Haskell: practical as well as cool

John Hughes, Simon Peyton Jones, Philip Wadler

December 2012

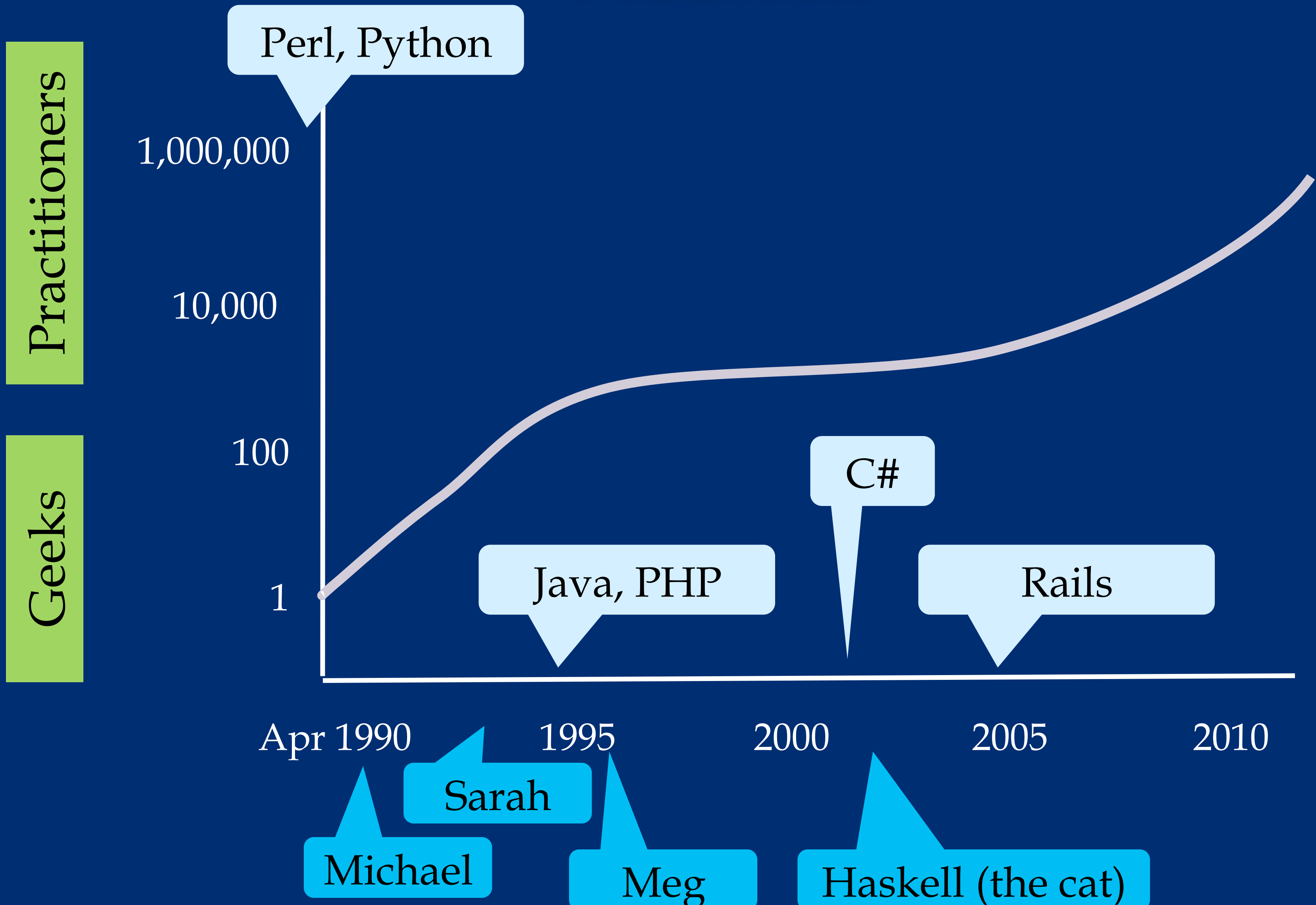
What is Haskell?

- A functional language
  - Purely functional
  - Lazy
  - Statically typed
- Designed 1988-1990
- For research, teaching, and practical use
- By a committee of academics


WG2.8 1992
greenbelt

WG2.8 1992

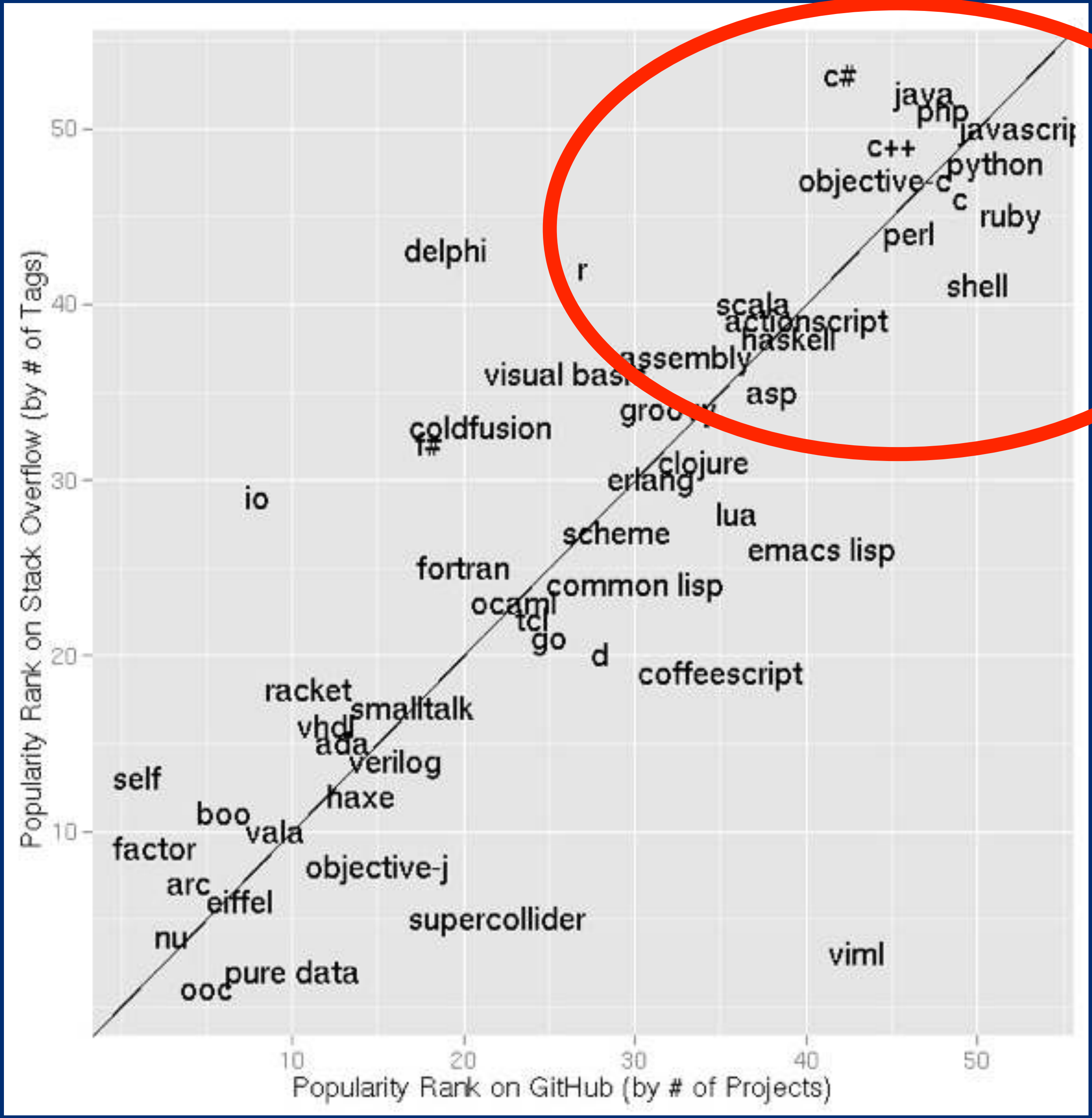
Haskell

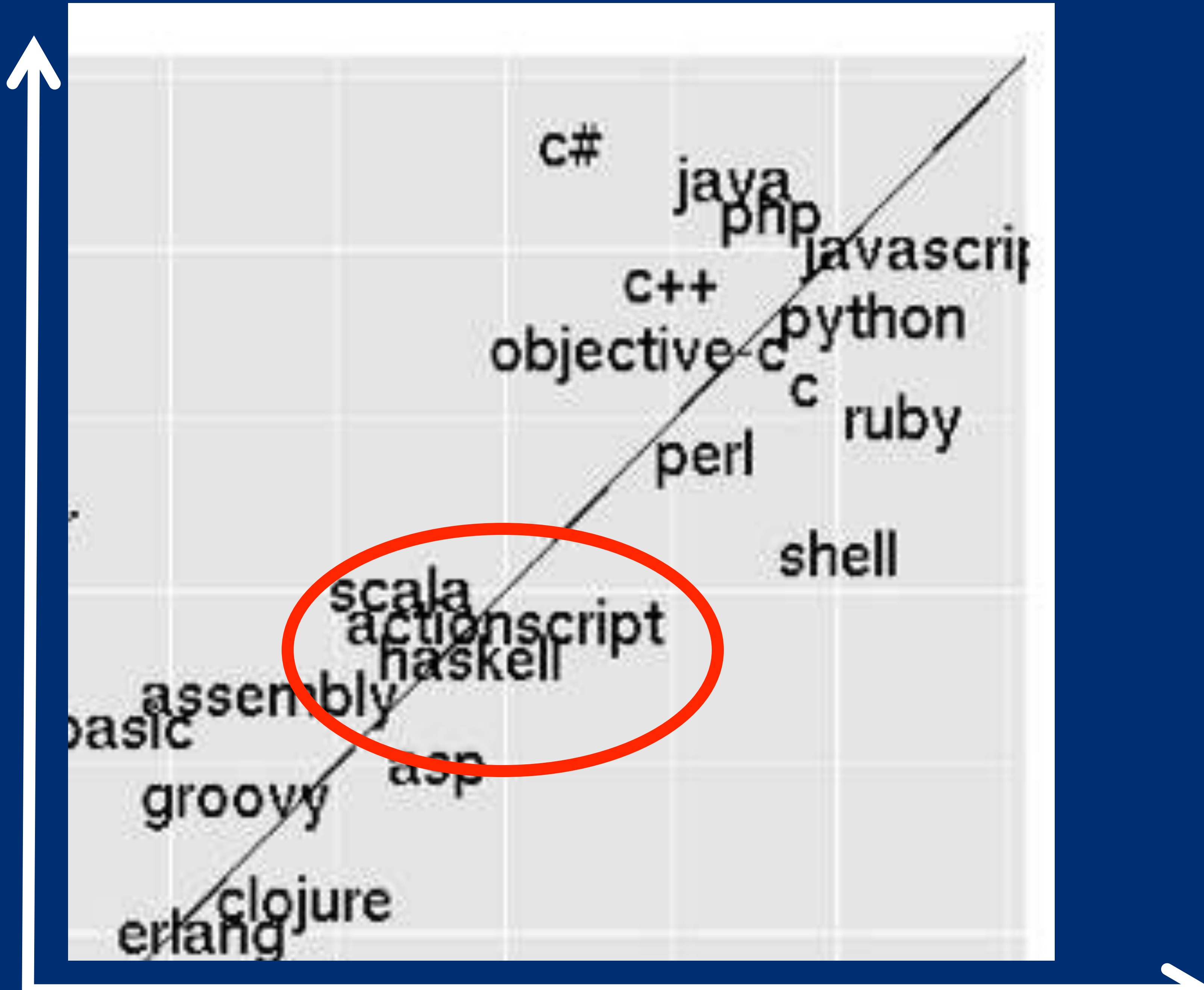


Practitioners
Geeks
Perl, Python
1,000,000
10,000
100
1
C#
Java, PHP
Rails
Apr 1990
1995
2000
2005
2010
Sarah
Michael
Meg
Haskell (the cat)


Haskell the cat (b. 2002)

http://redmonk.com Sept 2012





StackOverflow, # of tags

GitHub, # of projects

Why does Haskell have such a big mind share?

Keep faith with a few deep, simple principles, and see where they lead

- Purity
- Domain specific languages
- Types

"People will gladly adapt to the limitations of a great design."
Don Box

Purity

Any effect

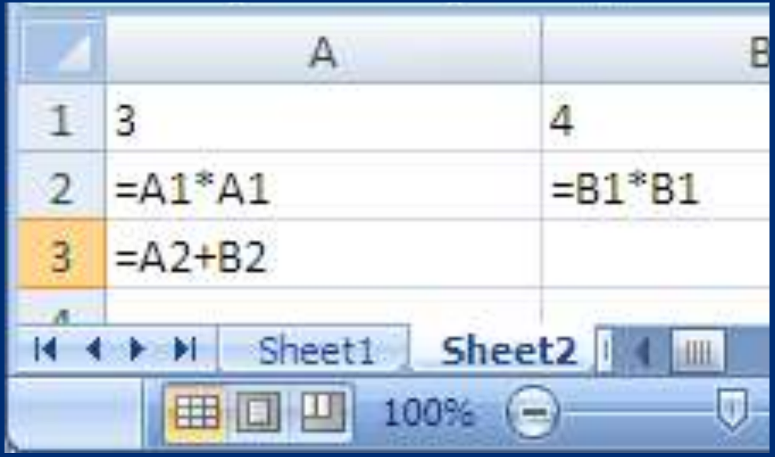
Spectrum

Pure (no effects)

C, C++, Java, C#, VB

Excel, Haskell

```
X := In1
X := X*X
X := X + In2*In2
```



Commands, control flow

Expressions, data flow

- Do this, then do that
- "X" is the name of a cell that has different values at different times

- No notion of sequence
- "A2" is the name of a (single) value

Any effect ⟵ Spectrum ⟶ Pure (no effects)

C, C++, Java, C#, VB	Excel, Haskell

Side effects are how computation is done

No side effects at all

- Do this, then do that
- "X" is the name of a cell that has different values at different times

- No notion of sequence
- "A2" is the name of a (single) value

BUT: side effects are useful

- I/O is a side effect. So side effects are **part of the specification** of what we want.

Result

Prolonged embarrassment

Laziness keeps yo… e

- Eve…

Comprehending Monads

Philip Wadler
University of Glasgow

…isely 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

Salvation through types

```
reverse :: [Char] -> [Char]
toUpper :: Char -> Char
useless :: () -> ()

getChar :: FileHandle -> IO Char
launchMissiles :: IO ()
```

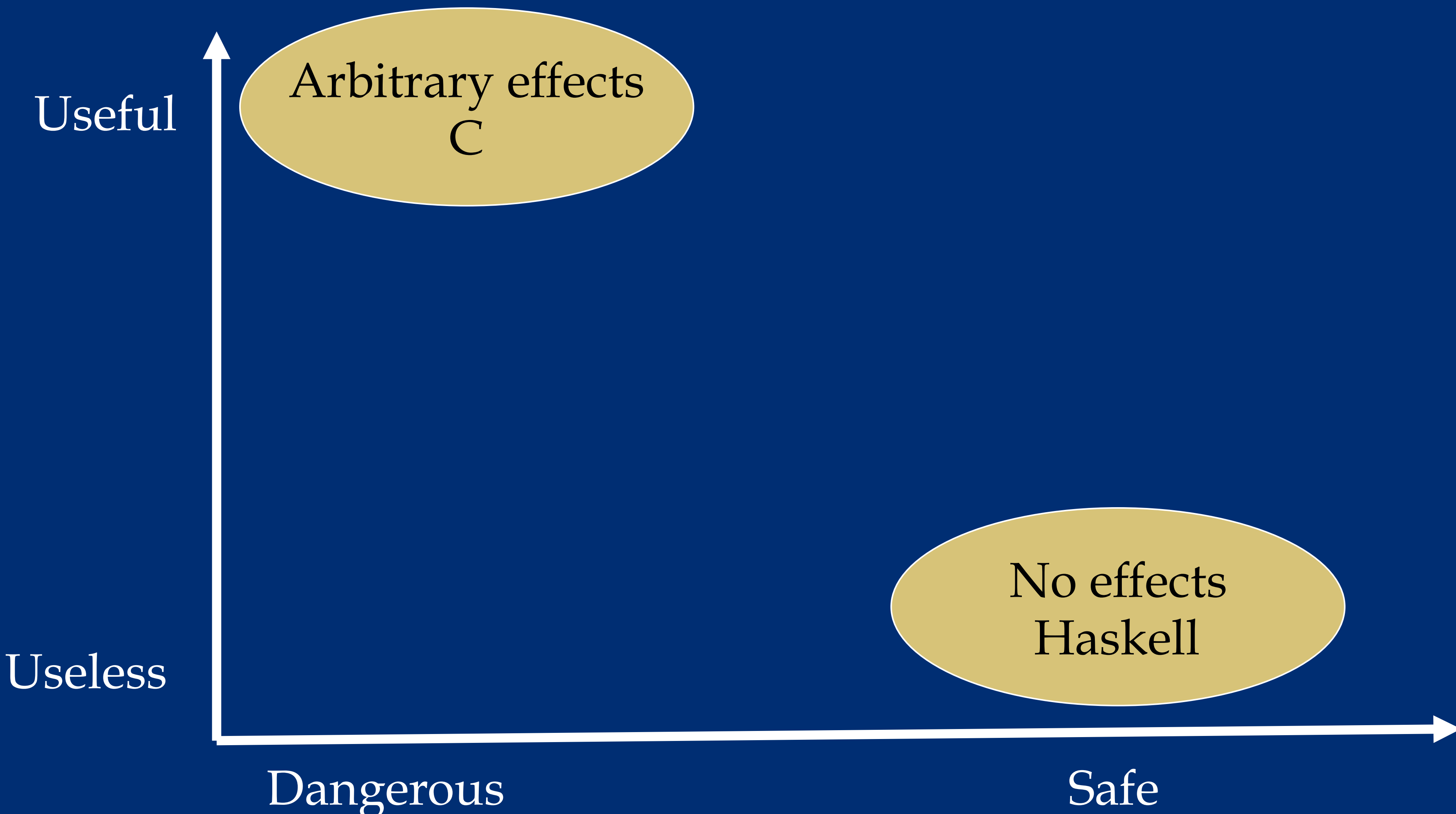
No side effects

I/O effects

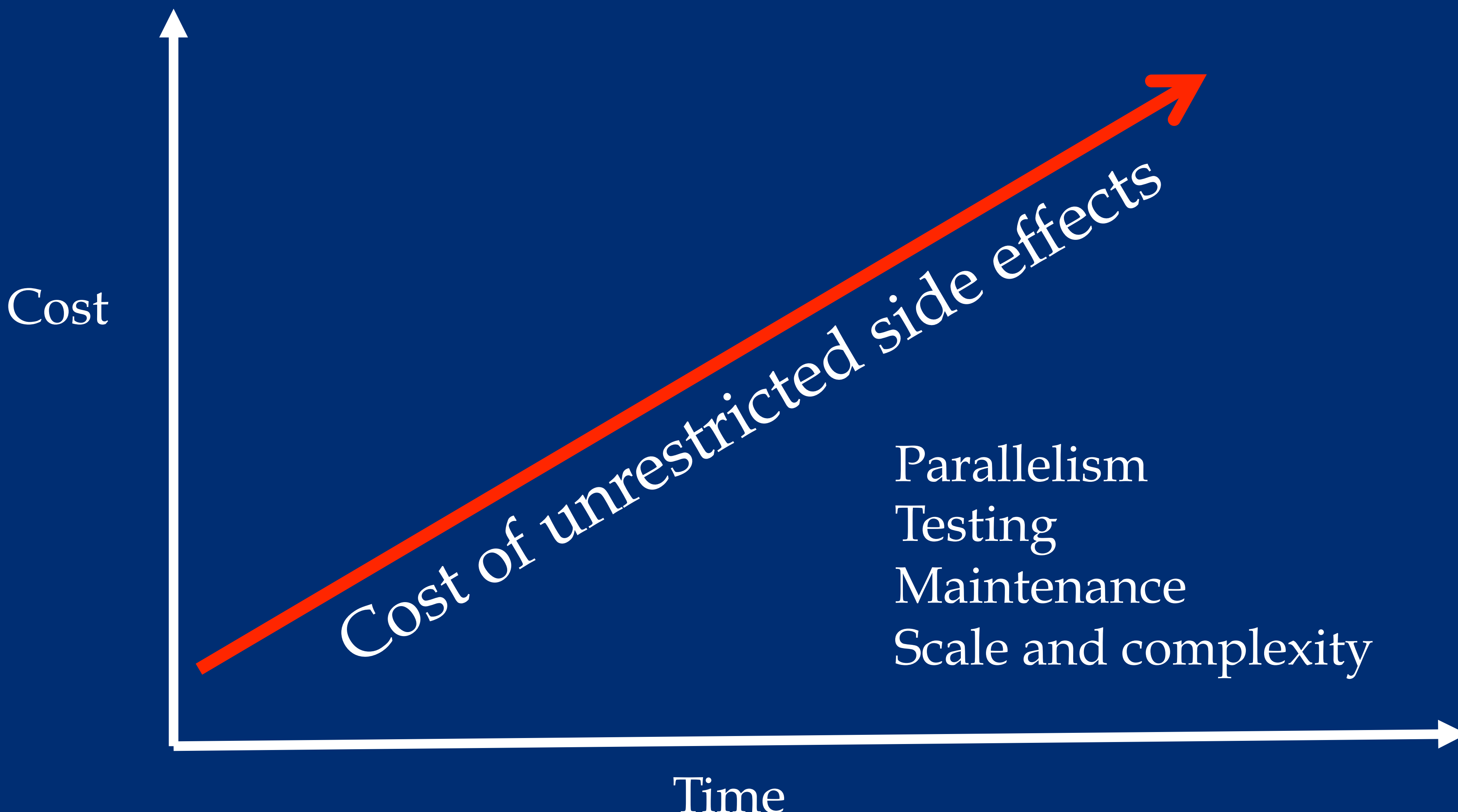
International side effects

Pure by default
Side effects where necessary

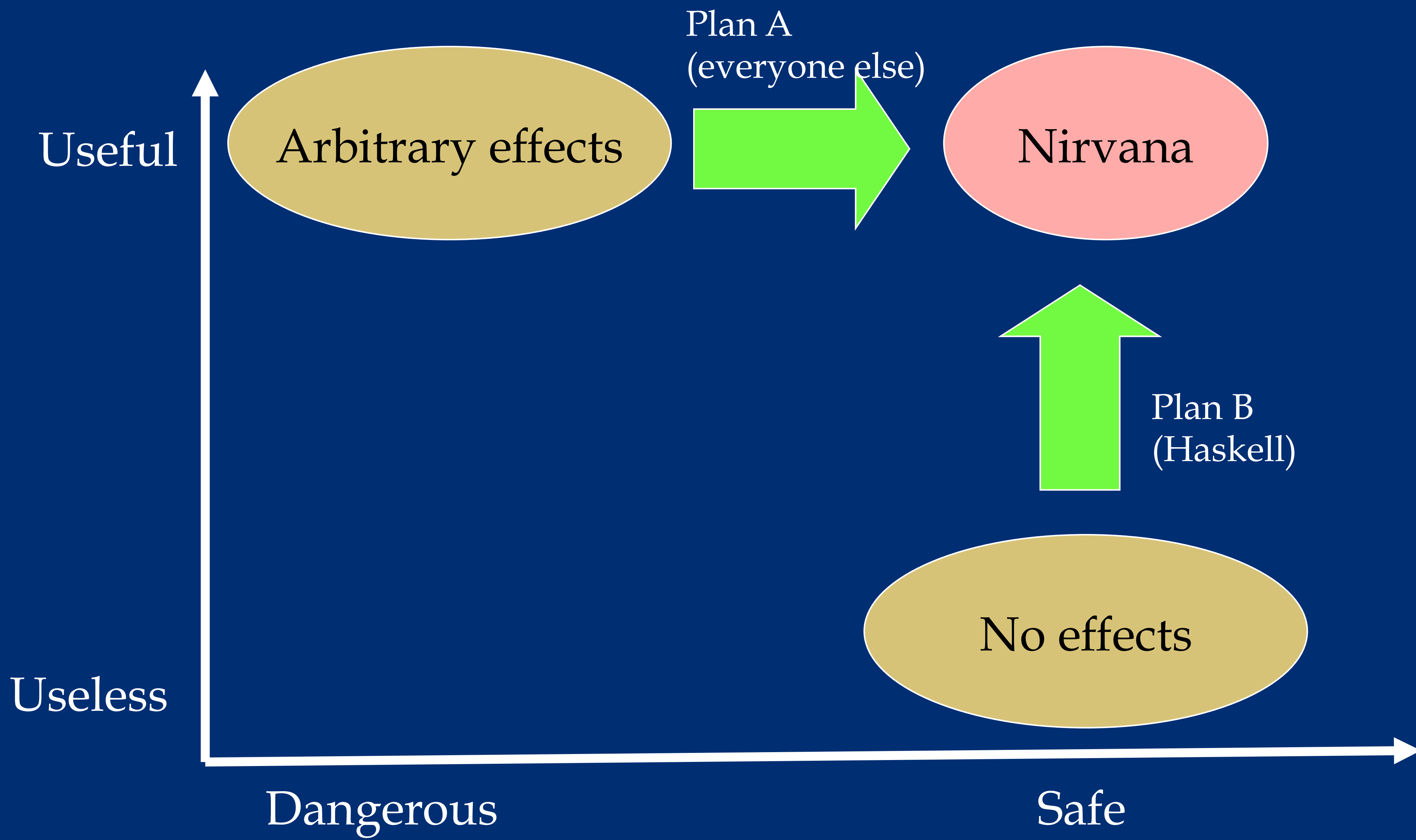
The challenge of effects



The challenge of effects



The challenge of effects



Domain specific languages

Goal
The program expresses as directly as possible what is the mind of the domain expert


```
b1 = addDur dqn [b 3, fs 4, g 4, fs 4]
b2 = addDur dqn [b 3, es 4, fs 4, es 4]
b3 = addSur dqn [as 3, fs 4, g 4, fs 4]
bassLine = timesM 3 b1 :+: timesM 2 b2 :+:
           timesM 4 b3 :+: timesM 5 b1
```

Embedded domain specific languages

- An EMBEDDED domain-specific langauge is just a library, whose API embodies the domain knowledge
- 80% of the benefit for 20% of the effort
- Haskell is particularly good at this, because of types, laziness, syntax.

EDSLs in Haskell

- Hardware description language (Lava)
- Orchestration (Orc)
- Reactive animations (Fran)
- Workflow
- Diagrams (disgrams-cairo)
- Financial contracts
- Data-parallel (Repa)
- Hard real-time applications (Atom)
- URLs, routes, MongoDB schema, database queries, HTML (Yesod)
- Parsers (Parsec)
- XML (HaXml)
- GPUs (Nicola, Accelerate)
- Test-case generation (Quickcheck)

Types

Types are wildly successful

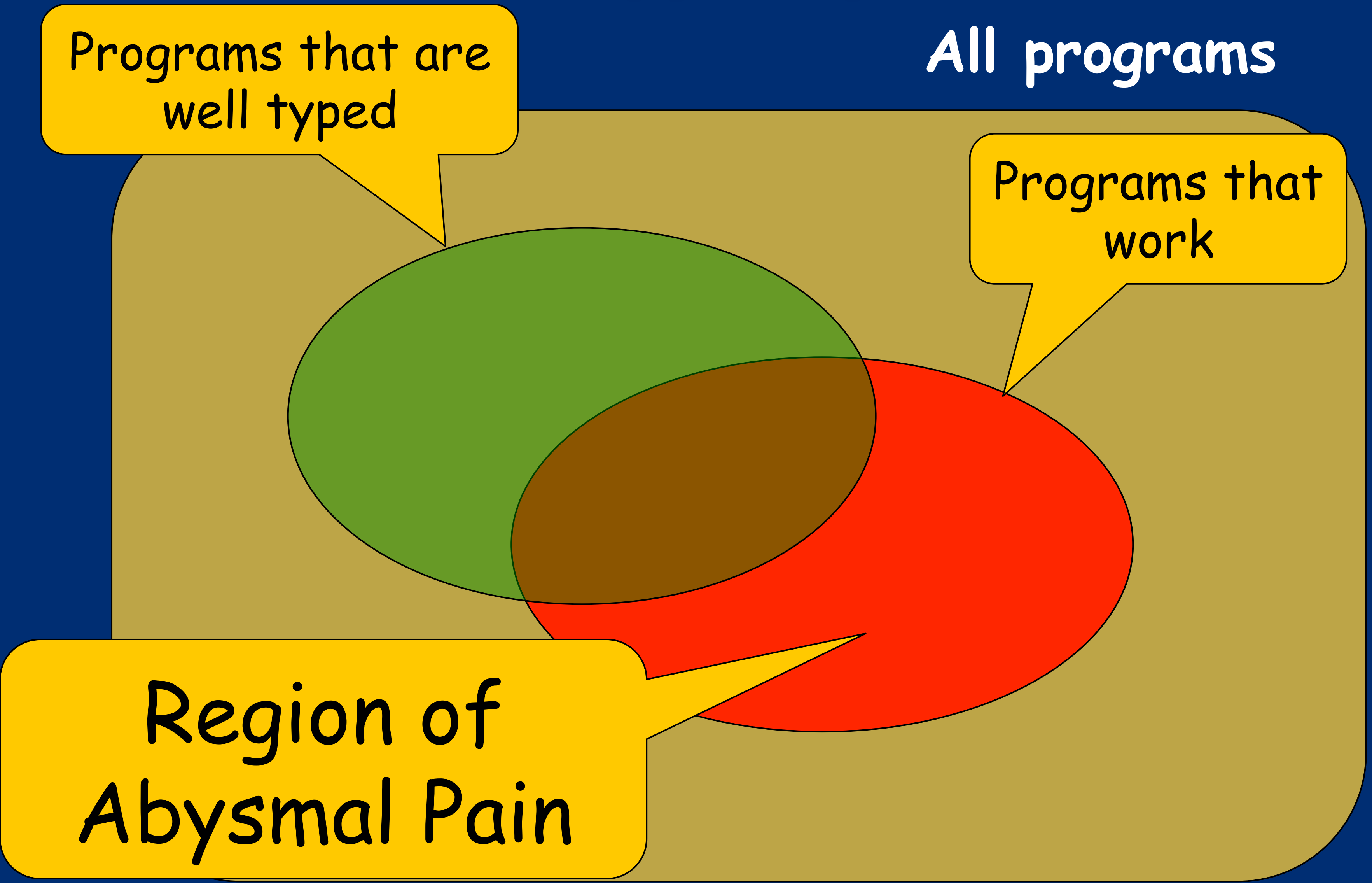
Static typing is by far the most widely-used program verification technology in use today: particularly good cost/benefit ratio

- Lightweight (so programmers use them)
- Machine checked (fully automated, every compilation)
- Ubiquitous (so programmers can't avoid them)

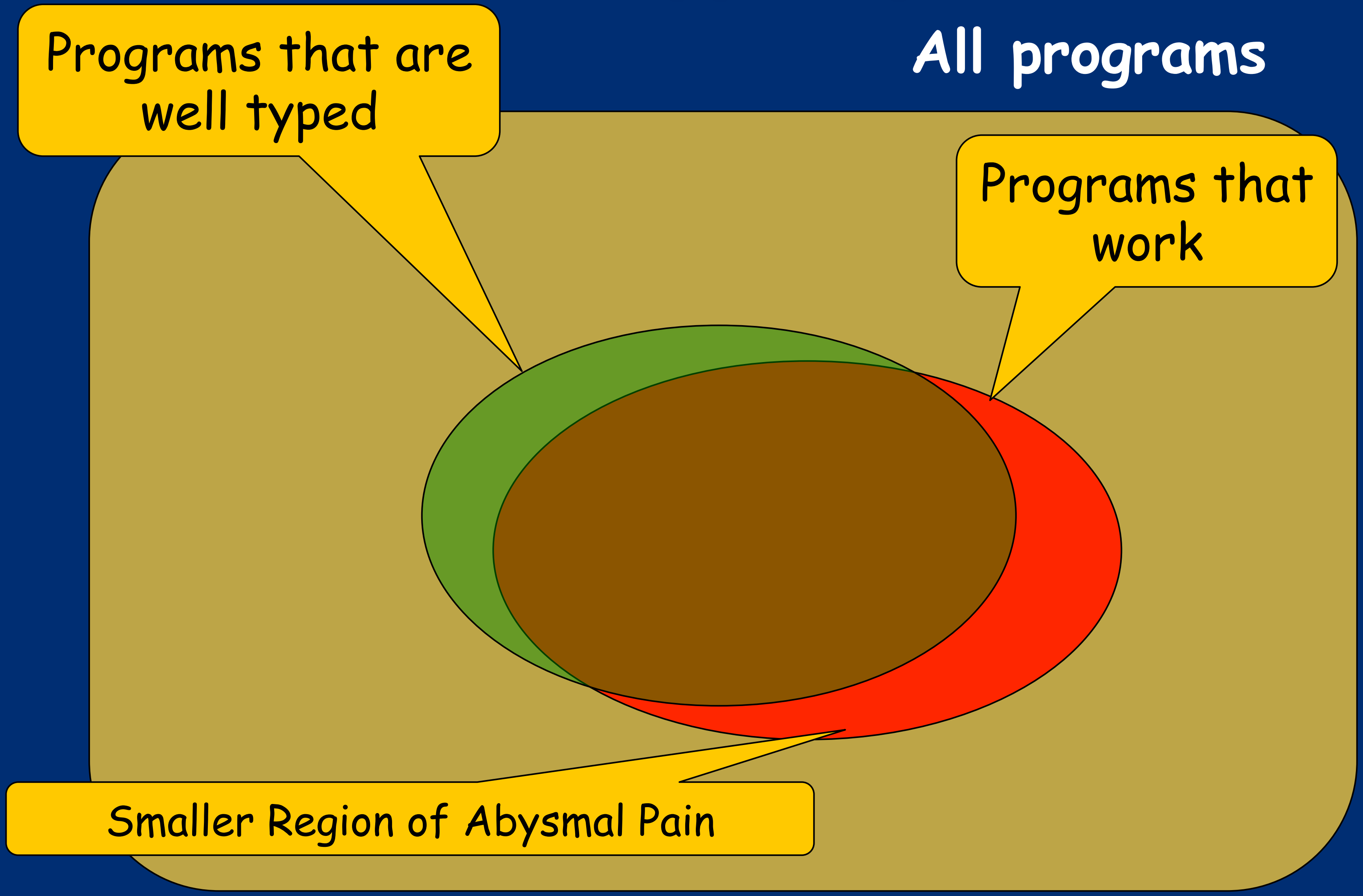
The joy of types

- [Old hat] Types guarantee the absence of certain classes of errors: "well typed programs don't go wrong"
  - True + 'c'
  - Seg-faults
- Types are a **design language**; types are the UML of Haskell
- The BIGGEST MERIT (though seldom mentioned) of types is their support for **software maintenance**

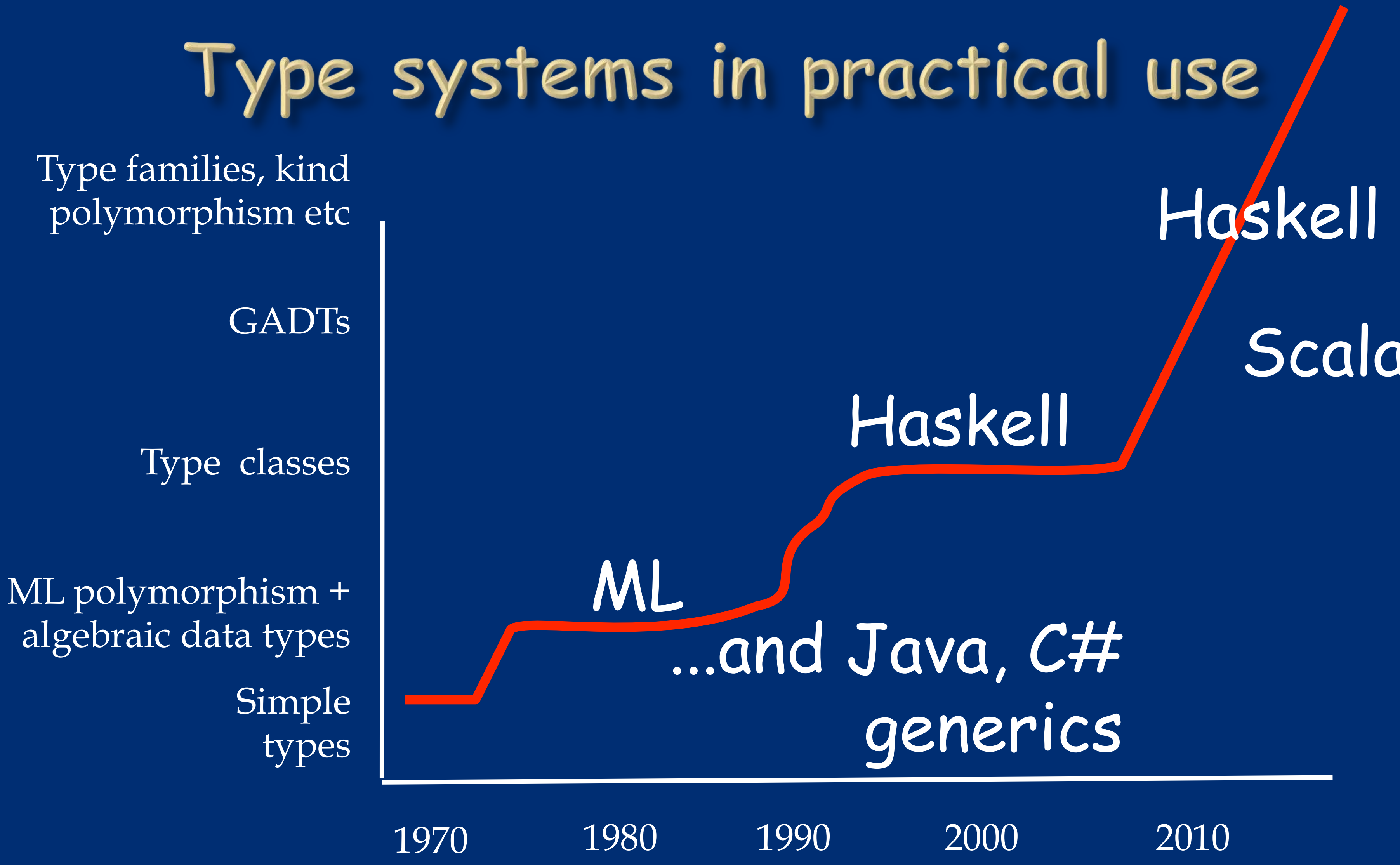
Bad type systems



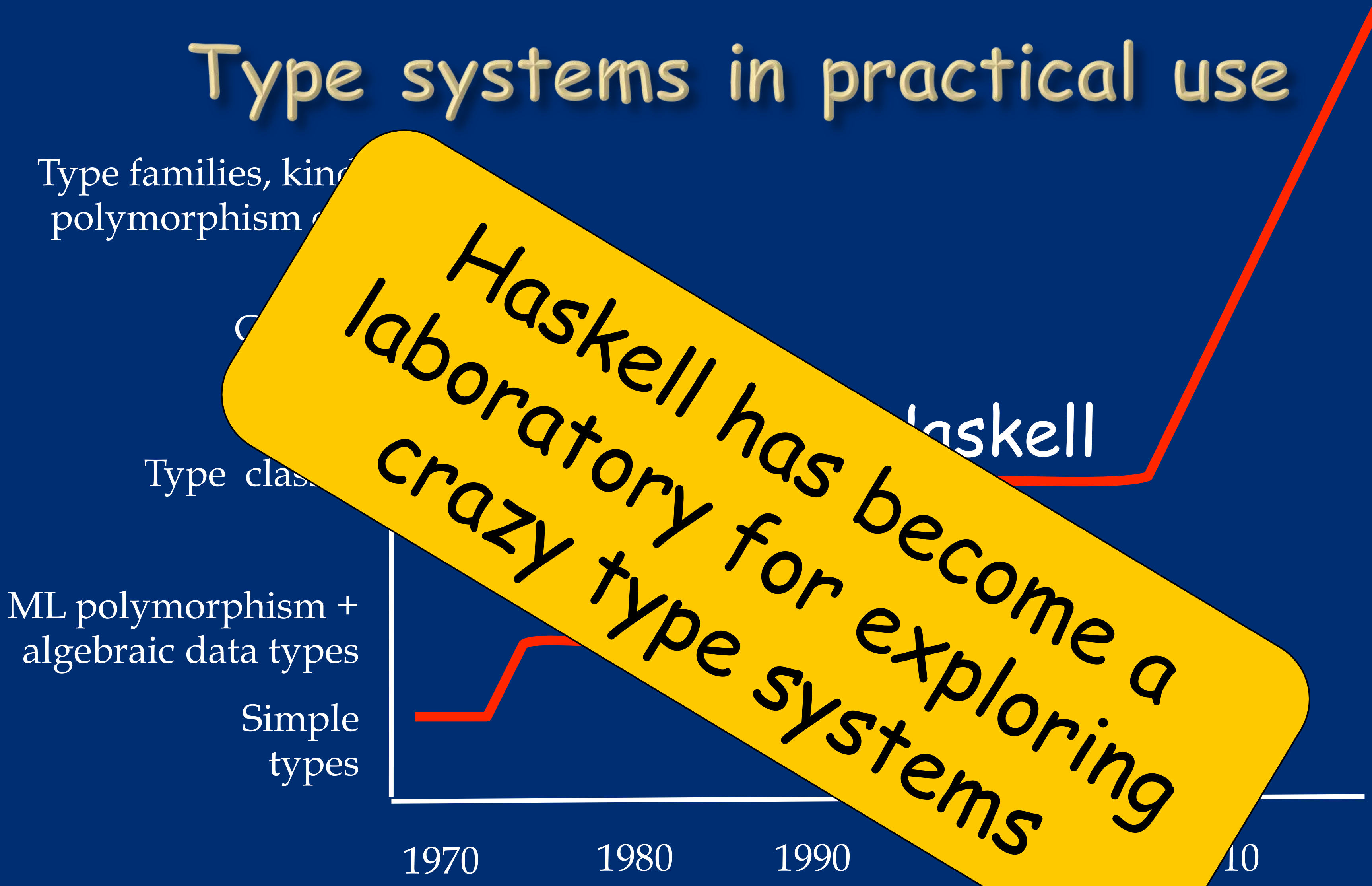
Better type systems



Type systems in practical use



Type systems in practical use



Haskell has become a laboratory for exploring crazy type systems

Transactions in Haskell

The context

- A web server
  - Lots of independent, I/O-performing threads
  - With shared state
- GHC's runtime natively supports super-lightweight threads
- But: how to control access to shared state?
- Usual answer: **locks and condition variables**

What's wrong with locks?

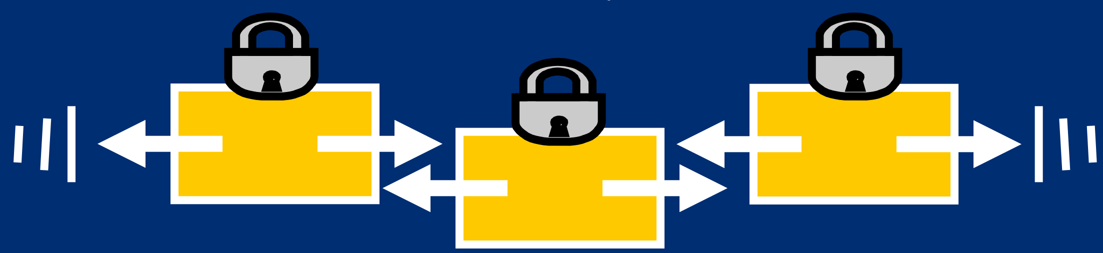
A 10-second review:

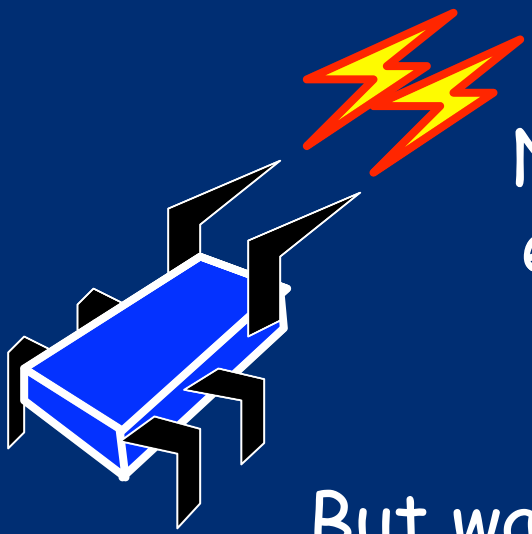
- **Races**: due to forgotten locks
- **Deadlock**: locks acquired in "wrong" order.
- **Lost wakeups:** forgotten notify to condition variable
- **Diabolical error recovery**: need to restore invariants and release locks in exception handlers

- These are serious problems. But even worse...

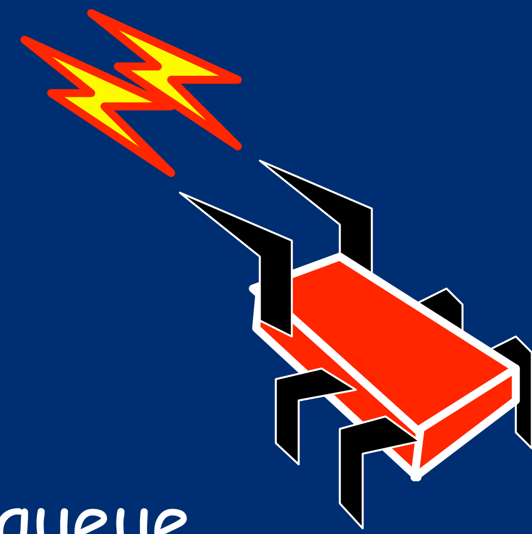
Locks are absurdly hard to get right

Scalable double-ended queue: one lock per cell





No interference if ends "far enough" apart



But watch out when the queue is 0, 1, or 2 elements long!

Locks are absurdly hard to get right

Coding style	Difficulty of concurrent queue
Sequential code	Undergraduate

Locks are absurdly hard to get right

Coding style	Difficulty of concurrent queue
Sequential code	Undergraduate
Locks and condition variables	Publishable result at international conference

Atomic memory transactions

Coding style	Difficulty of concurrent queue
Sequential code	Undergraduate
Locks and condition variables	Publishable result at international conference
Atomic blocks	**Undergraduate**

Atomic memory transactions

atomically { ... sequential get code ... }

- To a first approximation, just write the sequential code, and wrap **atomically** around it
- All-or-nothing semantics: **Atomic** commit
- Atomic block executes in **Isolation**
- Cannot deadlock (there are no locks!)
- Atomicity makes error recovery easy (e.g. exception thrown inside the **get** code)

ACID

Transactional memory

```
do { atomically (...increment Fred's account
                 ...decrement Bill's account...)
   ; print receipt
   ; launch missiles  }
```

	Outside atomically	Inside atomically
Input/output	Yes	NO
Deposit or withdraw	NO	Yes

```
atomically :: STM a -> IO a
```

TM effects only

Arbitrary I/O effects

Why does STM fit Haskell so well?

- **Efficient**: side effects are the exception, not the rule => efficient
- **Secure**
  - type system (without modification) keeps STM effects separate from I/O effects
  - no possibility of modifying transactional variables outside transactions
- **Compositional**: a little DSL for describing transactions

```
atomically   :: STM a -> IO a
retry        :: STM a
orElse       :: STM a -> STM a -> STM a
throw        :: Exception -> STM a
```

STM Conclusion

- Purity, supported by types, allows us to build a domain specific language for describing composable transactions.

Haskell
The world's finest imperative programming language

backup slides (put at end)

Mutable state

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

```
main = do { r <- newRef 0
          ; incR r
          ; s <- readRef r
          ; print s }

incR :: Ref Int -> IO ()
incR r = do { v <- readRef r
            ; writeRef r (v+1)
            }
```

Reads and writes are 100% explicit!

You can't say (r + 6), because r :: Ref Int

Concurrency in Haskell

```
forkIO :: IO () -> IO ThreadId
```

- forkIO spawns a thread
- It takes an action as its argument

```
webServer :: RequestPort -> IO ()
webServer p = do { conn <- acceptRequest p
                 ; forkIO (serviceRequest conn)
                 ; webServer p }

serviceRequest :: Connection -> IO ()
serviceRequest c = do { … interact with client … }
```

No event-loop spaghetti!

Coordination in Haskell

- How do threads coordinate with each other?

```
main = do { r <- newRef 0
          ; forkIO (incR r)
          ; incR r
          ; ... }

incR :: Ref Int -> IO ()
incR r = do { v <- readRef r
            ; writeRef r (v+1) }
```

Aargh!
A race

STM in Haskell

```
atomically  :: STM a -> IO a
newTVar     :: a -> STM (TVar a)
readTVar    :: TVar a -> STM a
writeTVar   :: TVar a -> a -> STM ()
```

```
incT :: TVar Int -> STM ()
incT r = do { v <- readTVar r; writeTVar r (v+1) }

main = do { r <- atomically (newTVar 0)
          ; forkIO (atomically (incT r))
          ; atomic (incT r)
          ; ... }
```

Purity and Testing

Does **NOT** read any global variables

Just does what it says on the tin —repeatably

```
reverse [1,2,3] == [3,2,1]
```

Does **NOT** modify any global state

Does **NOT** modify its argument

Purity and Properties

Pure functions have nice properties

They matter! Justify optimizations

```
reverse (reverse xs) == xs
```

```
(xs ++ ys) ++ zs == xs ++ (ys ++ zs)
```

Library functions: properties are well-known

What about *new* functions?

Properties as Tests

```
prop_reverse xs =
  reverse (reverse xs) == xs

prop_append xs ys zs =
  (xs++ys)++zs == xs++(ys++zs)
```

Heavy use of Haskell's classes!

```
Example*> prop_reverse [1,2,3]
True
Example*> quickCheck (prop_reverse::[Integer]->Bool)
+++ OK, passed 100 tests.
Example*> quickCheck (prop_append::
              [Integer]->[Integer]->[Integer]->Bool)
+++ OK, passed 100 tests.
```

Claessen and Hughes, ICFP 2000

Debugging Failures

```
prop_wrong xs =
  reverse xs == xs
```

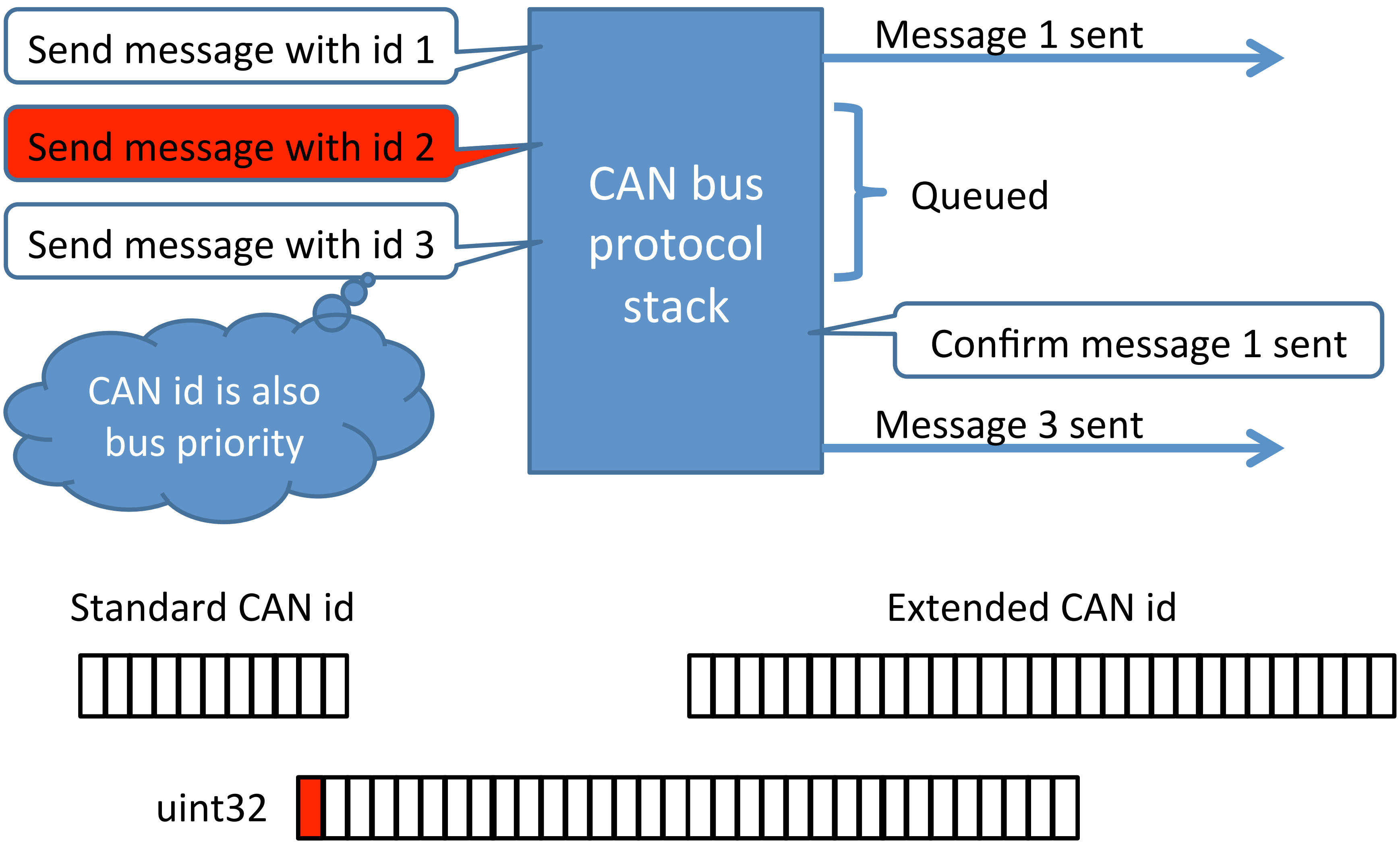
```
Example*> quickCheck (prop_wrong::[Integer]->Bool)
*** Failed! Falsifiable (after 6 tests and 5 shrinks):
[0,1]
```

A *minimal* failing test!

`[0]` passes
`[1]` passes
`[0,0]` passes

Just the necessary information to make the test fail!

A Real Bug


Send message with id 1
Send message with id 2
Send message with id 3
CAN id is also bus priority
CAN bus protocol stack
Message 1 sent
Queued
Confirm message 1 sent
Message 3 sent
Standard CAN id
Extended CAN id
uint32

QuickCheck Testing

- Less code!
  - One property generates many tests
- Better testing!
  - Combinations you'd never think to test
- Easy-to-debug minimized failing tests
- Very popular in Haskell
  - Versions for many other languages
- Found >200 bugs in software going into Volvo cars ☺

QuviQ

"Extending" Haskell

- Haskell has no **for**-loop, but…

```
forLoop i n s f
  | i >  n = s
  | i <= n = forLoop (i+1) n (f i s) f
```

- Now we can use it as

```
sumSq n = forLoop 1 n 0 (\i s -> i*i+s)
```

Used to embed *domain specific languages* in Haskell

The loop body is an anonymous function passed in to the **for** loop

Feldspar is a *program generator*

ample: Feldspar

- SL for DSP

$$\mathbf{a}\cdot\mathbf{b} = \sum_{i=1}^{n} a_i b_i$$

```
scProd' :: Numeric a =>
  Vector (Data a) -> Vector (Data a) -> Data a
scProd' a b =
  forLoop n 0 (\i s -> s + (a!i * b!i) )
  where n = min (length a) (length b)
```

```
void test(struct array * v0, struct array * v1, float * out)
{  uint32_t len0;
   float v3;
   len0 = min(getLength(v0), getLength(v1));
   (* out) = 0.0f;
   for(uint32_t v2 = 0; v2 < len0; v2 += 1)
   {  v3 = ((* out) + (at(float,v0,v2) * at(float,v1,v2)));
      (* out) = v3;
   }
}
```

Executable C code suitable for running in a radio base station!

A More "Haskellish" Scalar Product

$$\mathbf{a} \cdot \mathbf{b} = \sum_{i=1}^{n} a_i b_i$$

But is it less efficient?

```
scProd :: Numeric a =>
          Vector (Data a) -> Vector (Data a) -> Data a
scProd a b = sum (zipWith (*) a b)
```

```
void test(struct array * v0, struct array * v1, float * out)
{   uint32_t len0;
    float v3;
    len0 = min(getLength(v0), getLength(v1));
    (* out) = 0.0f;
    for(uint32_t v2 = 0; v2 < len0; v2 += 1)
    {   v3 = ((* out) + (at(float,v0,v2) * at(float,v1,v2)));
        (* out) = v3;
    }
}
```

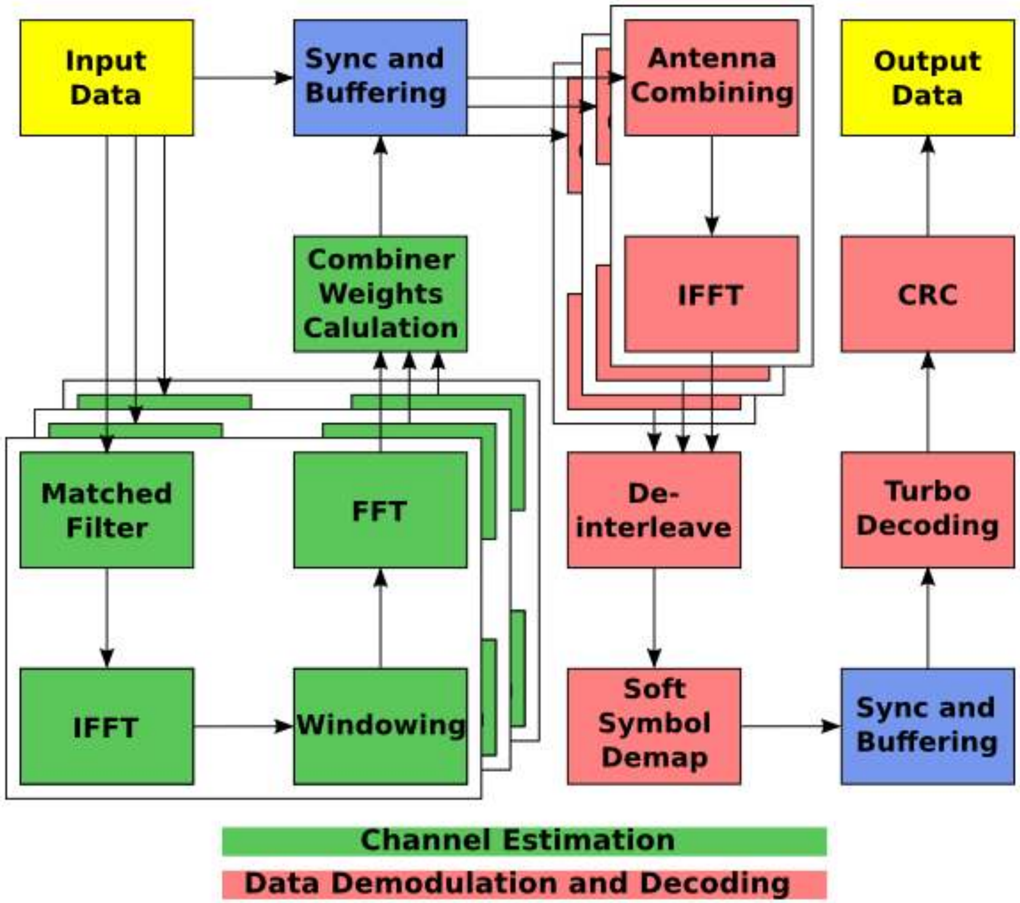
OLD

Use the Force

```
scProd2 :: Numeric a =>
      Vector (Data a) -> Vector (Data a) -> Data a
scProd2 a b = sum (force (zipWith (*) a b))
```

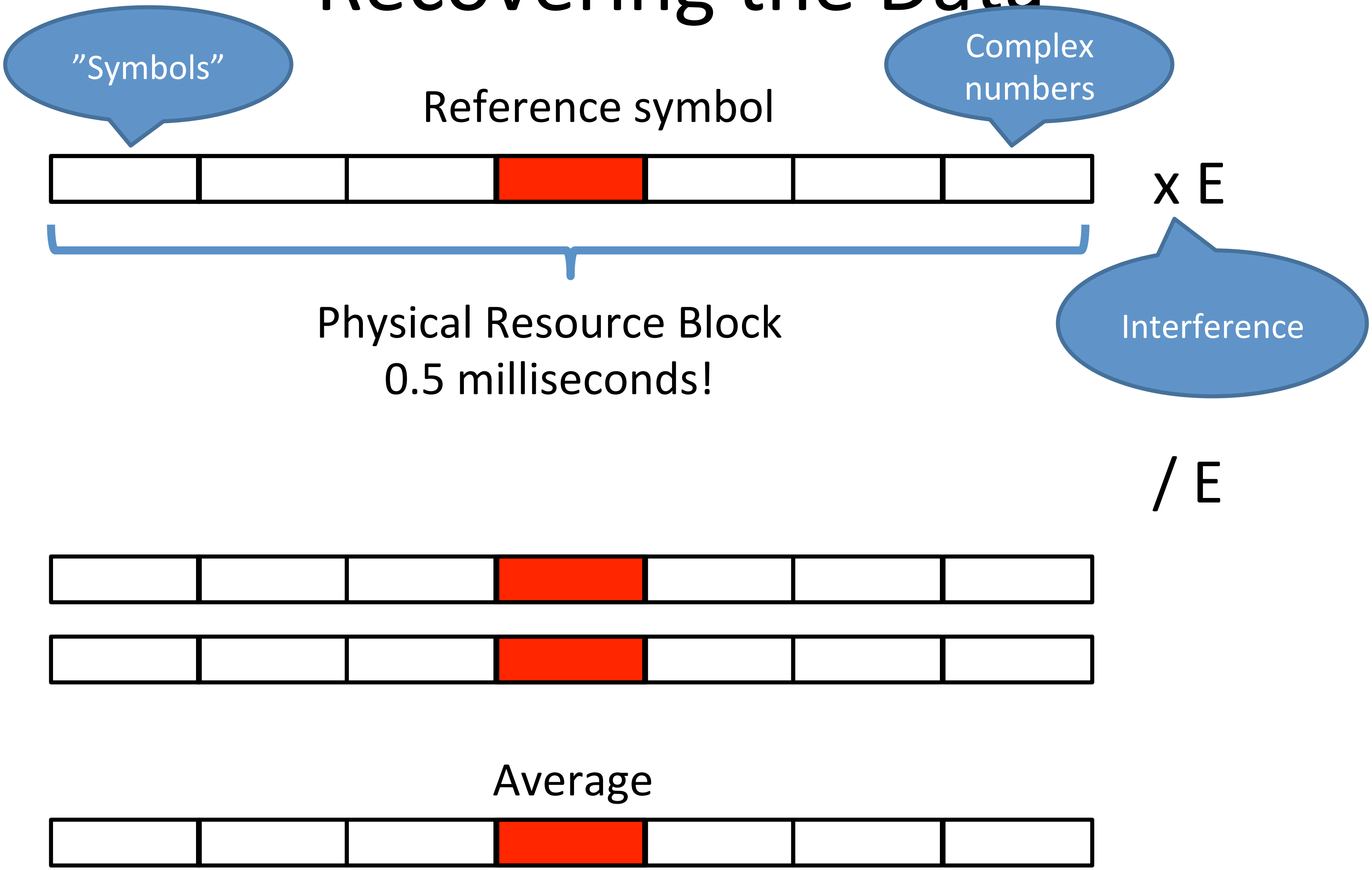
```
void test(struct array * v0, struct array * v1, float * out)
{   struct array v6 = {0}; float v4;
    initArray(&v6, sizeof(float), 100);
    for(uint32_t v5 = 0; v5 < 100; v5 += 1) {
        at(float,&v6,v5) = (at(float,v0,v5) * at(float,v1,v5));
    }
    (* out) = 0.0f;
    for(uint32_t v3 = 0; v3 < 100; v3 += 1) {
        v4 = ((* out) + at(float,&v6,v3));
        (* out) = v4;
    }
    freeArray(&v6);
}
```

LTE Uplink Receiver



- How a 4G base station figures out what your phone sent!

Recovering the Data



Combining Antennae in Feldspar

```
antennaComb chs input =
    map average -- Merging the symbols
  $ transpose   -- Swap dimensions
  $ zipWith (zipWith (*)) chs input
                -- Compensating for the channel

average :: Fraction a => Vector (Data a) -> Data a
average v = sum v / i2n (length v)
```

Fixing the sizes:

```
antennaCombFixed =
  antennaComb -::
    newSize2 4 1024 >-> newSize2 4 1024 >-> id
```

Fusion!

```
void test(struct array * v0, struct array * v1,
          struct array * out)
{ initArray(out, sizeof(float complex), 1024);
  for(uint32_t v2 = 0; v2 < 1024; v2 += 1) {
    float complex e0; float complex v4;
    e0 = (0.0f+0.0fi);
    for(uint32_t v3 = 0; v3 < 4; v3 += 1) {
      v4 =
        (e0 + (at(float complex,&at(struct array,v0,v3),v2)
            * at(float complex,&at(struct array,v1,v3),v2)));
      e0 = v4;
    }
    at(float complex,out,v2) = (e0 / (4.0f+0.0fi));
  }
}
```

- Just two nested loops!

Feldspar in a Nutshell

- Feldspar *restructures code* to eliminate intermediate data, fuse loops

- Can also *fuse parallelism* with sequential code
  - An easy way to explore alternative parallelisations

- http://github.com/feldspar

DSLs in Haskell

- Borrow parser, type-checker, module system… from Haskell
- Inherit Haskell’s expressive power
  - higher-order function, classes…
- Let the DSL designer focus on the cool, domain-specific stuff!

Haskell is Fun!

- **haskell.org**
  - The Haskell hub—where to download, online books & tutorials, you name it
- haskell-cafe@haskell.org
  - Community mailing list for all kinds of questions
- hackage.haskell.org
  - A bazillion libraries
- The Haskell Platform
  - Easy multi-platform download and installation of compiler and core libraries

Haskell Curry (1900–1982)

Currying

Every other programming language in the world

```
f :: (Integer,Integer) -> Integer
f(x,y) =  x*x + y*y

> f(3,4)
25
```

Every functional language

```
f :: Integer -> Integer -> Integer
f x y  =  x*x + y*y

> f 3 4
25
```

Currying

Every other programming language in the world

```
f :: (Integer,Integer) -> Integer
f(x,y) =  x*x + y*y

> f(3,4)
25
```

Every functional language

```
f :: Integer -> (Integer -> Integer)
(f x) y  =  x*x + y*y

> (f 3) 4
25
```

Currying

Every other programming language in the world

```
f :: (Integer,Integer) -> Integer
f(x,y) =  x*x + y*y

> f(3,4)
25
```

Every functional language

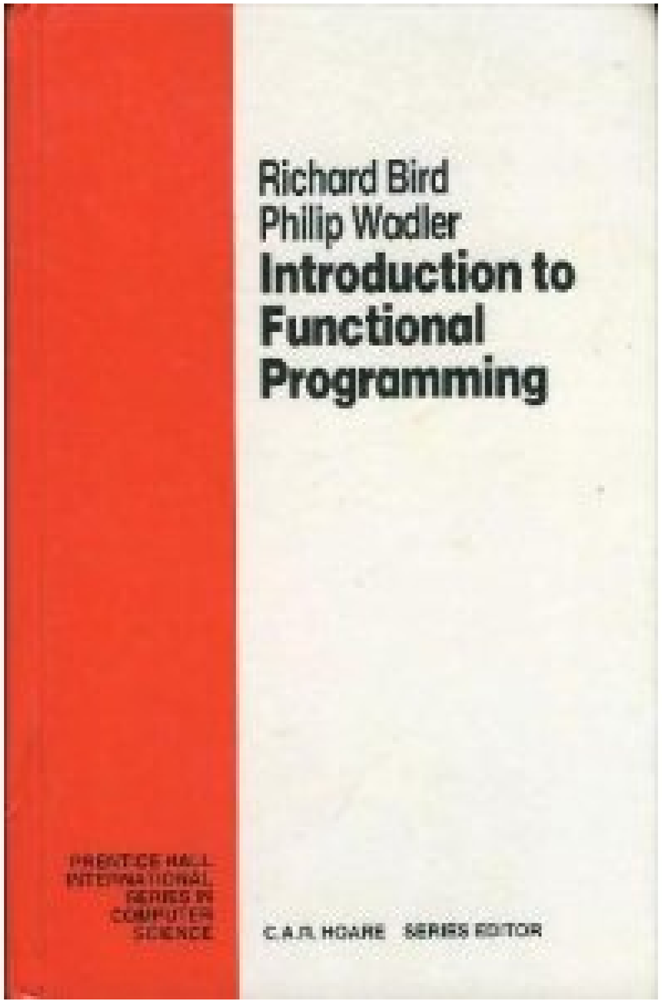
```
f :: Integer -> Integer -> Integer
f x y  =  x*x + y*y

> f 3 4
25
```


KEEP
CALM
AND
CURRY
ON

Haskell

Type Classes

Bird and Wadler (1988)



Polymorphism

- *Ad hoc* polymorphism
- Parametric polymorphism
- Subtype polymorphism

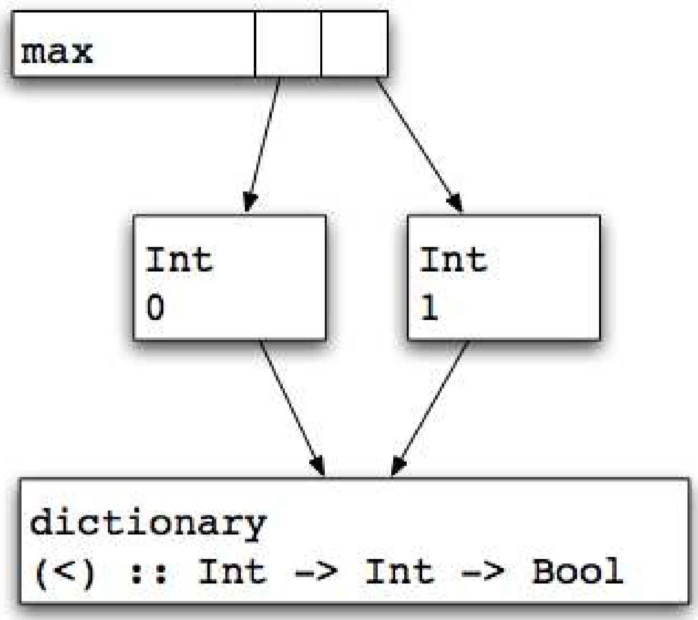
Type classes

```
class Ord a where
  (<) :: a -> a -> Bool

instance Ord Int where
  (<) = primitiveLessInt

instance Ord Char where
  (<) = primitiveLessChar

max  :: Ord a => a -> a -> a
max x y | x < y        =  y
        | otherwise    =  x

maximum :: Ord a => [a] -> a
maximum [x]       =  x
maximum (x:xs)    =  max x (maximum xs)

maximum [0,1,2] == 2
maximum "abc" == 'c'
```

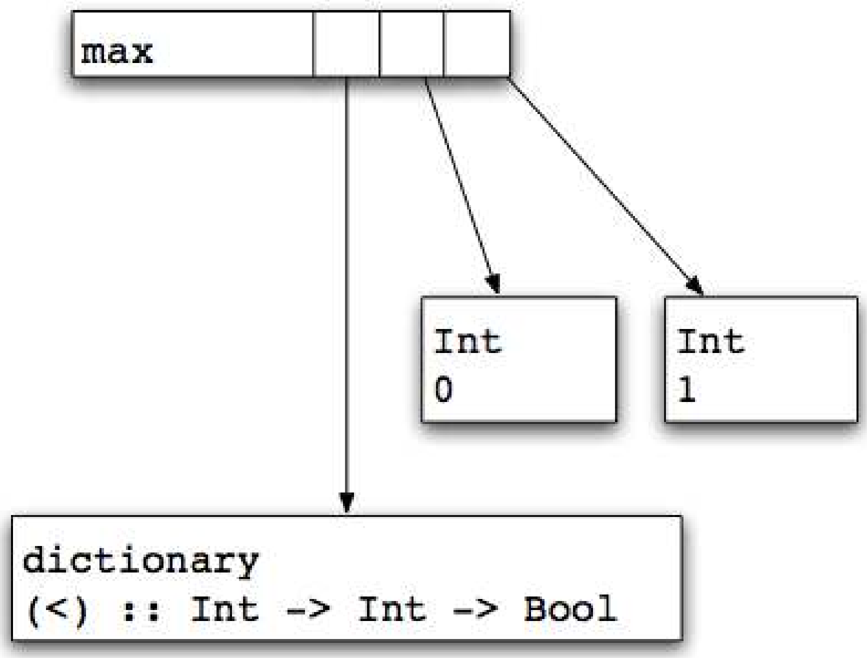
Translation

```
data Ord a = Ord { less :: a -> a -> Bool }

ordInt :: Ord Int
ordInt =  Ord { less = primitiveLessInt }

ordChar :: Ord Char
ordChar =  Ord { less = primitiveLessChar }

max  :: Ord a -> a -> a -> a
max d x y | less d x y   =  x
          | otherwise    =  y

maximum :: Ord a -> [a] -> a
maximum d [x]      =  x
maximum d (x:xs)   =  max d x (maximum d xs)

maximum ordInt [0,1,2] == 2
maximum ordChar "abc" == 'c'
```

Object-oriented



Type classes



Type classes, continued

```
instance Ord a => Ord [a] where
  [] < []                     =  False
  [] < y:ys                   =  True
  x:xs < []                   =  False
  x:xs < y:ys | x < y         =  True
              | y < x         =  False
              | otherwise     =  xs < ys

maximum ["zero","one","two"] == "zero"
maximum [[[0],[1]],[[0,1]]] == [[0,1]]
```

Translation, continued

```
ordList :: Ord a -> Ord [a]
ordList d  =  Ord { less = lt }
  where
  lt d [] []                              =  False
  lt d [] (y:ys)                          =  True
  lt d (x:xs) []                          =  False
  lt d (x:xs) (y:ys) | less d x y         =  True
                     | less d y x         =  False
                     | otherwise          =  lt d xs ys

maximum d0 ["zero","one","two"] == "zero"
maximum d1 [[[0],[1]],[[0,1]]] == [[0,1]]
  where
  d0 = ordList ordChar
  d1 = ordList (ordList ordInt)
```

Maximum of a list, in Java

```
public static <T extends Comparable<T>>
  T maximum(List<T> elts)
{
  T candidate = elts.get(0);
  for (T elt : elts) {
    if (candidate.compareTo(elt) < 0) candidate = elt;
  }
  return candidate;
}

List<Integer> ints = Arrays.asList(0,1,2);
assert maximum(ints) == 2;

List<String> strs = Arrays.asList("zero","one","two");
assert maximum(strs).equals("zero");

List<Number> nums = Arrays.asList(0,1,2,3.14);
assert maximum(nums) == 3.14;  // compile-time error
```

Naftalin and Wadler (2006)