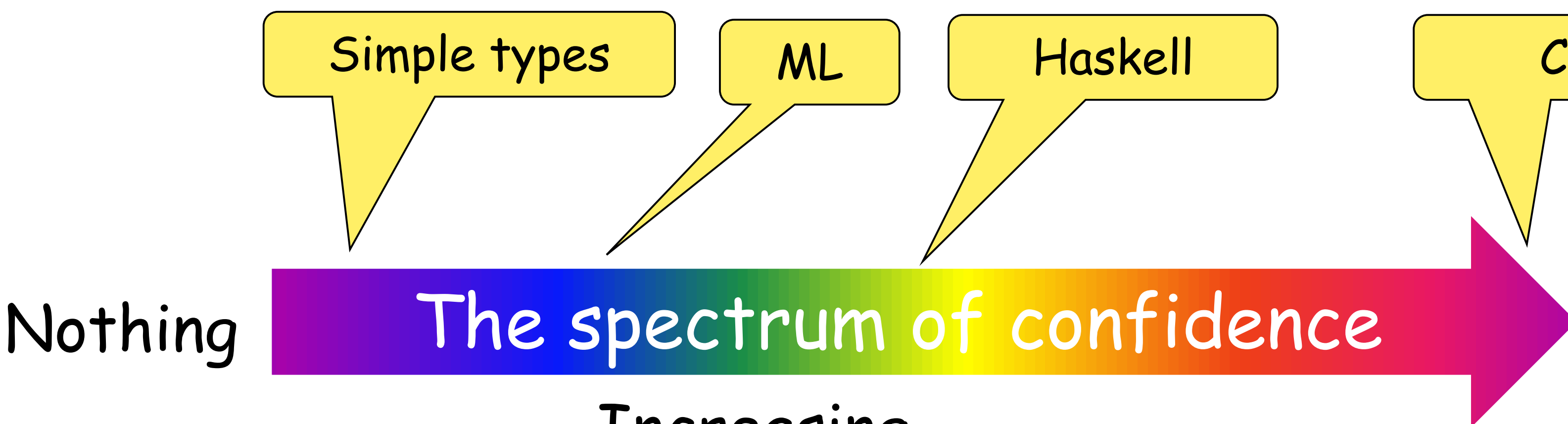
- Type families and associated types

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

- Data 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

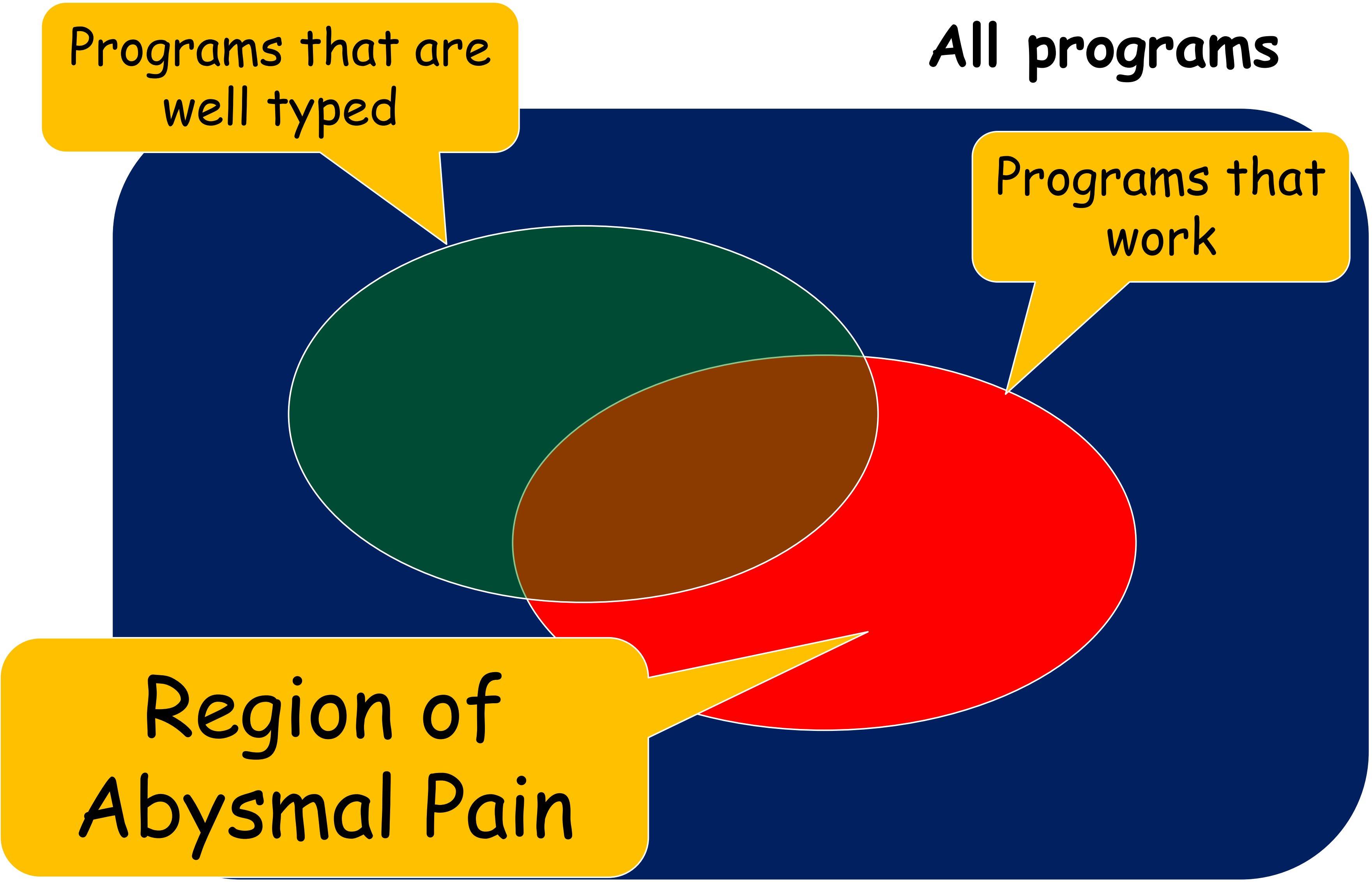


Hammer
(cheap, easy to use, limited effectivenes)

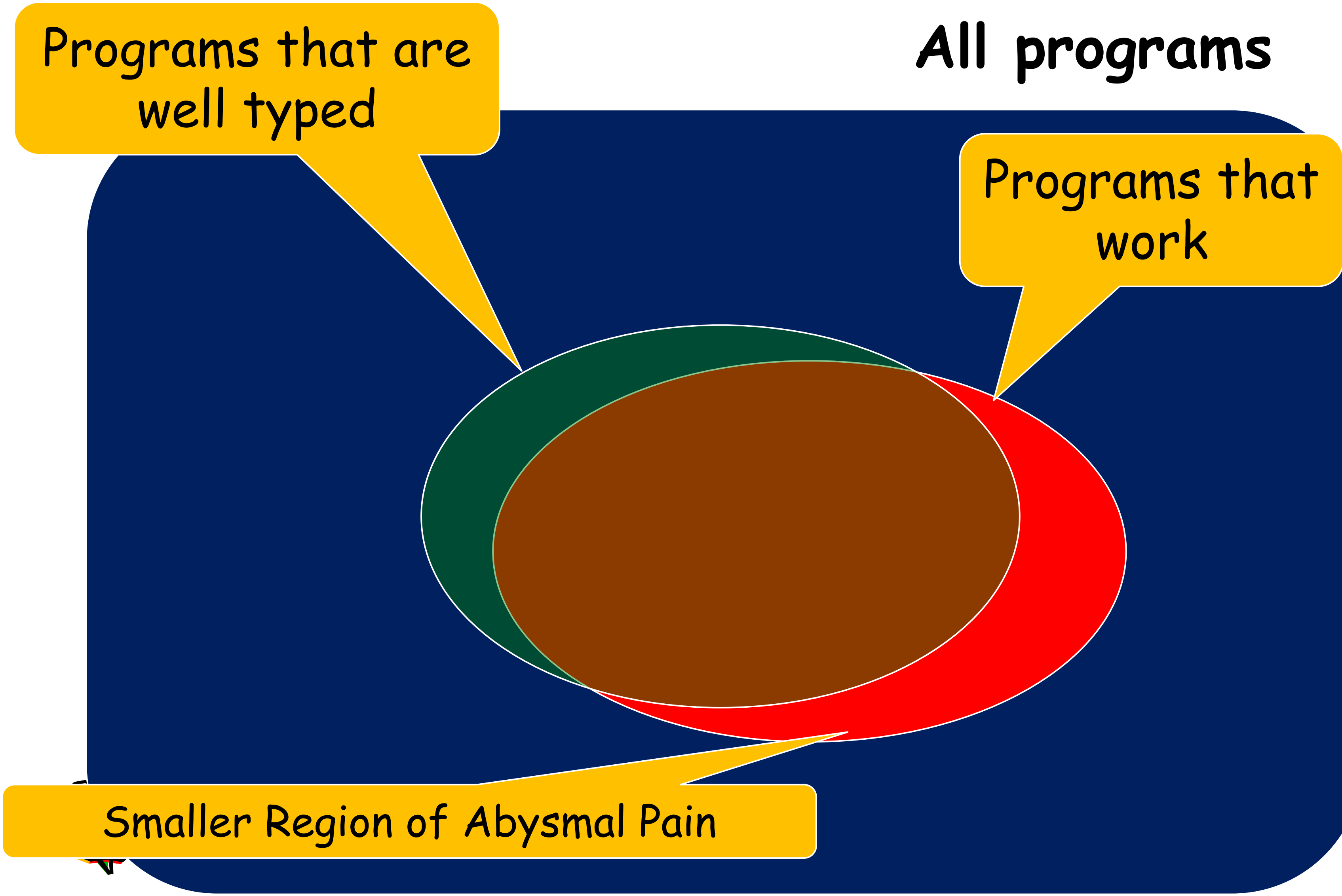
Increasing confidence that the program does what you want

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

Bad type systems



Sexy type systems



Plan for World Domination

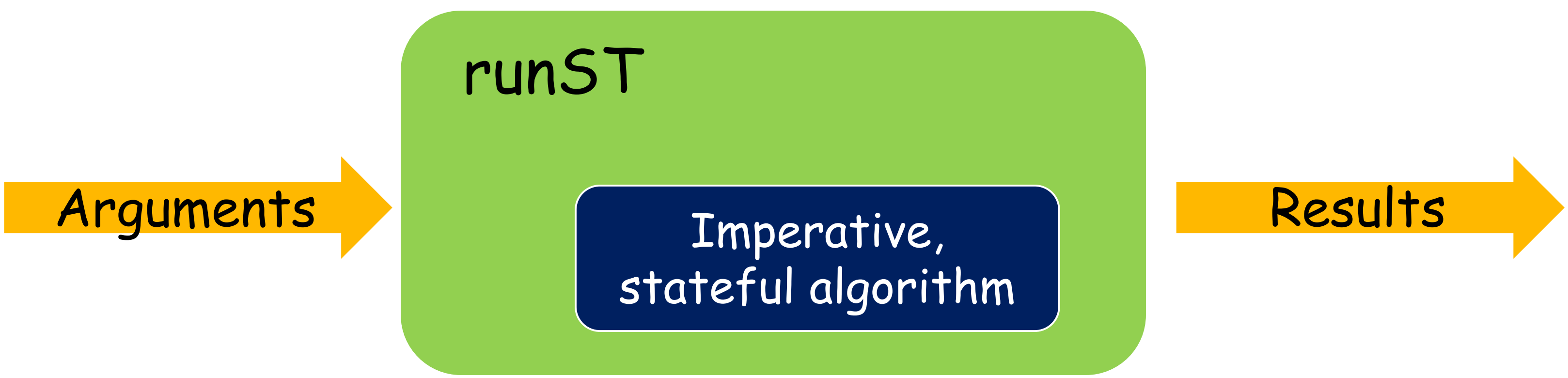
- Build on the demonstrated success of static types
- ...by making the type system more expressive
- ...so that more good programs are accepted (and more bad ones rejected)
- ...without losing the Joyful Properties (comprehensible to programmers)

Encapsulating it all (hail, John Launchbury)

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

Stateful computation

Pure result



A pure function

Encapsulating it all (hail, John Launchbury)

```
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

Escape from the ivory tower

- 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)

Fun

- Haskell is rich enough to be very useful for real applications
- But above all, Haskell is a language in which people **play**
  - Embedded domain-specific languages (animation, music, probabilistic, quantum, security...)
  - Programming as an art form (Conal Elliot, Dan Piponi...)
- Play leads to new discoveries
- And it's fun...

Luck and friendship

- 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

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