A TASTE OF HASKELL

Simon Peyton Jones
Microsoft Research

A tutorial presented at the O'Reilly Open Source Convention, July 2007

Video of this tutorial (3 hrs)
http://research.microsoft.com/~simonpj/papers/haskell-tutorial

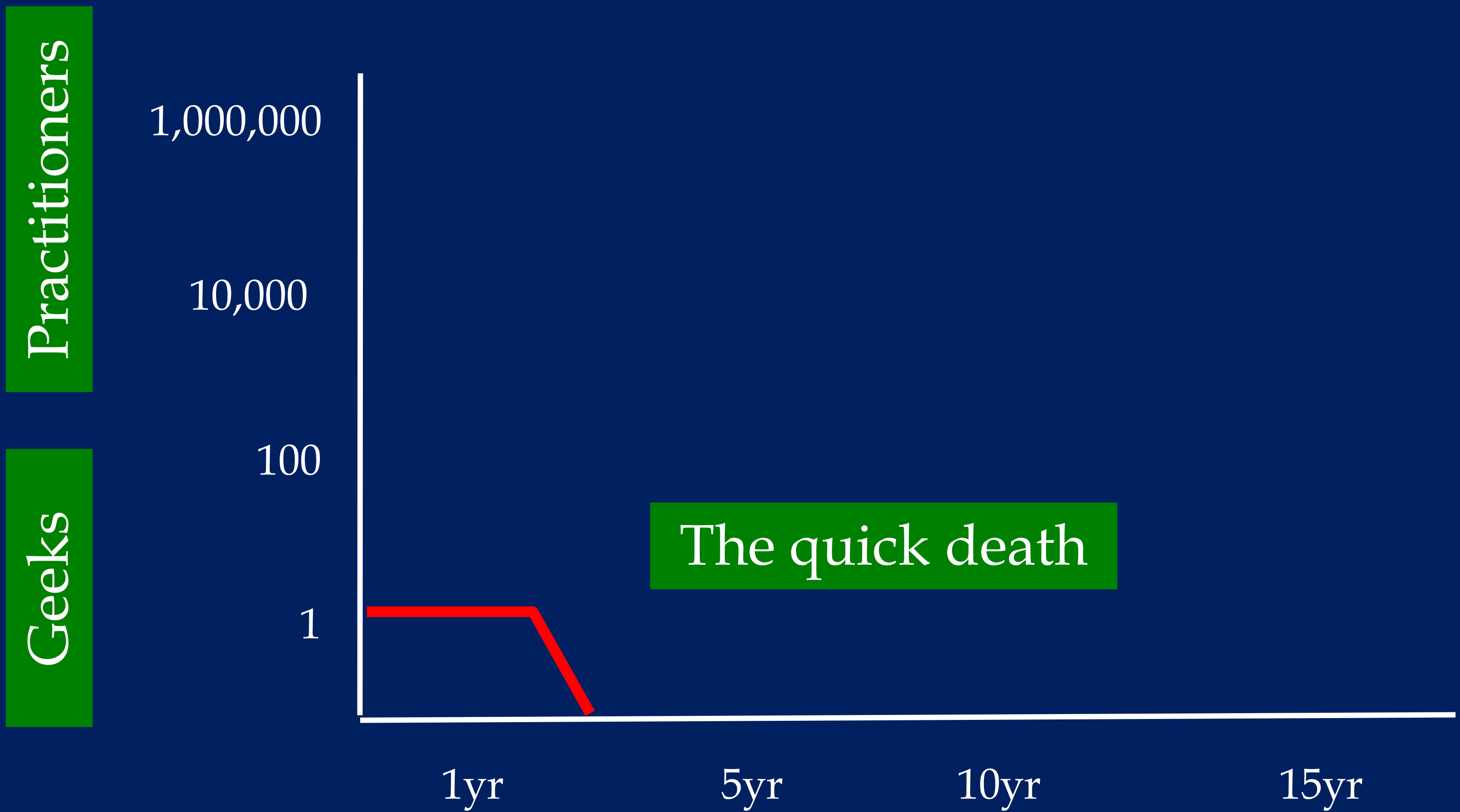
What is Haskell?

- Haskell is a programming language that is
  - purely functional
  - lazy
  - higher order
  - strongly typed
  - general purpose

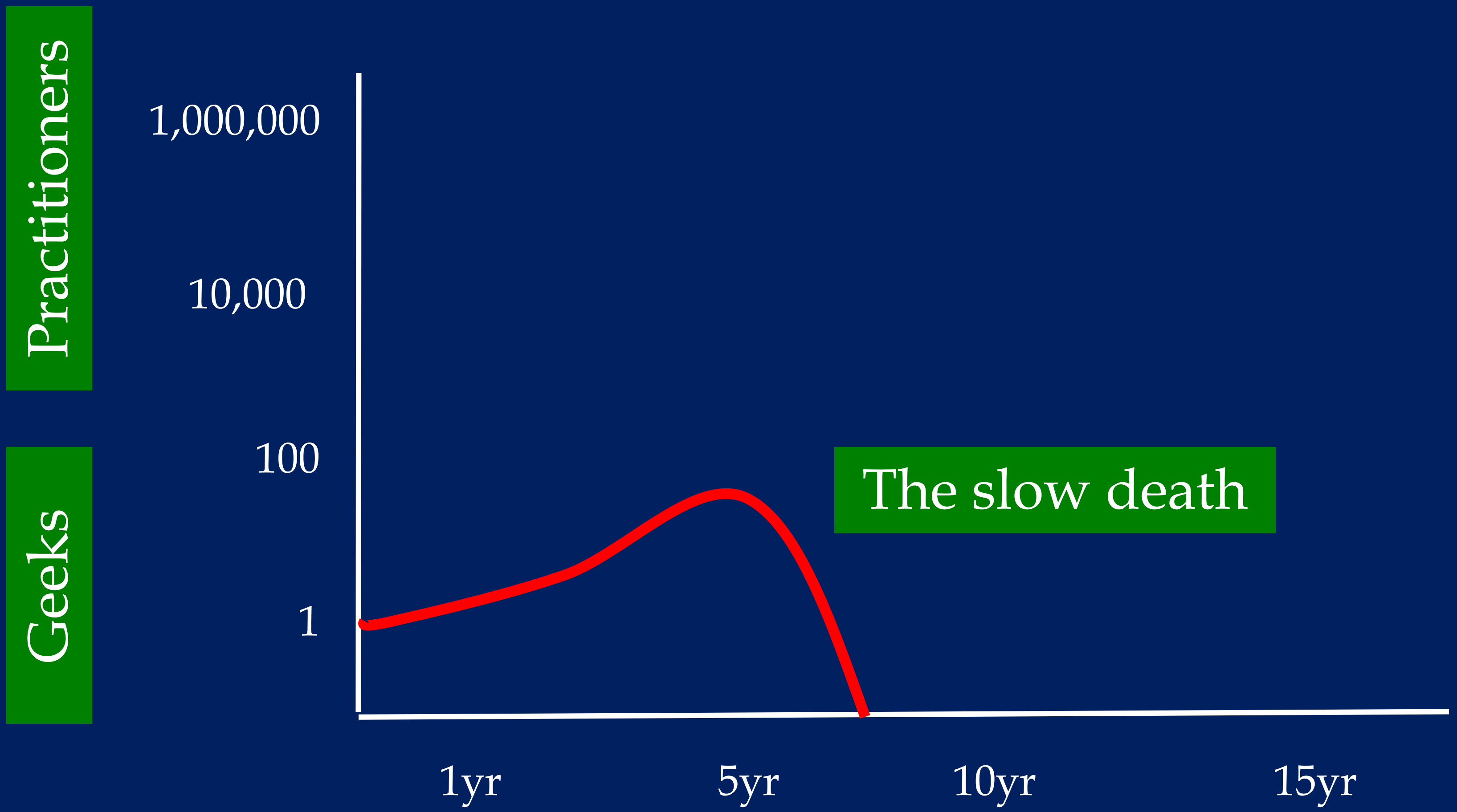
Why should I care?

- Functional programming will make you think differently about programming
  - Mainstream languages are all about **state**
  - Functional programming is all about **values**
- Whether or not you drink the Haskell Kool-Aid, you'll be a better programmer in whatever language you regularly use

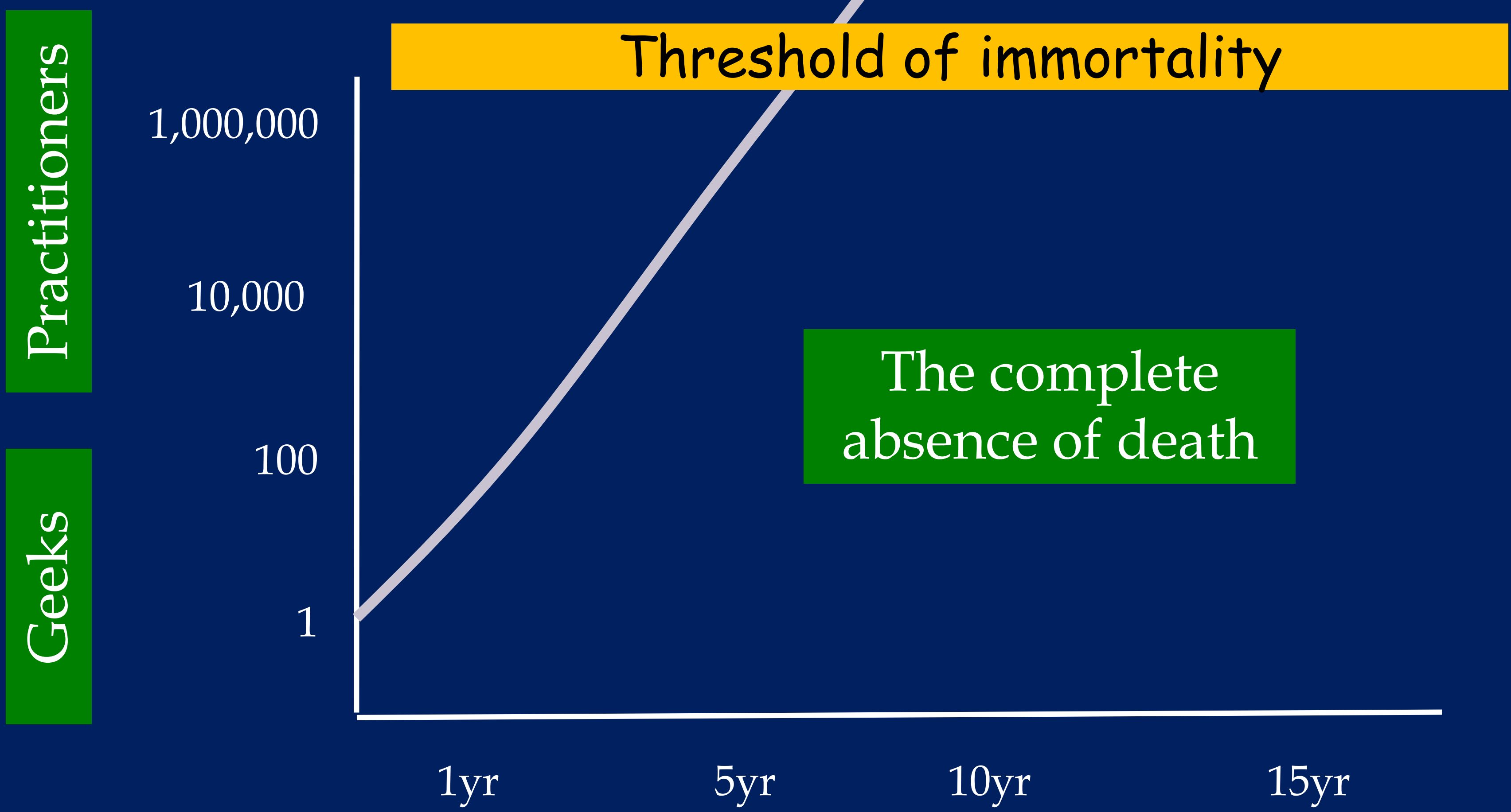
Most research languages



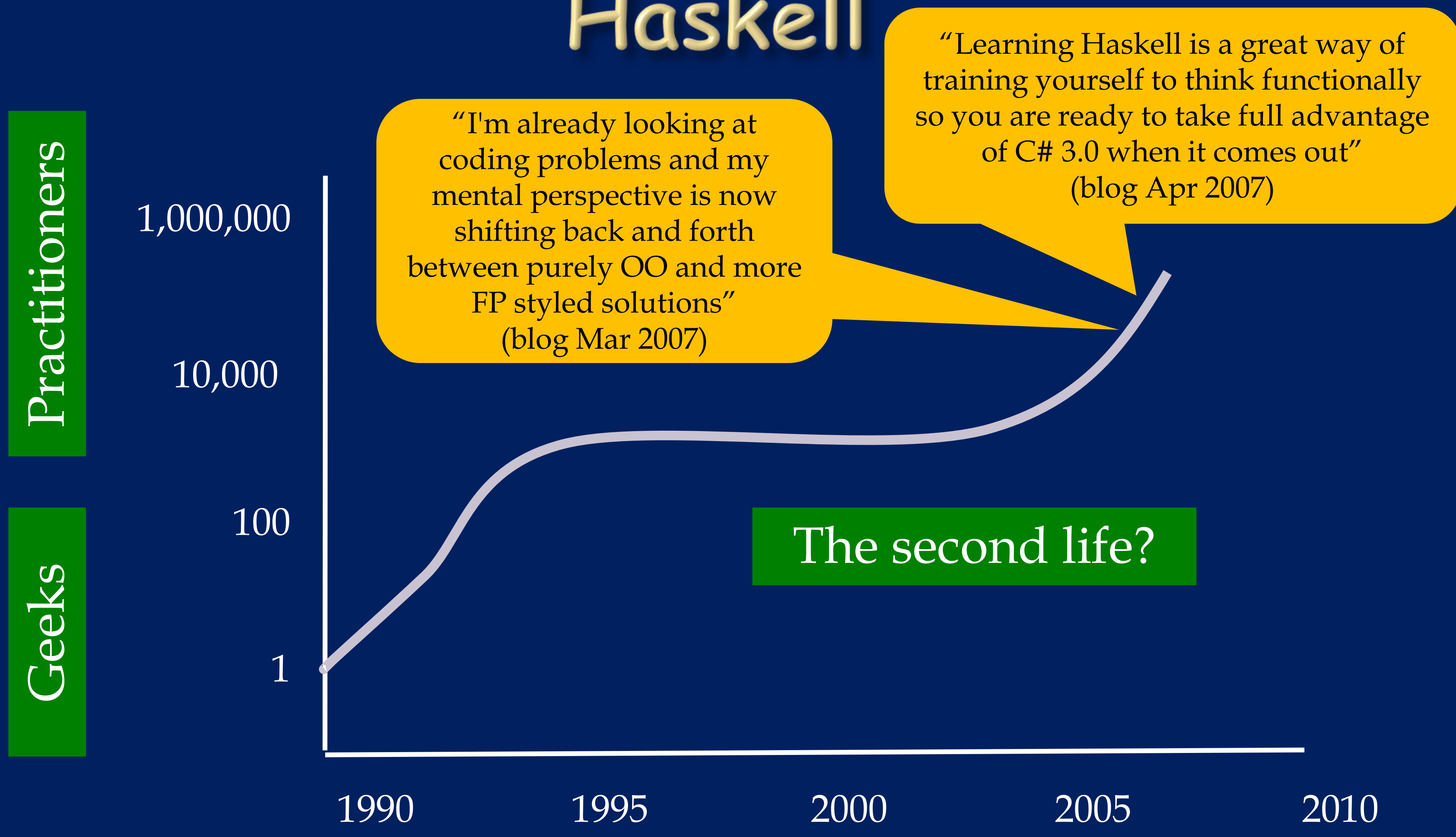
Successful research languages



C++, Java, Perl, Ruby



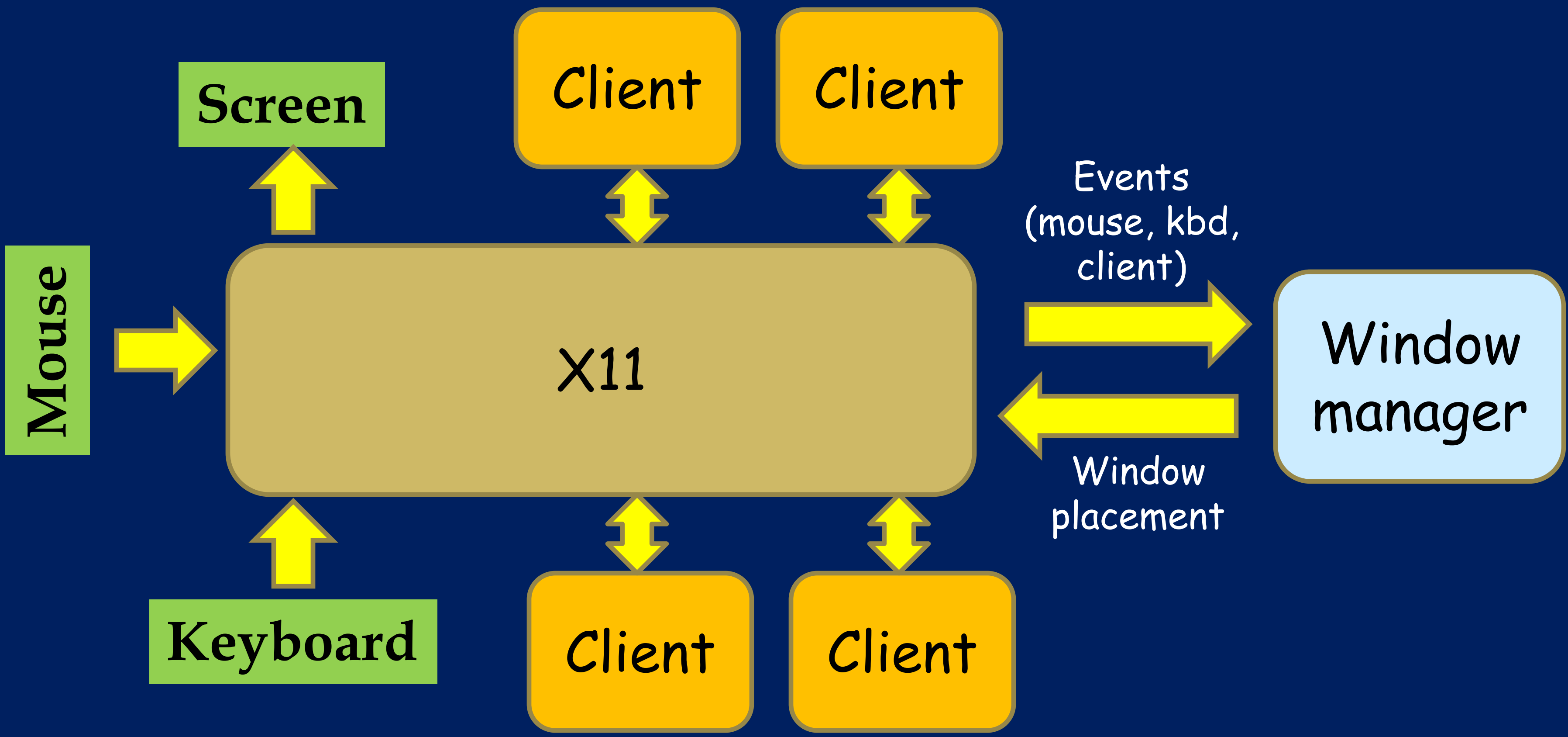
Haskell



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

xmonad

- xmonad is an X11 tiling window manager written entirely in Haskell



Why I'm using xmonad

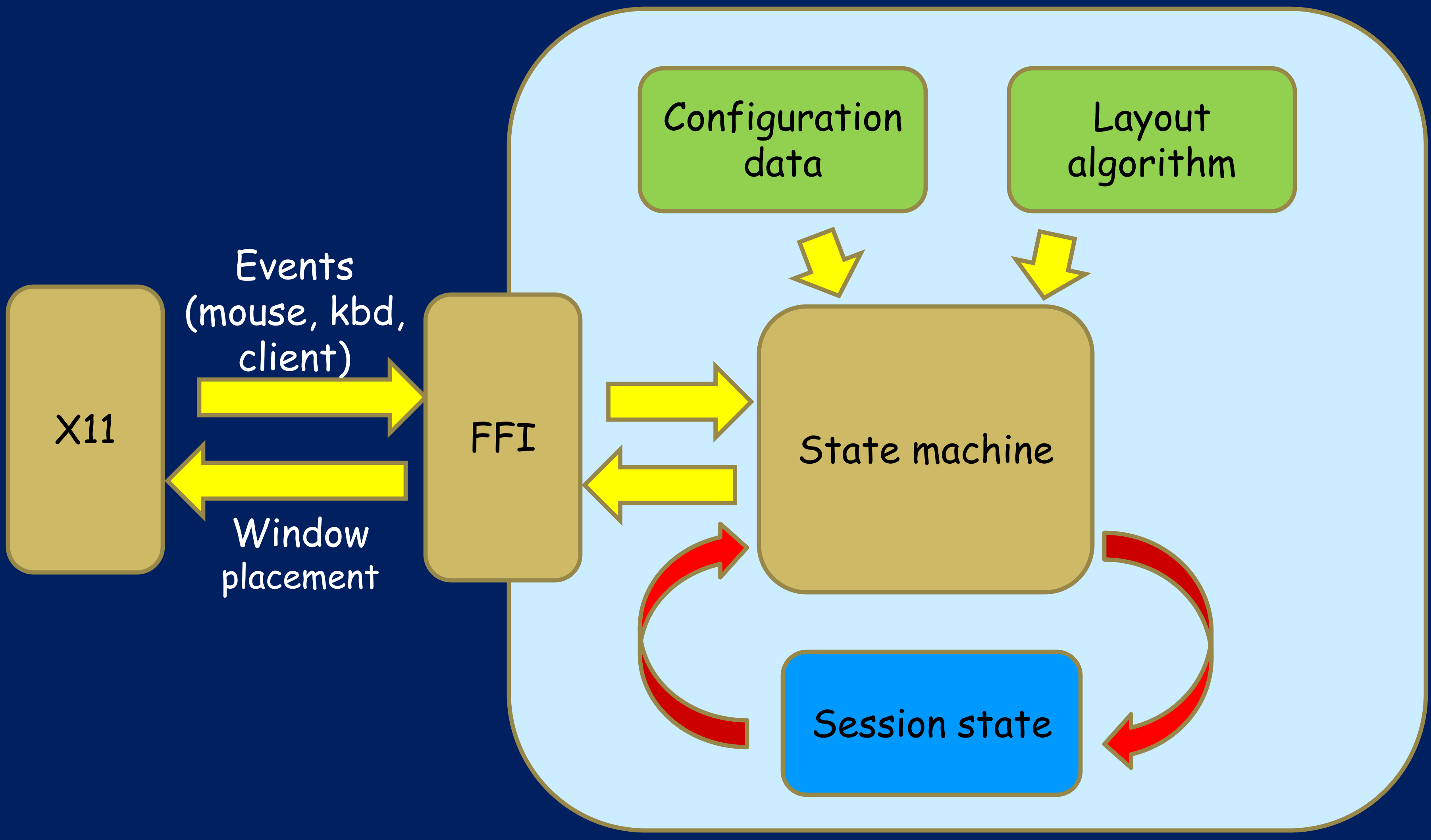
- Because it's
  - A real program
  - of manageable size
  - that illustrates many Haskell programming techniques
  - is open-source software
  - is being actively developed
  - by an active community

"Manageable size"

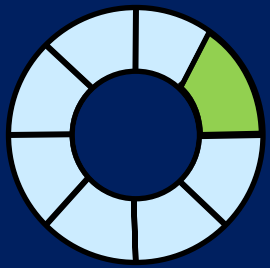
	Code	Comments	Language
metacity	>50k		C
ion3	20k	7k	C
larswm	6k	1.3k	C
wmii	6k	1k	C
dwm 4.2	1.5k	0.2k	C
xmonad 0.2	0.5k	0.7k	Haskell

Demo xmonad

Inside xmonad



The window stack



A ring of windows
One has the focus

Export list

```haskell
module Stack( Stack, insert, swap, ...) where

import Graphics.X11( Window )

type Stack = ...

insert :: Window -> Stack
-- Newly inserted window has focus
insert = ...

swap :: Stack -> Stack
-- Swap focus with next
swap = ...
```

Import things defined elsewhere

Define new types

Specify type of insert

Comments

The window stack

Stack should not exploit the fact that it's a stack of **windows**

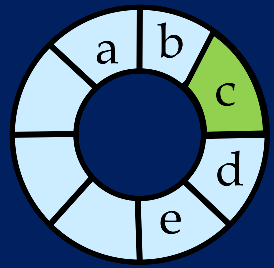
```
module Stack( Stack, insert, swap, ...) where



type Stack w = ...

insert :: w -> Stack w
-- Newly inserted window has focus
insert = ...

swap :: Stack w -> Stack w
-- Swap focus with next
swap = ...
```

No import any more

A stack of values of type w

Insert a 'w' into a stack of w's

The window stack



A ring of windows
One has the focus

A list takes one of two forms:

- **[]**, the empty list
- **(w:ws),** a list whose head is w, and tail is ws

The type "[w]" means "list of w"

```
type Stack w = [w]
-- Focus is first element of list,
-- rest follow clockwise

swap :: Stack w -> Stack w
-- Swap topmost pair
swap []                = []
swap (w  : [])         = w : []
swap (w1 : w2 : ws) = w2 : w1 : ws
```

The ring above is represented [c,d,e,...,a,b]

Functions are defined by pattern matching

w1:w2:ws means w1 : (w2 : ws)

Syntactic sugar

```
swap []            = []
swap (w:[])        = w:[]
swap (w1:w2:ws)    = w2:w1:ws
```

```
swap []            = []
swap [w]           = [w]
swap (w1:w2:ws)    = w2:w1:ws
```

[a,b,c] means a:b:c:[]

```
swap (w1:w2:ws) = w2:w1:ws
swap ws         = ws
```

Equations are matched top-to-bottom

```
swap ws = case ws of
               []          -> []
               [w]         -> [w]
               (w1:w2:ws)  -> w2:w1:ws
```

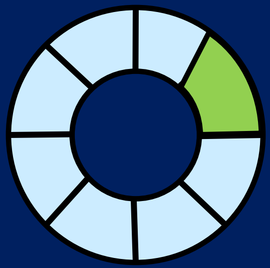
case expressions

Running Haskell

- Download:
  - ghc: http://haskell.org/ghc
  - Hugs: http://haskell.org/hugs
- Interactive:
  - ghci Stack.hs
  - hugs Stack.hs
- Compiled:
  - ghc –c Stack.hs

Demo ghci

Rotating the windows



A ring of windows
One has the focus

```
focusNext :: Stack -> Stack
focusNext (w:ws) = ws ++ [w]
focusnext []     = []
```

Pattern matching forces us to think of all cases

Type says "this function takes two arguments, of type [a], and returns a result of type [a]"

```
(++) :: [a] -> [a] -> [a]
-- List append; e.g. [1,2] ++ [4,5] = [1,2,4,5]
```

Definition in Prelude (implicitly imported)

Recursion

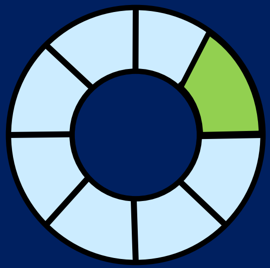
Recursive call

```
(++) :: [a] -> [a] -> [a]
-- List append; e.g. [1,2] ++ [4,5] = [1,2,4,5]

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

Execution model is simple rewriting:

```
[1,2] ++ [4,5]
= (1:2:[]) ++ (4:5:[])
= 1 : ((2:[]) ++ (4:5:[]))
= 1 : 2 : ([] ++ (4:5:[]))
= 1 : 2 : 4 : 5 : []
```

Rotating backwards



A ring of windows
One has the focus

```
focusPrev :: Stack -> Stack
focusPrev ws = reverse (focusNext (reverse ws))
```

Function application by mere juxtaposition

```
reverse :: [a] -> [a]
-- e.g. reverse [1,2,3] = [3,2,1]
reverse []     = []
reverse (x:xs) = reverse xs ++ [x]
```

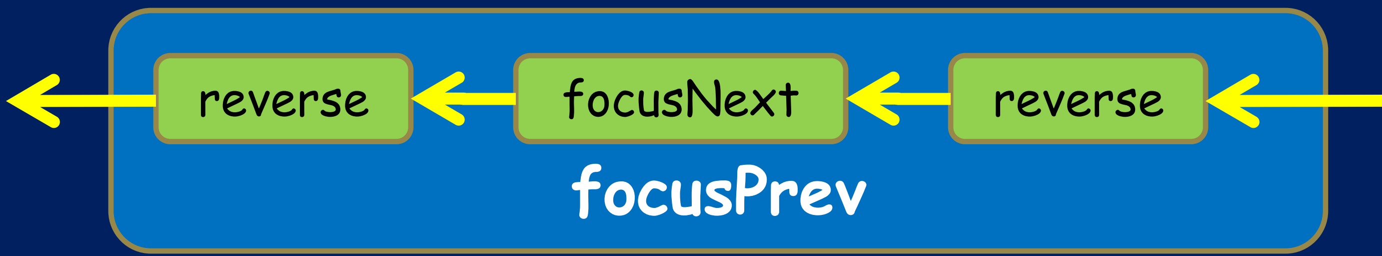
Function application
binds more tightly than anything else:
(reverse xs) ++ [x]

Function composition

```
focusPrev :: Stack -> Stack
focusPrev ws = reverse (focusNext (reverse ws))
```

can also be written

```
focusPrev :: Stack -> Stack
focusPrev = reverse . focusNext . reverse
```



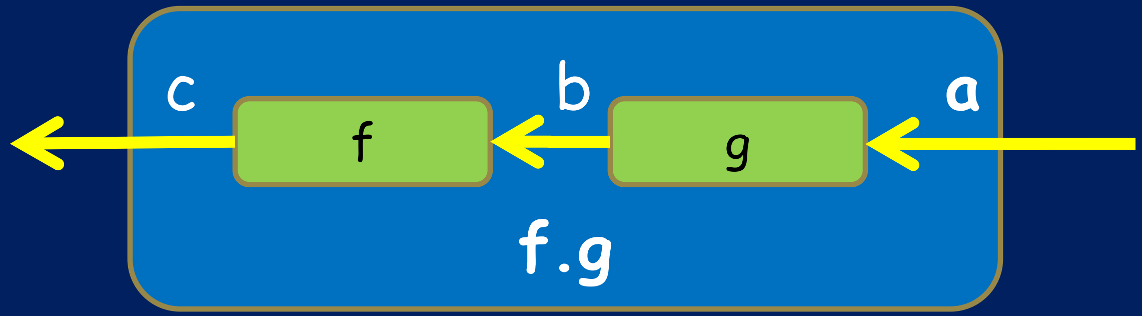
Definition of (.) from Prelude

```
(f . g) x = f (g x)
```

Function composition

Functions as arguments

```
(.) :: (b->c) -> (a->b) -> (a->c)
(f . g) x = f (g x)
```



Just testing

Just testing

- It's good to write tests as you write code
- E.g. focusPrev undoes focusNext; swap undoes itself; etc

```
module Stack where

...definitions...

-- Write properties in Haskell
type TS = Stack Int -- Test at this type

prop_focusNP :: TS -> Bool
prop_focusNP s = focusNext (focusPrev s) == s

prop_swap :: TS -> Bool
prop_swap s = swap (swap s) == s
```

Test interactively

Test.QuickCheck is simply a Haskell library (not a "tool")

```
bash$ ghci Stack.hs
Prelude> :m +Test.QuickCheck

Prelude Test.QuickCheck> quickCheck prop_swap
+++ OK, passed 100 tests

Prelude Test.QuickCheck> quickCheck prop_focusNP
+++ OK, passed 100 tests
```

...with a strange-looking type

```
Prelude Test.QuickCheck> :t quickCheck
quickCheck :: Testable prop => prop -> IO ()
```

Demo QuickCheck

Test batch-mode

runHaskell Foo.hs <args>
runs Foo.hs, passing it <args>

A 25-line Haskell script

Look for "prop_" tests in here

```
bash$ runhaskell QC.hs Stack.hs
prop_swap: +++ OK, passed 100 tests
prop_focusNP: +++ OK, passed 100 tests
```

Things to notice

Things to notice...

No side effects. At all.

```
swap :: Stack w -> Stack w
```

- A call to swap returns a new stack; the old one is unaffected.

```
prop_swap s = swap (swap s) == s
```

- A variable 's' stands for an immutable **value**, not for a **location** whose value can change with time. Think spreadsheets!

Things to notice...

Purity makes the interface explicit

```
swap :: Stack w -> Stack w      -- Haskell
```

- Takes a stack, and returns a stack; that's all

```
void swap( stack s )            /* C */
```

- Takes a stack; may modify it; may modify other persistent state; may do I/O

Things to notice...

Pure functions are easy to test

```
prop_swap s = swap (swap s) == s
```

- In an imperative or OO language, you have to
  - 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

Things to notice...

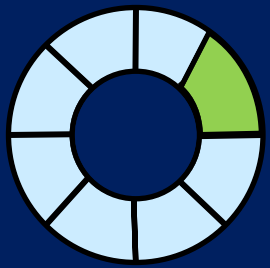
Types are everywhere

```
swap :: Stack w -> Stack w
```

- Usual static-typing rant omitted...
- In Haskell, types express high-level design, in the same way that UML diagrams do; with the advantage that the type signatures are machine-checked
- Types are (almost always) optional: type inference fills them in if you leave them out

Improving the design

Improving the design



A ring of windows
One has the focus

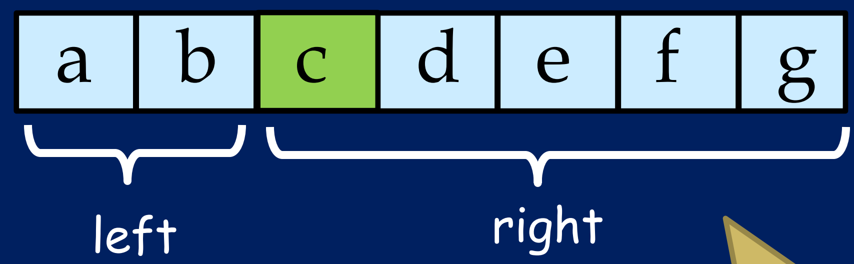
```
type Stack w = [w]
-- Focus is head of list

enumerate:: Stack w -> [w]
-- Enumerate the windows in layout order
enumerate s = s
```

- Changing focus moves the windows around: confusing!

Improving the design

A **sequence** of windows
One has the focus

- Want: a fixed layout, still with one window having focus



Data type declaration

Constructor of the type

Represented as
MkStk [b,a] [c,d,e,f,g]

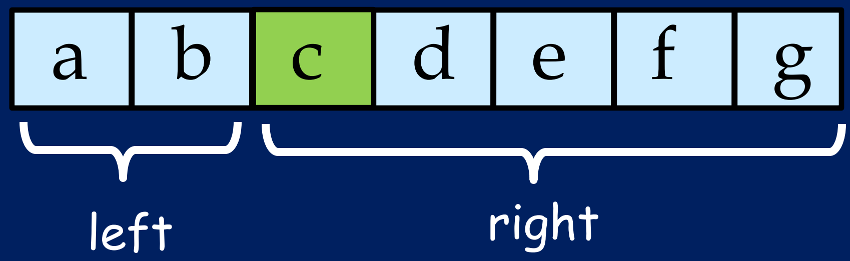
```
data Stack w = MkStk [w] [w]  -- left and right resp
-- Focus is head of 'right' list
-- Left list is *reversed*
-- INVARIANT: if 'right' is empty, so is 'left'
```

Improving the design

A **sequence** of windows
One has the focus

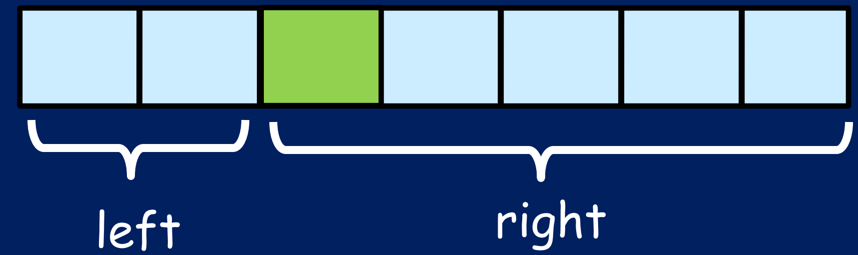
- Want: a fixed layout, still with one window having focus

Represented as
MkStk [b,a] [c,d,e,f,g]



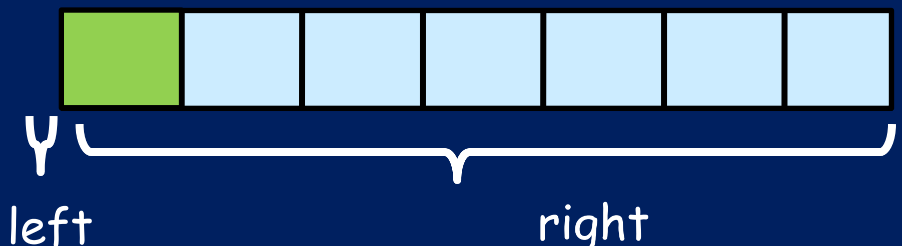

```
data Stack w = MkStk [w] [w]  -- left and right resp
-- Focus is head of 'right' list
-- Left list is *reversed*
-- INVARIANT: if 'right' is empty, so is 'left'

enumerate :: Stack w -> [w]
enumerate (MkStack ls rs) = reverse ls ++ rs
```

Moving focus




```
data Stack w = MkStk [w] [w]  -- left and right resp

focusPrev :: Stack w -> Stack w
focusPrev (MkStk (l:ls) rs) = MkStk ls (l:rs)
focusPrev (MkStk []     rs) = ...???...
```

Nested pattern matching

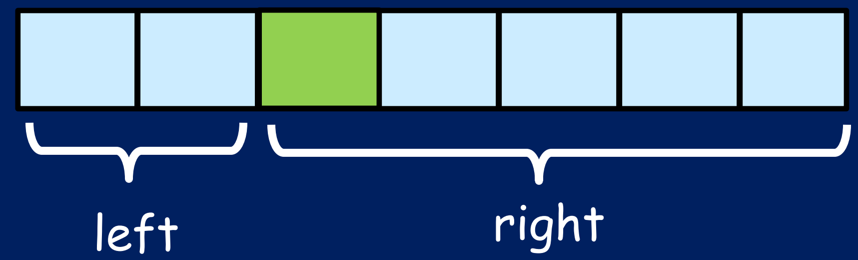


Choices for left=[]:

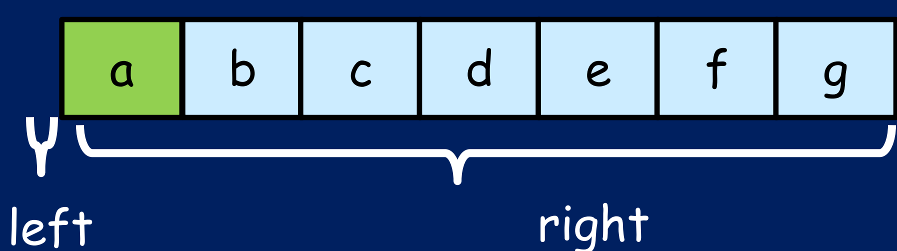
- no-op
- move focus to end

We choose this one

Moving focus

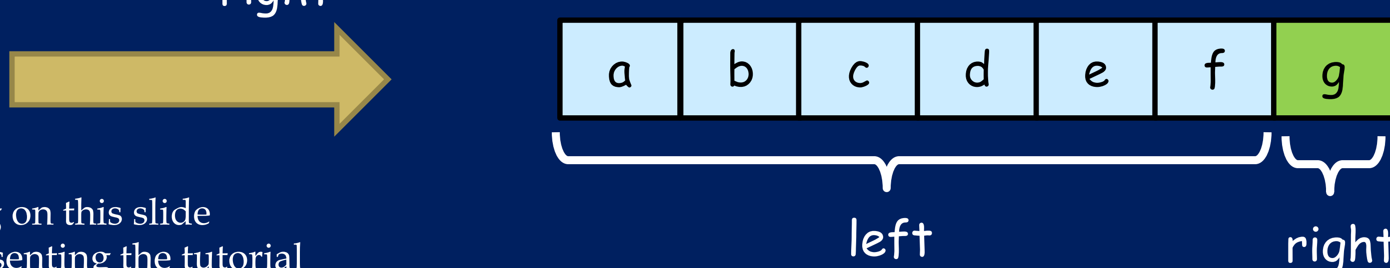



```
data Stack w = MkStk [w] [w]  -- left and right resp
-- Focus is head of 'right'

focusPrev :: Stack w -> Stack w
focusPrev (MkStk (l:ls) rs) = MkStk ls (l:rs)
focusPrev (MkStk []     rs) = case (reverse rs) of
                                (l:ls) -> MkStk ls [l]
```



Choices:

- no-op
- move focus to end



Note: I fixed a bug on this slide subsequent to presenting the tutorial

Oops..

```
Warning: Pattern match(es) are non-exhaustive
   In the case expression: ...
   Patterns not matched: []
```

```
data Stack w = MkStk [w] [w]  -- left and right resp
-- Focus is head of ‘right’

focusPrev :: Stack w -> Stack w
focusPrev (MkStk (l:ls) rs) = MkStk ls (l:rs)
focusPrev (MkStk []     rs) = case (reverse rs) of
                                (l:ls) -> MkStk ls [l]
                                []     -> MkStk [] []
```

- Pattern matching forces us to confront all the cases
- Efficiency note: reverse costs O(n), but that only happens once every n calls to focusPrev, so amortised cost is O(1).

Data types

- A new **data type** has one or more constructors
- Each **constructor** has zero or more arguments

```
data Stack w = MkStk [w] [w]

data Bool = False | True

data Colour = Red | Green | Blue

data Maybe a = Nothing | Just a
```

Built-in syntactic sugar for lists, but otherwise lists are just another data type

```
data [a] = []
         | a : [a]
```

Data types

```
data Stack w = MkStk [w] [w]

data Bool = False | True

data Colour = Red | Green | Blue

data Maybe a = Nothing | Just a
```

- Constructors are used:
  - as a function to construct values ("right hand side")
  - in patterns to deconstruct values ("left hand side")

```
isRed :: Colour -> Bool
isRed Red   = True
isRed Green = False
isRed Blue  = False
```

Patterns

Values

Data types

```
data Maybe a = Nothing | Just a

data Stack w = MkStk [w] [w]
-- Invariant for (MkStk ls rs)
--      rs is empty => ls is empty
```

- Data types are used
  - to describe data (obviously)
  - to describe "outcomes" or "control"

```
module Stack( focus, ... ) where

  focus :: Stack w -> Maybe w
  -- Returns the focused window of the stack
  -- or Nothing if the stack is empty
  focus (MkStk _ [])     = Nothing
  focus (MkStk _ (w:_)) = Just w
```

A bit like an exception...

...but you can't forget to catch it
No "null-pointer dereference" exceptions

```
module Foo where
import Stack

foo s = ...case (focus s) of
              Nothing -> ...do this in empty case...
              Just w  -> ...do this when there is a focus...
```

Data type abstraction

```
module Operations( ... ) where

import Stack( Stack, focusNext )

f :: Stack w -> Stack w
f (MkStk as bs) = ...
```

OK: `Stack` is imported

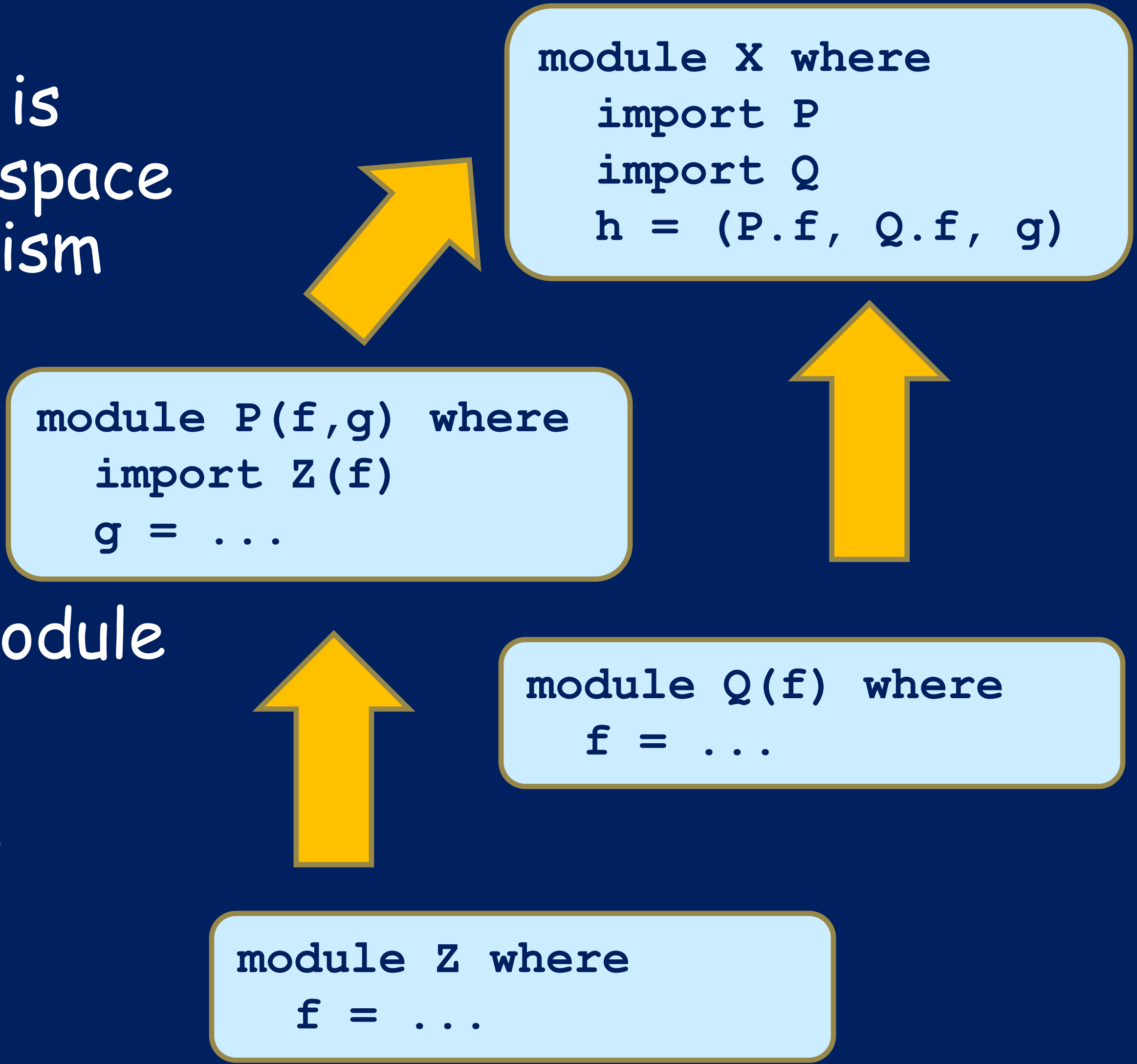
NOT OK: `MkStk` is not imported

```
module Stack( Stack, focusNext, focusPrev, ... ) where

data Stack w = MkStk [w] [w]

focusNext :: Stack w -> Stack w
focusNext (MkStk ls rs) = ...
```

`Stack` is exported,
but not its constructors;
so its representation is hidden

Haskell's module system

- Module system is merely a name-space control mechanism
- Compiler typically does lots of cross-module inlining
- Modules can be grouped into **packages**



Type classes

The need for type classes

```
delete :: Stack w -> w -> Stack w
-- Remove a window from the stack
```

- Can this work for ANY type w?

```
delete :: ∀w. Stack w -> w -> Stack w
```

- No – only for w's that support **equality**

```
sort :: [a] -> [a]
-- Sort the list
```

- Can this work for ANY type a?
- No – only for a's that support **ordering**

The need for type classes

```
serialise :: a -> String
-- Serialise a value into a string
```

- Only for w's that support **serialisation**

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

- Only for numbers that support **multiplication**
- But square should work for any number that does; e.g. Int, Integer, Float, Double, Rational

Type classes

"for all types w that support the Eq operations"

```
delete :: ∀w. Eq w => Stack w -> w -> Stack w
```

- If a function works for every type that has particular properties, the type of the function says just that

```
sort      :: Ord a  => [a] -> [a]
serialise :: Show a => a -> String
square    :: Num n  => n -> n
```

- Otherwise, it must work for any type whatsoever

```
reverse :: [a] -> [a]
filter  :: (a -> Bool) -> [a] -> [a]
```

Type classes

Works for any type 'n' that supports the Num operations

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

FORGET all you know about OO classes!

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

The class declaration says what the Num operations are

```
instance Num Int where
  a + b    = plusInt a b
  a * b    = mulInt a b
  negate a = negInt a
  ...etc..
```

An instance declaration for a type T says how the Num operations are implemented on T's

```
plusInt :: Int -> Int -> Int
mulInt  :: Int -> Int -> Int
etc, defined as primitives
```

How type classes work

When you write this...

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

...the compiler generates this

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

The "`Num n =>`" turns into an extra **value argument** to the function.
It is a value of data type `Num n`

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

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

How type classes work

When you write this...

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

...the compiler generates this

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

```
instance Num Int where
  a + b    = plusInt a b
  a * b    = mulInt a b
  negate a = negInt a
  ...etc..
```

```
dNumInt :: Num Int
dNumInt = MkNum plusInt
                mulInt
                negInt
                ...
```

An instance decl for type T translates to a value declaration for the Num dictionary for T

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

All this scales up nicely

- You can build big overloaded functions by calling smaller overloaded functions

```
sumSq :: Num n => n -> n -> n
sumSq x y = square x + square y
```

```
sumSq :: Num n -> n -> n -> n
sumSq d x y = (+) d (square d x)
                    (square d y)
```

Extract addition operation from d

Pass on d to square

All this scales up nicely

- You can build big instances by building on smaller instances

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

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

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

dEqList :: Eq a -> Eq [a]
dEqList d = MkEq eql
  where
    eql []     []     = True
    eql (x:xs) (y:ys) = (==) d x y && eql xs ys
    eql _      _      = False
```

Example: complex numbers

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

inc :: Num a => a -> a
inc x = x + 1
```

Even literals are overloaded

"1" means "fromInteger 1"

```
data Cpx a = Cpx a a

instance Num a => Num (Cpx a) where
  (Cpx r1 i1) + (Cpx r2 i2) = Cpx (r1+r2) (i1+i2)
  fromInteger n = Cpx (fromInteger n) 0
```

A completely different example: Quickcheck

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

class Testable a where
  test :: a -> RandSupply -> Bool

class Arbitrary a where
  arby :: RandSupply -> a

instance Testable Bool where
  test b r = b

instance (Arbitrary a, Testable b)
      => Testable (a->b) where
  test f r = test (f (arby r1)) r2
           where (r1,r2) = split r
```

```
split :: RandSupply -> (RandSupply, RandSupply)
```

A completely different example: Quickcheck

```
prop_swap :: TS -> Bool
```

```
test prop_swap r

= test (prop_swap (arby r1)) r2
where (r1,r2) = split r

= prop_swap (arby r1)
```

Using instance for (->)

Using instance for Bool

A completely different example: Quickcheck

```
class Arbitrary a where
  arby :: RandSupply -> a

instance Arbitrary Int where
  arby r = randInt r

instance Arbitrary a
      => Arbitrary [a] where
  arby r | even r1 = []
         | otherwise = arby r2 : arby r3
    where
      (r1,r') = split r
      (r2,r3) = split r'
```

Generate Nil value

Generate cons value

```
split :: RandSupply -> (RandSupply, RandSupply)
randInt :: RandSupply -> Int
```

A completely different example: Quickcheck

- QuickCheck uses type classes to auto-generate
  - random values
  - testing functions

  based on the type of the function under test
- Nothing is built into Haskell; QuickCheck is just a library
- Plenty of wrinkles, esp
  - test data should satisfy preconditions
  - generating test data in sparse domains

Type classes = OOP?

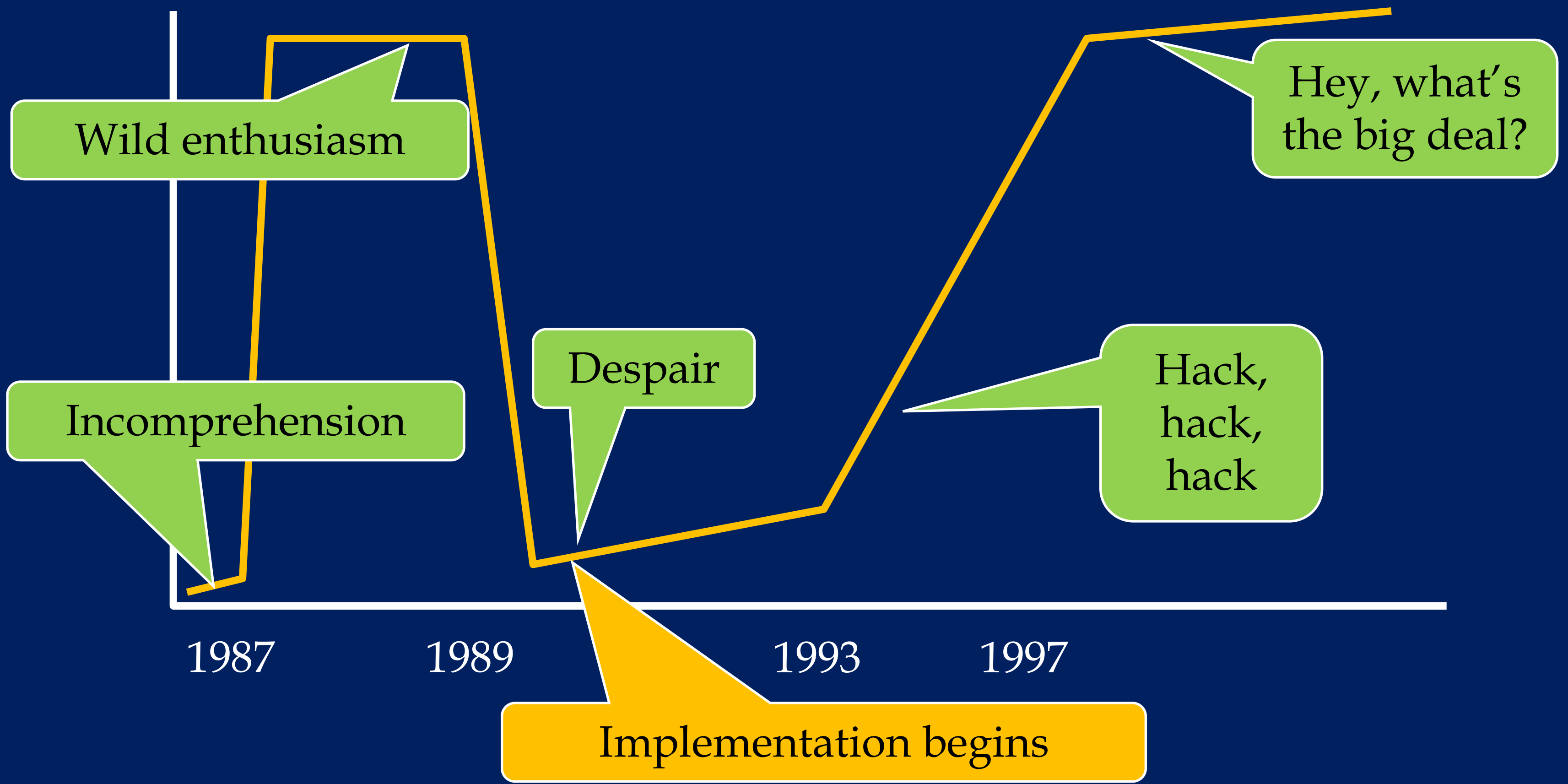
- In OOP, a value carries a method suite
- With type classes, the method suite travels separately from the value
  - Old types can be made instances of new type classes (e.g. introduce new Serialise class, make existing types an instance of it)
  - Method suite can depend on **result** type
    e.g. fromInteger :: Num a => Integer -> a
  - Polymorphism, not subtyping

Type classes have proved extraordinarily convenient in practice

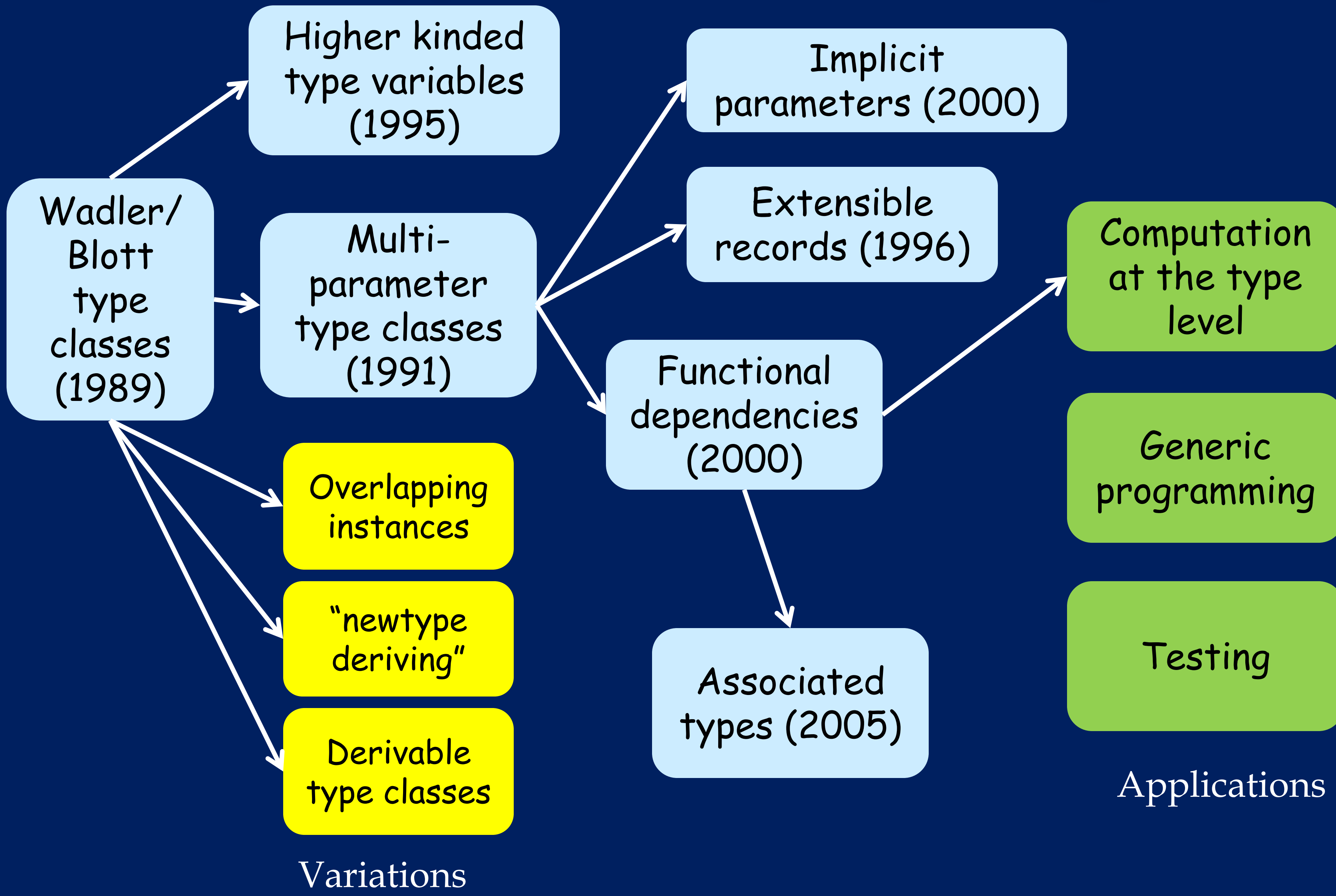
- Equality, ordering, serialisation
- Numerical operations. Even numeric constants are overloaded; e.g. f x = x*2
- And on and on....time-varying values, pretty-printing, collections, reflection, generic programming, marshalling, monads, monad transformers....

Type classes over time

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



Type-class fertility



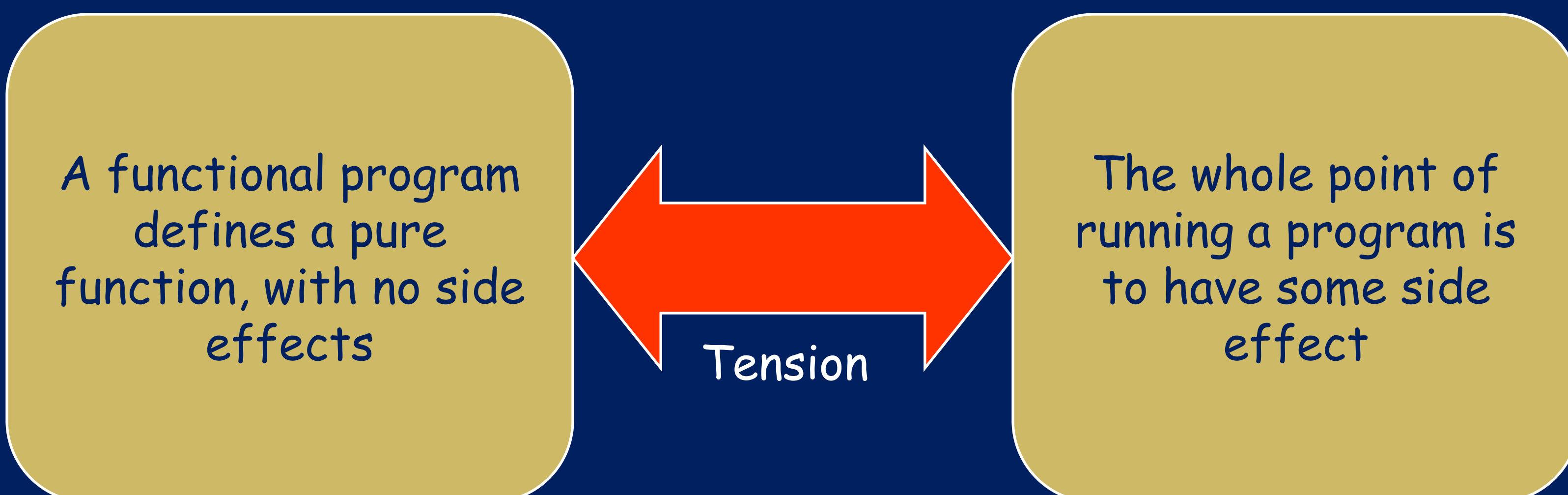
Type classes summary

- 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
- Long term impact yet to become clear

Doing I/O

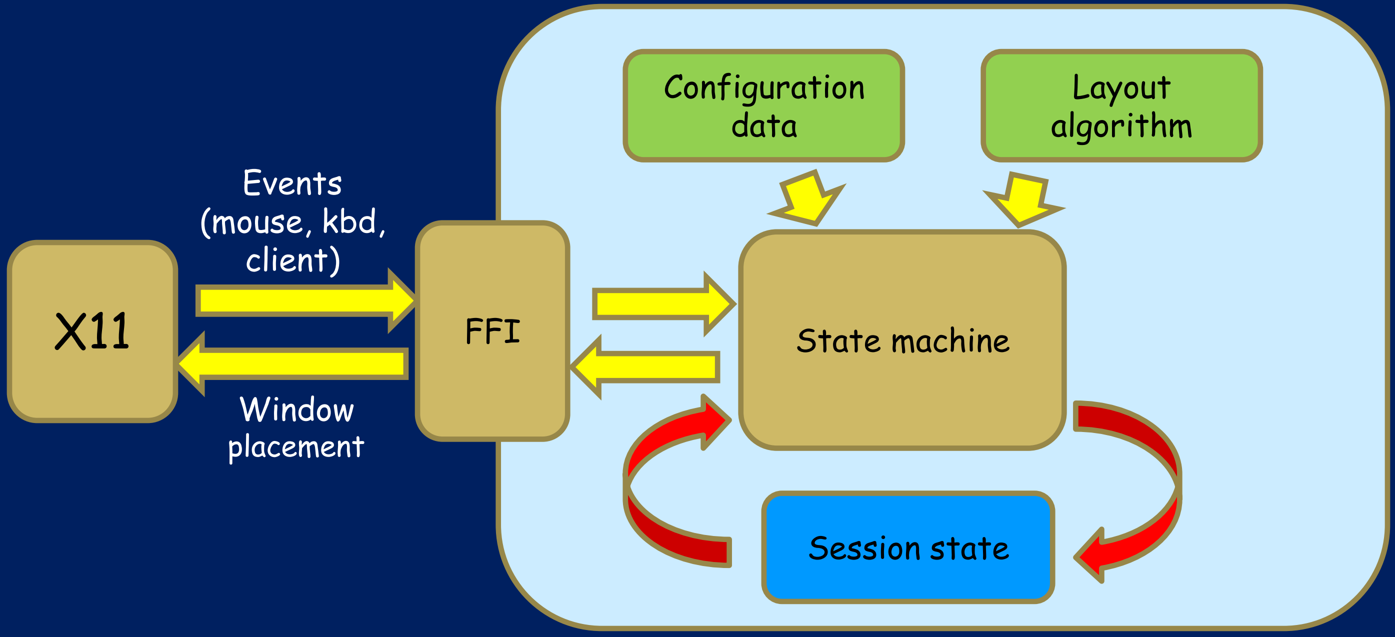
Where is the I/O in xmonad?

- All this pure stuff is very well, but sooner or later we have to
  - talk to X11, whose interface is not at all pure
  - do input/output (other programs)



Where is the I/O in xmonad?

- All this pure stuff is very well, but sooner or later we have to
  - talk to X11, whose interface is not at all pure
  - do input/output (other programs)



Doing I/O

- Idea:

```
putStr :: String -> ()
-- Print a string on the console
```

- BUT: now

```
swap :: Stack w -> Stack w
```

might do arbitrary stateful things 

- And what does this do?

```
[putStr “yes”, putStr “no”]
```

  - What order are the things printed?
  - Are they printed at all?

Order of evaluation!

Laziness!

The main idea

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

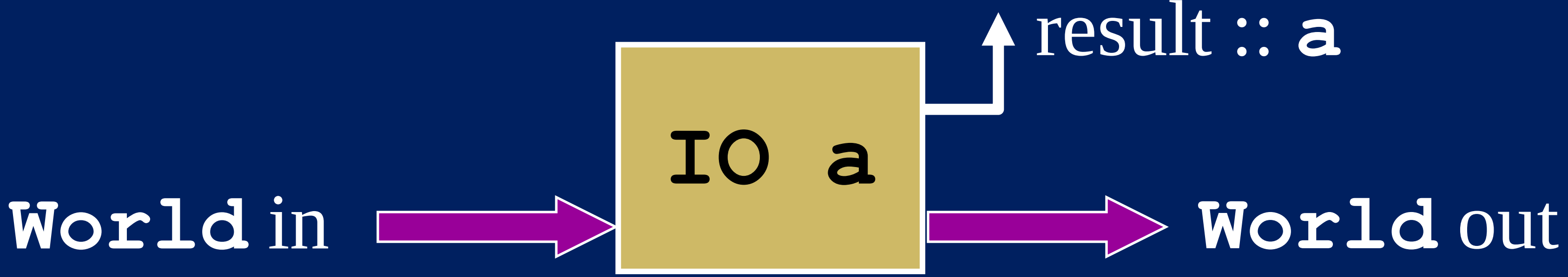
```
putStr :: String -> IO ()
-- Print a string on the console
```

- "Actions" sometimes called "computations"
- An action is a first class value
- Evaluating an action has no effect; performing the action has an effect

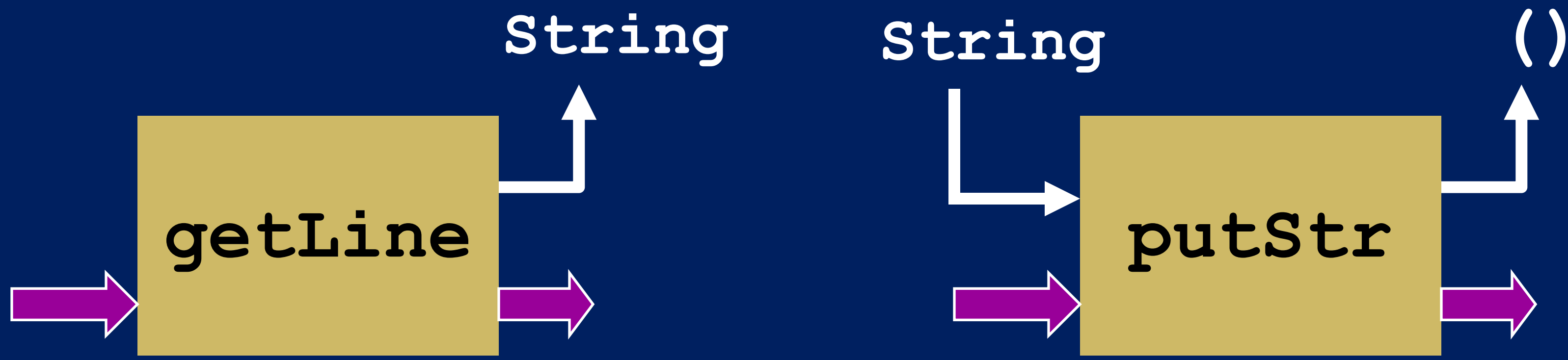
A helpful picture

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

```
type IO a = World -> (a, World)
-- An approximation
```



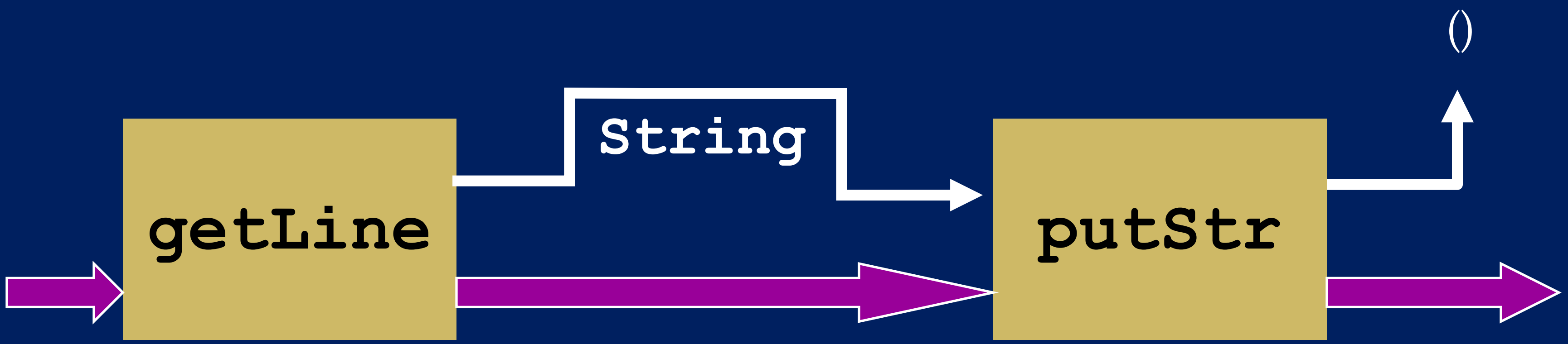
Simple I/O




```
getLine :: IO String
putStr  :: String -> IO ()


main :: IO ()
main = putStr “Hello world”
```

Main program is an action of type IO ()

Connecting actions up

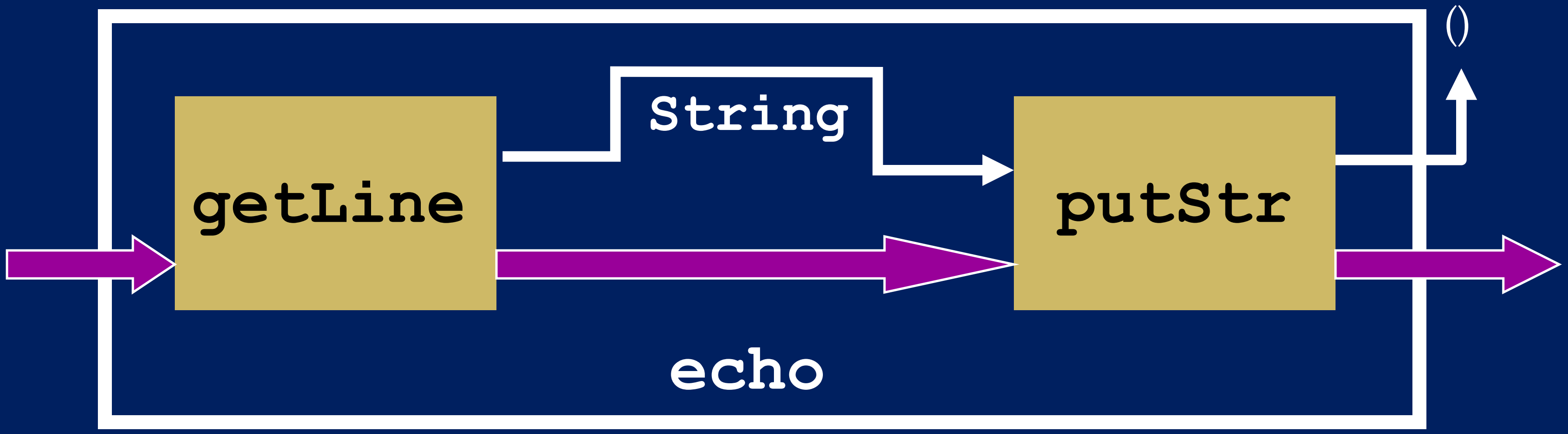


Goal:

read a line and then write it back out

Connecting actions up

```
echo :: IO ()
echo = do { l <- getLine; putStr l }
```



We have connected two actions to make a new, bigger action.

Getting two lines

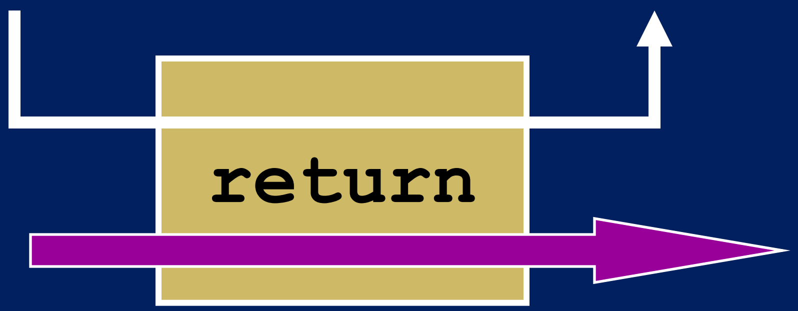
```
getTwoLines :: IO (String,String)
getTwoLines = do { s1 <- getLine
                 ; s2 <- getLine
                 ; ???? }
```

We want to just return (s1,s2)

The `return` combinator

```
getTwoLines :: IO (String,String)
getTwoLines = do { s1 <- getLine
                 ; s2 <- getLine
                 ; return (s1, s2) }
```

```
return :: a -> IO a
```



Desugaring do notation

- "do" notation adds only syntactic sugar
- Deliberately imperative look and feel

```
do { x<-e; s }    =    e >>= (\x -> do { s })
do { e }          =    e
```

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

Desugaring “do” notation

```
echo :: IO ()
echo = do { l <- getLine; putStr l }
```

```
echo = getLine >>= (\l -> putStr l)
```

A “lambda abstraction”

`(\x -> e)` means

“a function taking one parameter, x, and returning e”

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

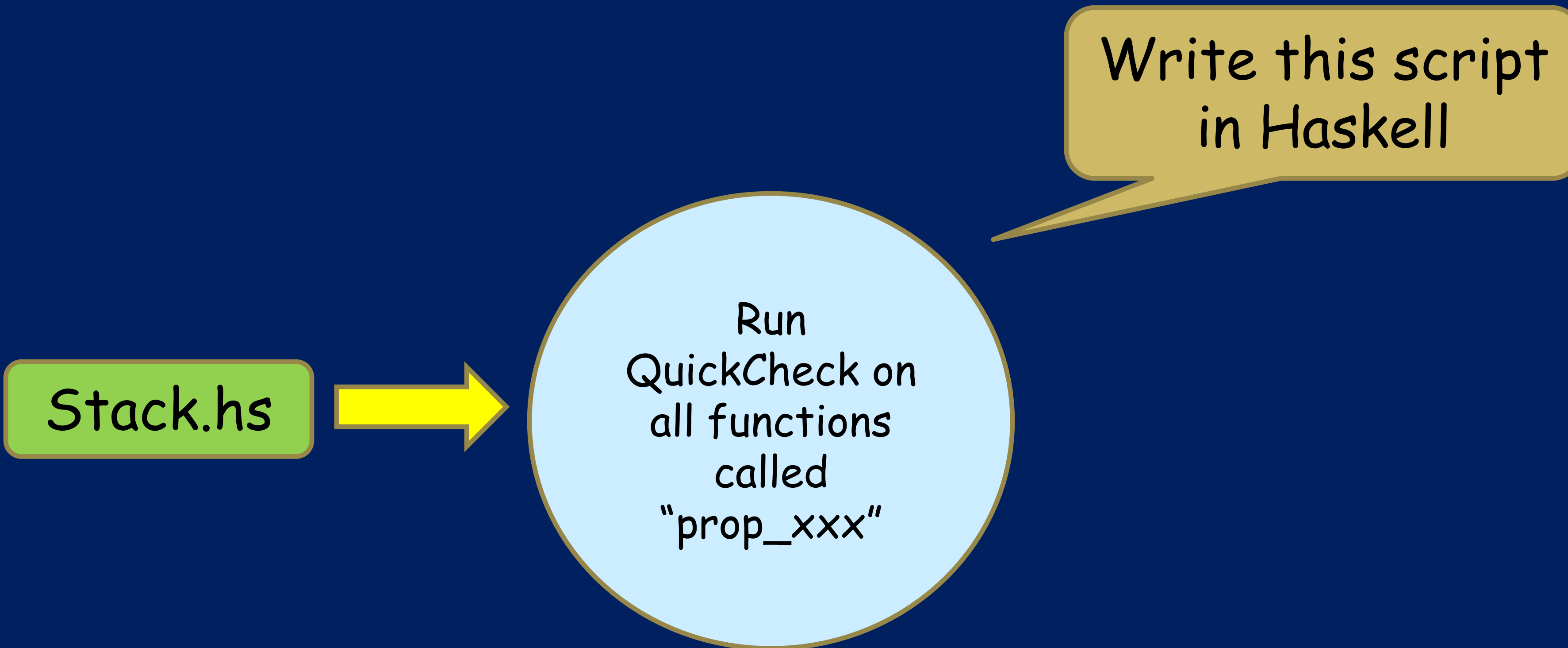
Using layout instead of braces

```
getTwoLines :: IO (String,String)
getTwoLines = do s1 <- getLine
                 s2 <- getLine
                 return (s1, s2)
```

- You can use
  - explicit braces/semicolons
  - or layout
  - or any mixture of the two

Scripting in Haskell

An example: scripting in Haskell




```
bash$ runhaskell QC.hs Stack.hs
prop_swap: +++ OK, passed 100 tests
prop_focusNP: +++ OK, passed 100 tests
```

Scripting in Haskell

```
module Main where

import System; import List

main :: IO ()
main = do { as <- getArgs
          ; mapM_ process as }

process :: String -> IO ()
process file = do  { cts <- readFile file
                   ; let tests = getTests cts

                   ; if null tests then
                         putStrLn (file ++ ": no properties to check")
                     else do

                   { writeFile "script" $
                        unlines ([":l " ++ file] ++ concatMap makeTest tests)
                   ; system ("ghci -v0 < script")
                   ; return () }}

getTests :: String -> [String]
getTests cts = nub $ filter ("prop_" `isPrefixOf`) $
                map (fst . head . lex) $ lines cts

makeTest :: String -> [String]
makeTest test = ["putStr \"" ++ p ++ ": \"", "quickCheck " ++ p]
```

Scripting in Haskell

Executables have module Main at top

Import libraries

Module Main must define main :: IO ()

```
module Main where

import System
import List

main :: IO ()
main = do { as <- getArgs
          ; mapM_ process as }
```

```
getArgs :: IO [String]
-- Gets command line args
```

```
mapM_ :: (a -> IO b) -> [a] -> IO ()
-- mapM_ f [x1, ..., xn]
-- = do { f x1;
--        ...
--        f xn;
--        return () }
```

Scripting in Haskell

```
process :: String -> IO ()
-- Test one file
process file
  = do { cts <- readFile file
       ; let tests = getTests cts
       ...
```

```
readFile:: String -> IO String
-- Gets contents of file
```

```
getTests:: String -> [String]
-- Extracts test functions
-- from file contents
```

e.g. tests = ["prop_rev", "prop_focus"]

Scripting in Haskell

```
process file = do   { cts <- readFile file
                    ; let tests = getTests cts

                    ; if null tests then
                           putStrLn (file ++ ": no properties to check")
                      else do

                    { writeFile "script" (
                         unlines ([":l " ++ file] ++
                                    concatMap makeTest tests))

                    ; system ("ghci -v0 < script")
                    ; return () }}
```

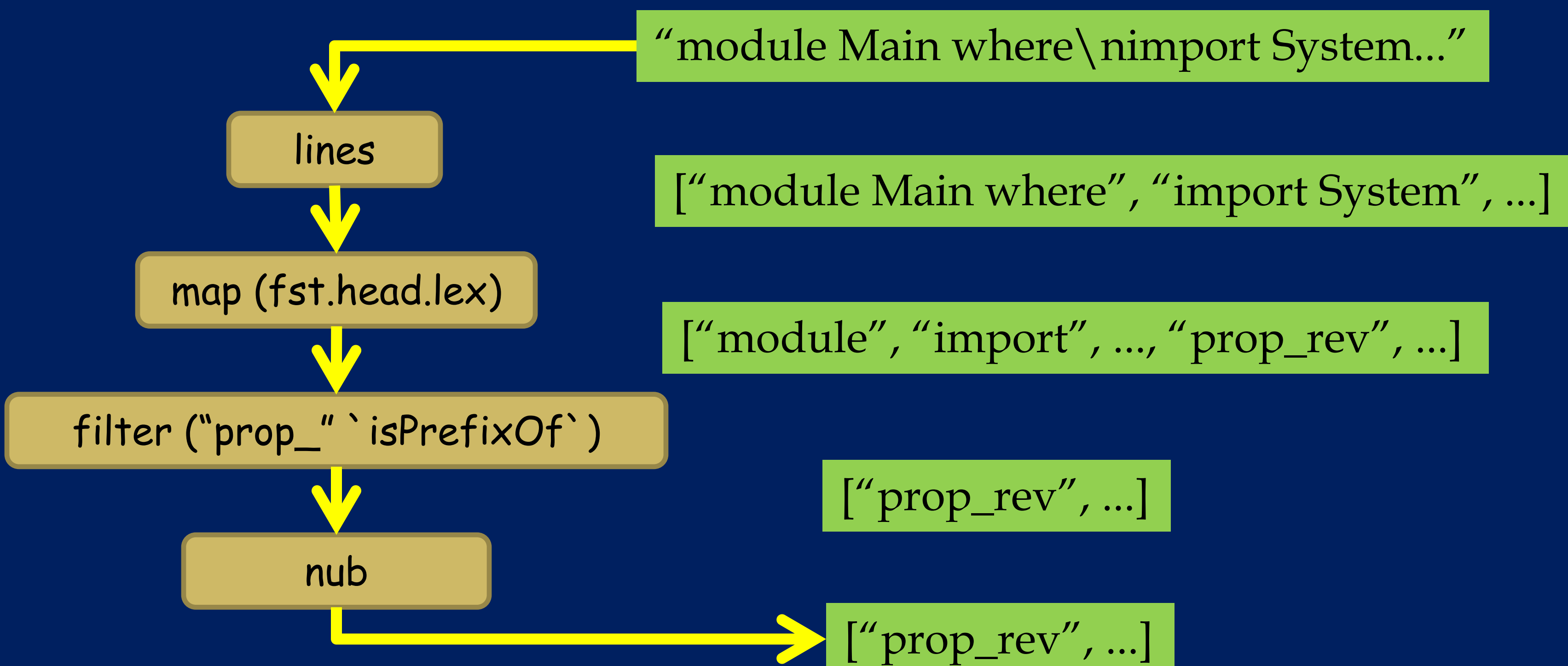
```
putStrLn  :: String -> IO ()
writeFile :: String -> String -> IO ()
system    :: String -> IO ExitCode

null      :: [a] -> Bool
makeTest  :: String -> [String]
concatMap :: (a->[b]) -> [a] -> [b]
unlines   :: [String] -> String
```

script

```
:l Stack.hs
putStr "prop_rev"
quickCheck prop_rev
putStr "prop_focus"
quickCheck prop_focus
```

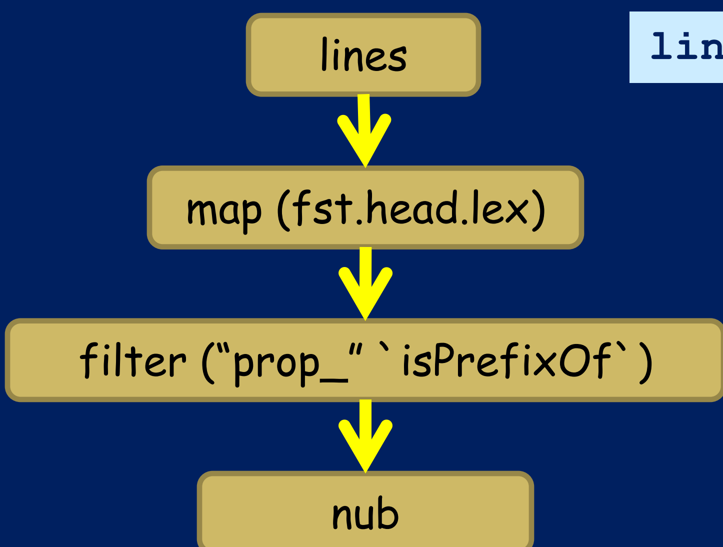
Scripting in Haskell

```haskell
getTests :: String -> [String]
getTests cts = nub (
               filter ("prop_" `isPrefixOf`) (
               map (fst . head . lex) (
               lines cts )))
```



Scripting in Haskell

```
getTests :: String -> [String]
getTests cts = nub (
               filter ("prop_" `isPrefixOf`) (
               map (fst . head . lex) (
               lines cts )))
```




```
lines :: String -> [String]

lex :: String -> [(String,String)]

filter :: (a->Bool) -> [a] -> [a]
isPrefixOf :: String -> String -> Bool

nub :: [String] -> [String]
-- Remove duplicates
```

Scripting in Haskell

```
makeTest :: String -> [String]
makeTest test = ["putStr \"" ++ p ++ ": \"“,
                 "quickCheck " ++ p ]
```

e.g

```
makeTest “prop_rev”

= [“putStr \”prop_rev: \””,
   “quickCheck prop_rev”]
```

What have we learned

- Scripting in Haskell is quick and easy (e.g. no need to compile, although you can)
- It is strongly typed; catches many errors
- But there are still many un-handled error conditions (no such file, not lexically-analysable, ...)

What have we learned

- Libraries are important; Haskell has a respectable selection
  - Regular expressions
  - Http
  - File-path manipulation
  - Lots of data structures (sets, bags, finite maps etc)
  - GUI toolkits (both bindings to regular toolkits such as Wx and GTK, and more radical approaches)
  - Database bindings

...but not (yet) as many as Perl, Python, *C#* etc

The types tell the story

```
type Company = String

sort :: [Company] -> [Company]
-- Sort lexicographically
-- Two calls given the same
-- arguments will give the
-- same results

sortBySharePrice :: [Company] -> IO [Company]
-- Consult current prices, and sort by them
-- Two calls given the same arguments may not
-- deliver the same results
```

I deliver a list of Company

I may do some I/O and then deliver a list of Company

Haskell: the world's finest imperative programming language

- Program divides into a mixture of
  - Purely functional code (most)
  - Necessarily imperative code (some)
- The type system keeps them rigorously separate
- Actions are first class, and that enables new forms of program composition (e.g. mapM_)

First-class control structures

Values of type (IO t) are first class

So we can define our own "control structures"

```
forever :: IO () -> IO ()
forever a = a >> forever a

repeatN :: Int -> IO () -> IO ()
repeatN 0 a = return ()
repeatN n a = a >> repeatN (n-1) a
```

e.g.

```
forever (do { e <- getNextEvent
            ; handleEvent e })
```

Foreign function interface

In the end we have to call C!

This call does not block

Calling convention

Header file and name of C procedure

Haskell

```
foreign import ccall unsafe "HsXlib.h XMapWindow"
  mapWindow :: Display -> Window -> IO ()
```

`mapWindow` calls `XMapWindow`

Haskell name and type of imported function

C

```
void XMapWindow( Display *d, Window *w ) {
  ...
}
```

Marshalling

All the fun is getting data across the border

```
data Display = MkDisplay Addr#
data Window  = MkWindow  Addr#
```

Addr#: a built-in type representing a C pointer

```
foreign import ccall unsafe "HsXlib.h XMapWindow"
  mapWindow :: Display -> Window -> IO ()
```

'foreign import' knows how to unwrap a single-constructor type, and pass it to C

Marshalling

All the fun is getting data across the border

```
data Display   = MkDisplay Addr#
data XEventPtr = MkXEvent  Addr#

foreign import ccall safe "HsXlib.h XNextEvent"
  xNextEvent:: Display -> XEventPtr -> IO ()
```

But what we want is

```
data XEvent = KeyEvent ... | ButtonEvent ...
            | DestroyWindowEvent ... | ...

nextEvent:: Display -> IO XEvent
```

Marshalling

```
data Display   = MkDisplay Addr#
data XEventPtr = MkXEvent  Addr#

foreign import ccall safe
                    "HsXlib.h XNextEvent"
   xNextEvent:: Display -> XEventPtr -> IO ()
```

Getting what we want is tedious...

```
data XEvent = KeyEvent ... | ButtonEvent ...
             | DestroyWindowEvent ... | ...

nextEvent:: Display -> IO XEvent
nextEvent d
  = do { xep <- allocateXEventPtr
       ; xNextEvent d xep
       ; type <- peek xep 3
       ; if type == 92 then
            do     { a <- peek xep 5
                   ; b <- peek xep 6
                   ; return (KeyEvent a b) }
           else if ... }
```

...but there are tools that automate much of the grotesque pain (hsc2hs, c2hs etc).

The rest of Haskell

Laziness

- Haskell is a **lazy** language
- Functions and data constructors don't evaluate their arguments until they need them

```
cond :: Bool -> a -> a -> a
cond True  t e = t
cond False t e = e
```

- Same with local definitions

```
abs :: Int -> Int
abs x | x>0       = x
      | otherwise = neg_x
      where
        neg_x = negate x
```

NB: new syntax guards

Why laziness is important

- Laziness supports **modular programming**
- Programmer-written functions instead of built-in language constructs

```
(||) :: Bool -> Bool -> Bool
True  || x = True
False || x = x
```

Short-circuiting "or"

Laziness and modularity

```haskell
isSubString :: String -> String -> Bool
x `isSubStringOf` s = or [ x `isPrefixOf` t
                         | t <- tails s ]
```

```haskell
tails :: String -> [String]
-- All suffixes of s
tails []     = [[]]
tails (x:xs) = (x:xs) : tails xs
```

type String = [Char]

```haskell
or :: [Bool] -> Bool
-- (or bs) returns True if any of the bs is True
or []     = False
or (b:bs) = b || or bs
```

Why laziness is important

- Typical paradigm:
  - generate all solutions (an enormous tree)
  - walk the tree to find the solution you want

```
nextMove :: Board -> Move
nextMove b = selectMove allMoves
  where
    allMoves = allMovesFrom b
```

A gigantic (perhaps infinite) tree of possible moves

Why laziness is important

- Generally, laziness unifies **data** with **control**
- Laziness also keeps Haskell pure, which is a Good Thing

Other language features

Advanced types

- Unboxed types
- Multi-parameter type classes
- Functional dependencies
- GADTs
- Implicit parameters
- Existential types
- etc etc

Concurrent Haskell (threads, communication, synchronisation)

Software Transactional Memory (STM)

Template Haskell
(meta programming)

Haskell language

Nested Data Parallel Haskell

Rewrite rules
(domain-specific compiler extensions)

Generic programming
One program that works over lots of different data structures

Monads, monad transformers, and arrows

Haskell's tool ecosystem

Programming environments (emacs, vim, Visual Studio)

Debugger

Interpreters (e.g. GHCi, Hugs)

Space and time profiling

Compilers (e.g. GHC, Jhc, Yhc)

Generators
- parser (cf yacc)
- lexer (cf lex)
- FFI

Haskell language

Coverage testing

Documentation generation (Haddock)

Testing (e.g. QuickCheck, Hunit)

Packaging and distribution (Cabal, Hackage)

LIBRARIES

Time profiling

GHC timing profile viewer

File View Help

Report Mon Mar 19 15:52 2007 Time and Allocation Profiling Report (Final)

Command catch_opt_prof +RTS -p -RTS Bernoulli_Safe -regress -nolog -time

Total time 1.25 sec

Total alloc 72,214,048 bytes

Cost Centre	Module	Entries	Individual %time	Individual %alloc	Inherited %time	Inherited %alloc
⊟ MAIN	MAIN	0	0.0	0.0	100.0	100.0
⊟ main	Main	1	0.0	0.0	96.0	99.6
⊟ execNormal	Main	2	0.0	0.0	92.0	99.6
concatMapM	General.General	3	8.0	0.0	8.0	0.0
⊟ execFile	Main	8	0.0	0.0	84.0	99.6
compile	Prepare.Compile	1	12.0	0.0	12.0	0.0
⊟ execMiddle	Main	12	0.0	0.0	56.0	82.1
⊞ loadStage	Main	7	0.0	0.0	8.0	14.8
⊞ getTask	Main	12	0.0	0.0	48.0	67.3
⊟ analyse	Analyse.All	2	0.0	0.0	16.0	17.4
⊟ precond	Analyse.Precond	24	0.0	0.0	16.0	16.8
⊞ backs	Analyse.Back	891	0.0	0.1	16.0	13.8

Viewer written in Haskell using GTK binding

Space profiling

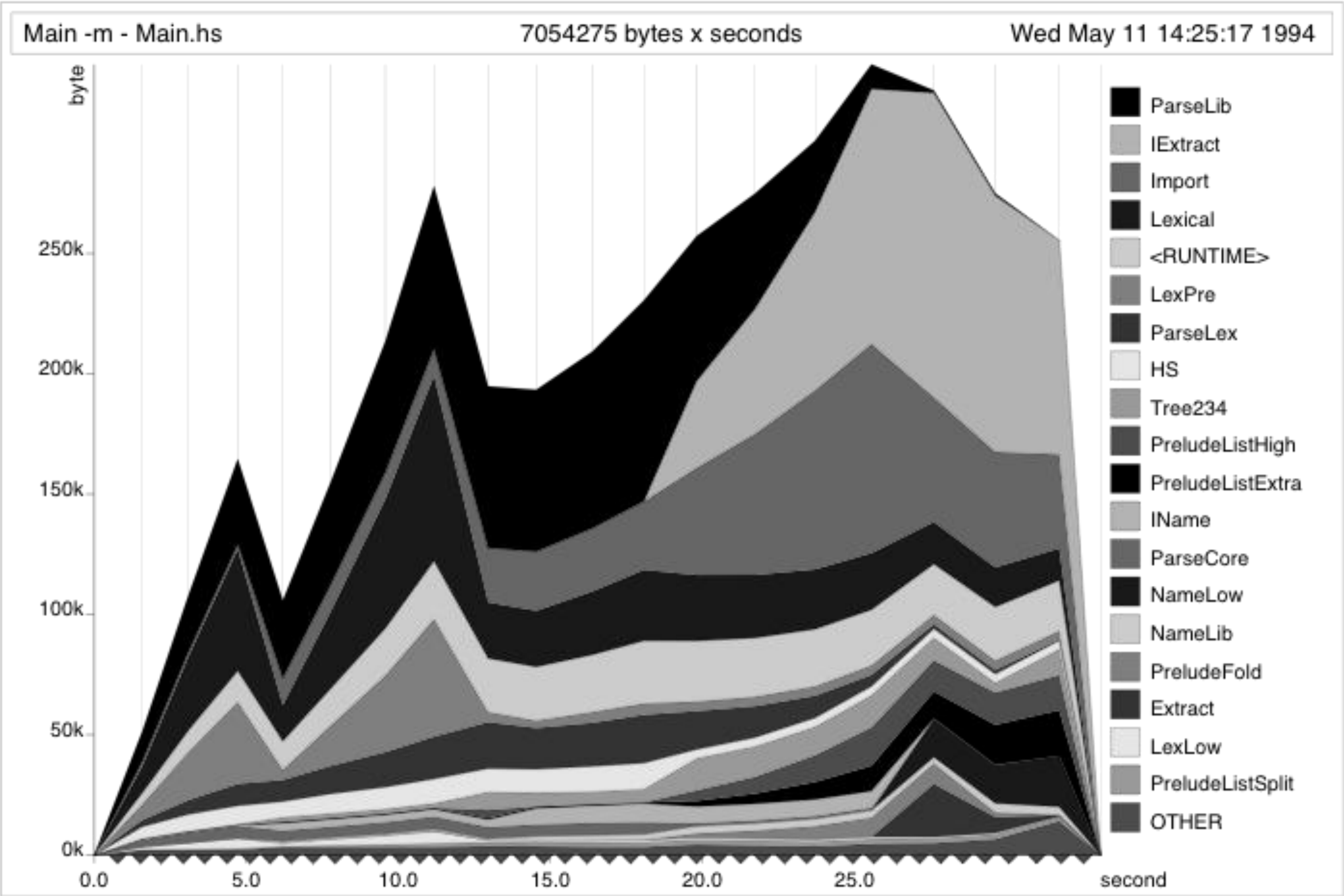


Fig. 18. Heap production of nhc by module, when compiling a small program.

Coverage checking (hpc)

Haskell program coverage - HaskellWiki - Windows Internet Explorer

http://haskell.org/haskellwiki/Haskell_program_coverage#Example_of_HTML_Summary_from_hpc-markup

Haskell program coverage - HaskellWiki

This is an example of the table that provides the summary of coverage, with links the the individually marked-up files.

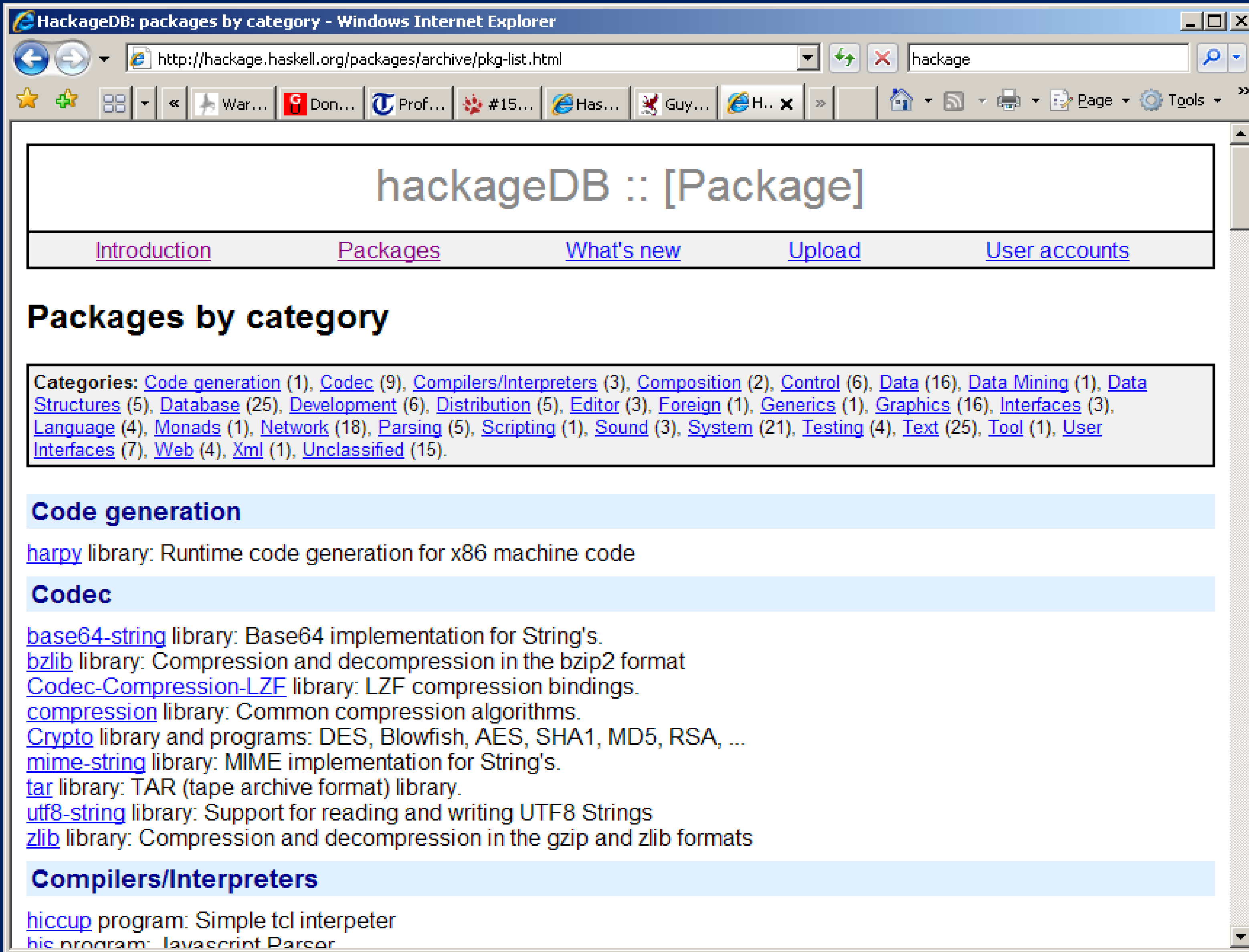
module	Top Level Definitions		Alternatives		Expressions	
	%	covered / total	%	covered / total	%	covered / total
`module CSG`	100 %	0/0	100 %	0/0	100 %	0/0
`module Construct`	48 %	17/35	52 %	25/48	60 %	381/635
`module Data`	24 %	6/25	13 %	11/81	39 %	254/646
`module Eval`	70 %	22/31	60 %	65/108	57 %	361/628
`module Geometry`	75 %	42/56	69 %	45/65	70 %	300/427
`module Illumination`	61 %	11/18	49 %	46/93	46 %	279/600
`module Intersections`	63 %	14/22	38 %	83/213	38 %	382/1001
`module Interval`	47 %	8/17	41 %	16/39	41 %	69/165
`module Main`	100 %	1/1	100 %	1/1	100 %	6/6
`module Misc`	0 %	0/1	0 %	0/1	0 %	0/10
`module Parse`	80 %	16/20	68 %	26/38	72 %	192/264
`module Primitives`	16 %	1/6	16 %	1/6	20 %	5/24
`module Surface`	36 %	4/11	24 %	13/53	18 %	43/231

Coverage checking (hpc)

```
reciprocal :: Int -> (String, Int)
reciprocal n | n > 1 = ('0' : '.' : digits, recur)
             | otherwise = error
                  "attempting to compute reciprocal of number <= 1"
  where
  (digits, recur) = divide n 1 []
divide :: Int -> Int -> [Int] -> (String, Int)
divide n c cs | c `elem` cs = ([], position c cs)
              | r == 0      = (show q, 0)
              | r /= 0      = (show q ++ digits, recur)
  where
  (q, r) = (c*10) `quotRem` n
  (digits, recur) = divide n r (c:cs)

position :: Int -> [Int] -> Int
position n (x:xs) | n==x      = 1
                  | otherwise = 1 + position n xs

showRecip :: Int -> String
showRecip n =
  "1/" ++ show n ++ " = " ++
  if r==0 then d else take p d ++ "(" ++ drop p d ++ ")"
  where
  p = length d - r
  (d, r) = reciprocal n

main = do
  number <- readLn
  putStrLn (showRecip number)
  main
```

Yellow: not executed
Red: boolean gave False
Green: boolean gave True

HackageDB (Haskell's CPAN)



hackageDB :: [Package]

Introduction Packages What's new Upload User accounts

Packages by category

Categories: Code generation (1), Codec (9), Compilers/Interpreters (3), Composition (2), Control (6), Data (16), Data Mining (1), Data Structures (5), Database (25), Development (6), Distribution (5), Editor (3), Foreign (1), Generics (1), Graphics (16), Interfaces (3), Language (4), Monads (1), Network (18), Parsing (5), Scripting (1), Sound (3), System (21), Testing (4), Text (25), Tool (1), User Interfaces (7), Web (4), Xml (1), Unclassified (15).

Code generation

harpy library: Runtime code generation for x86 machine code

Codec

base64-string library: Base64 implementation for String's.
bzlib library: Compression and decompression in the bzip2 format
Codec-Compression-LZF library: LZF compression bindings.
compression library: Common compression algorithms.
Crypto library and programs: DES, Blowfish, AES, SHA1, MD5, RSA, ...
mime-string library: MIME implementation for String's.
tar library: TAR (tape archive format) library.
utf8-string library: Support for reading and writing UTF8 Strings
zlib library: Compression and decompression in the gzip and zlib formats

Compilers/Interpreters

hiccup program: Simple tcl interpeter
hjs program: Javascript Parser

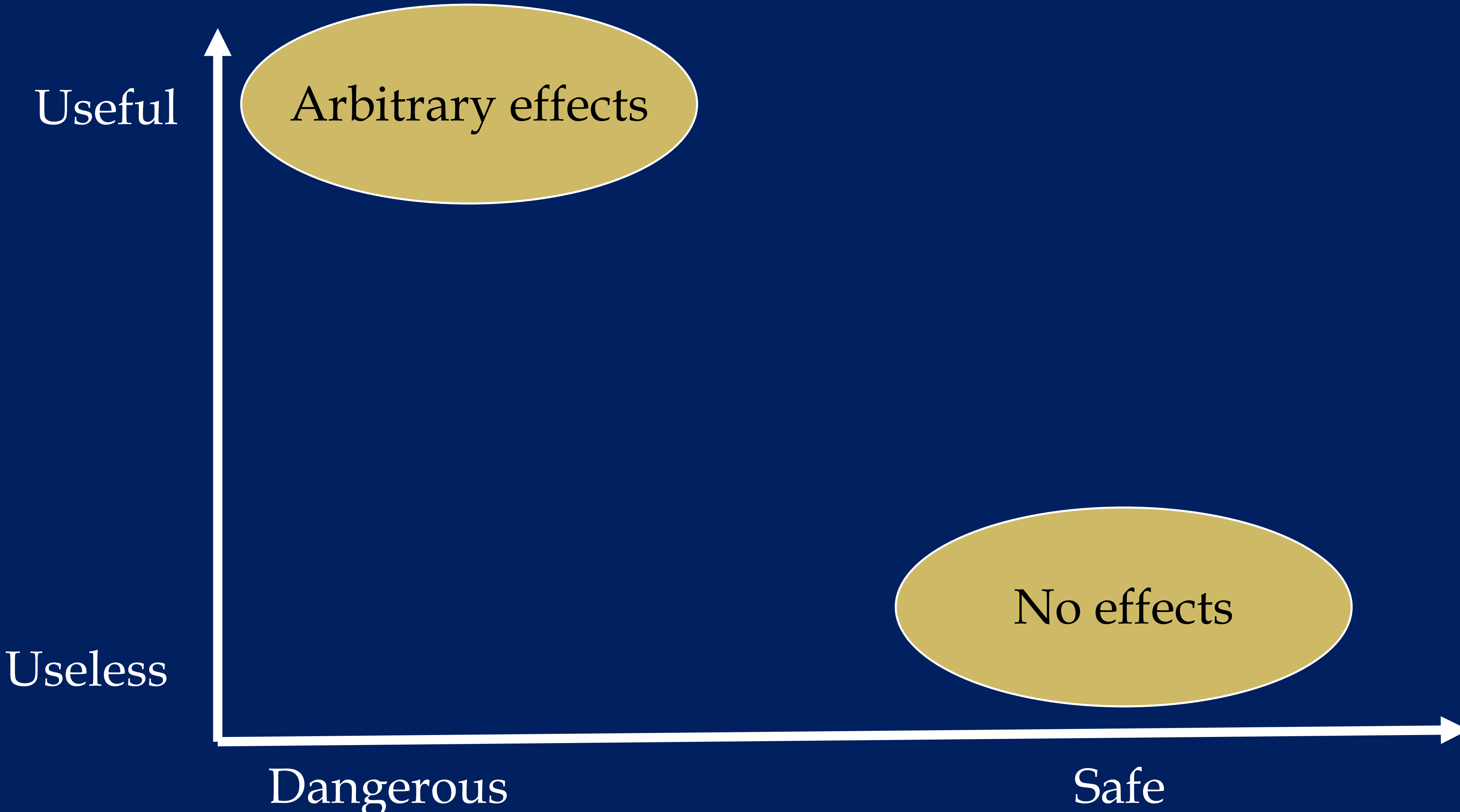
Cabal (Haskell's installer)

- A downloaded package, p, comes with
  - `p.cabal`: a package description
  - `Setup.hs`: a Haskell script to build/install

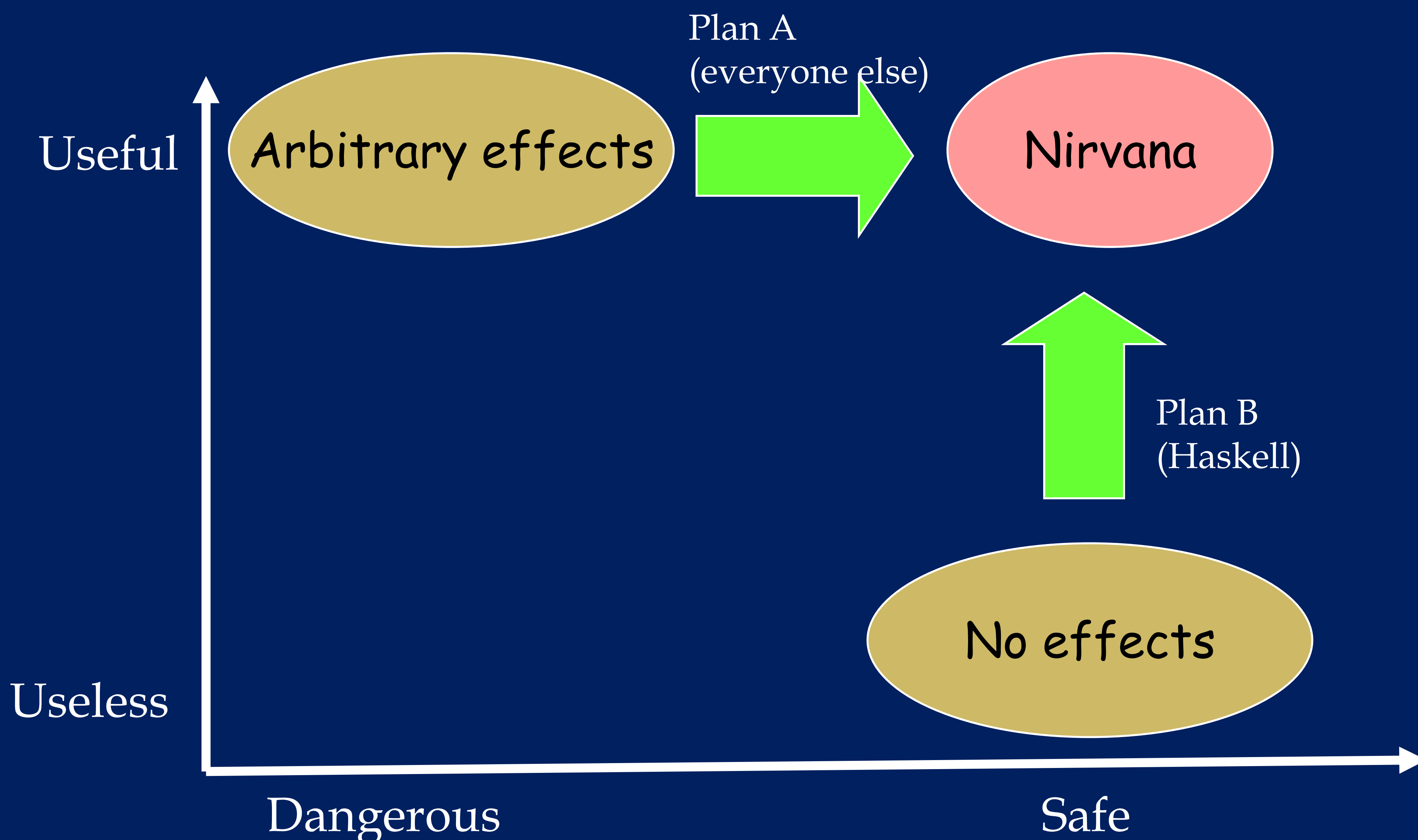
```
bash$ ./Setup.hs configure
bash$ ./Setup.hs build
bash$ ./Setup.hs install
```

Standing back...

The central challenge



The challenge of effects



Two basic approaches: Plan A

Arbitrary effects → Default = Any effect
Plan = Add restrictions

Examples

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

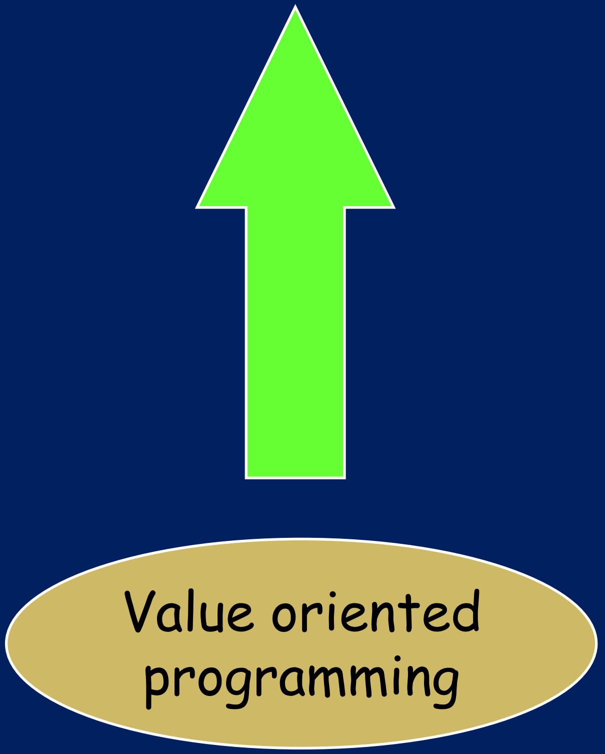
Two basic approaches: Plan B

Default = No effects
Plan = Selectively permit effects

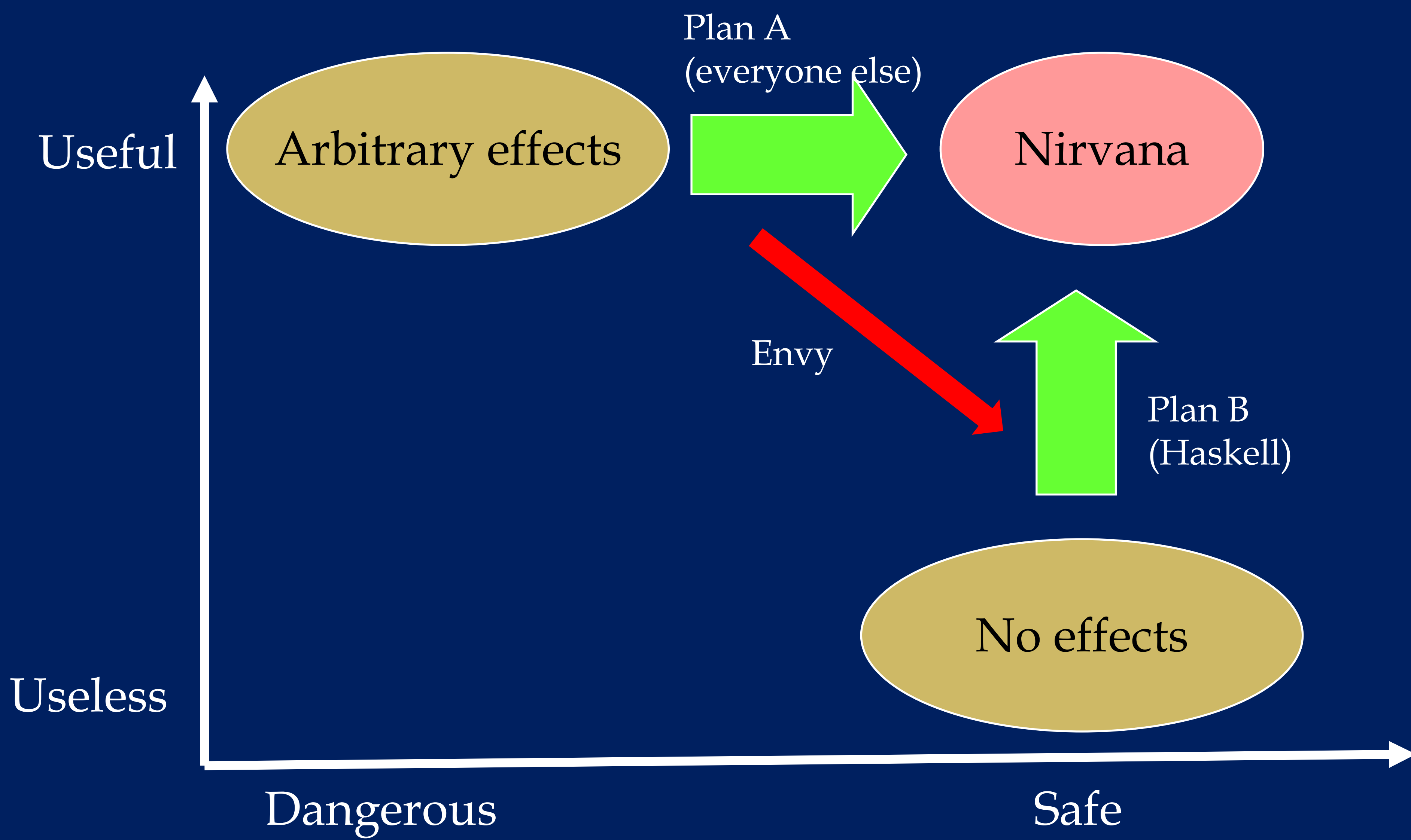
Types play a major role

Two main approaches:

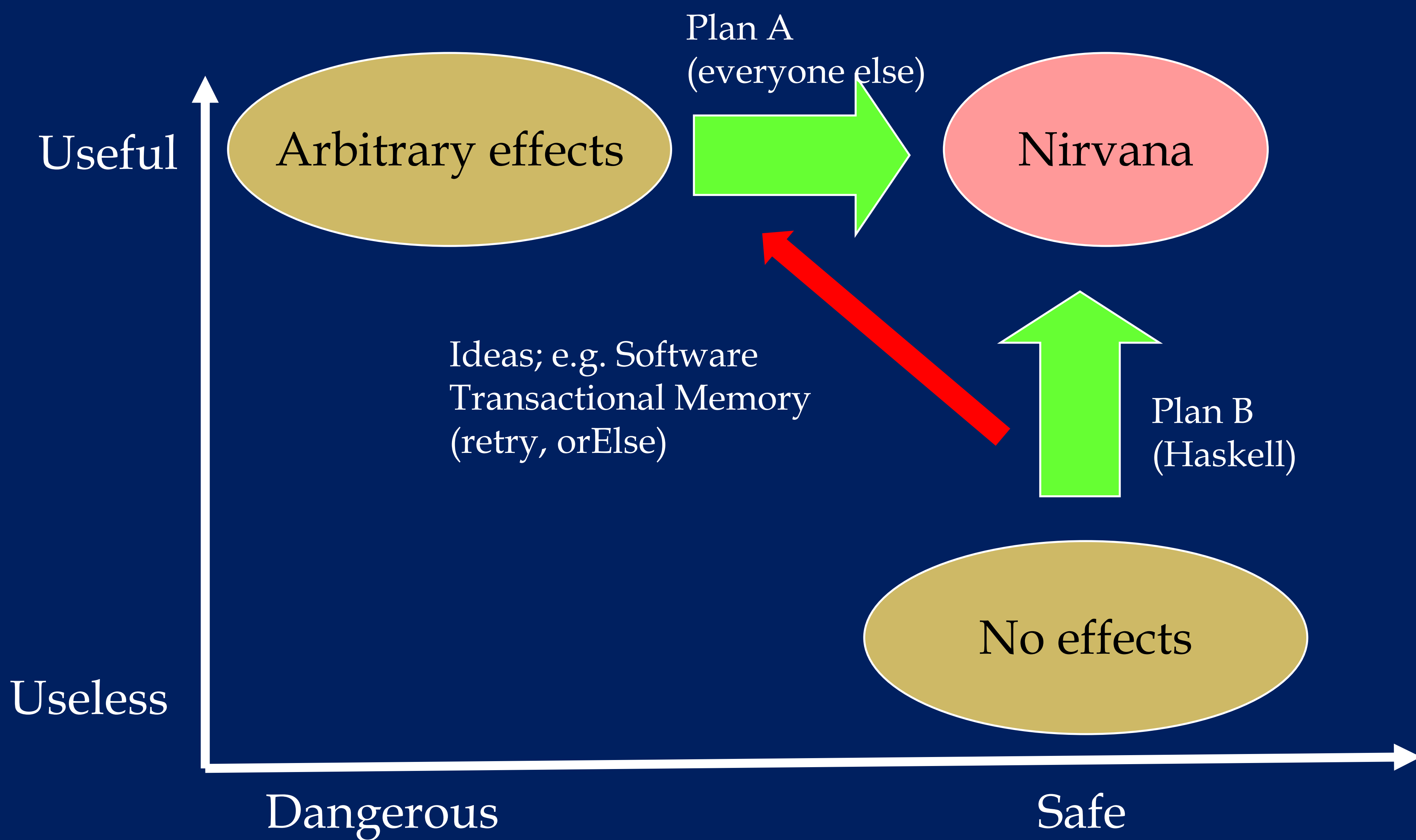
- Domain specific languages (SQL, XQuery, MDX, Google map/reduce)
- Wide-spectrum functional languages + controlled effects (e.g. Haskell)



Lots of cross-over



Lots of cross-over



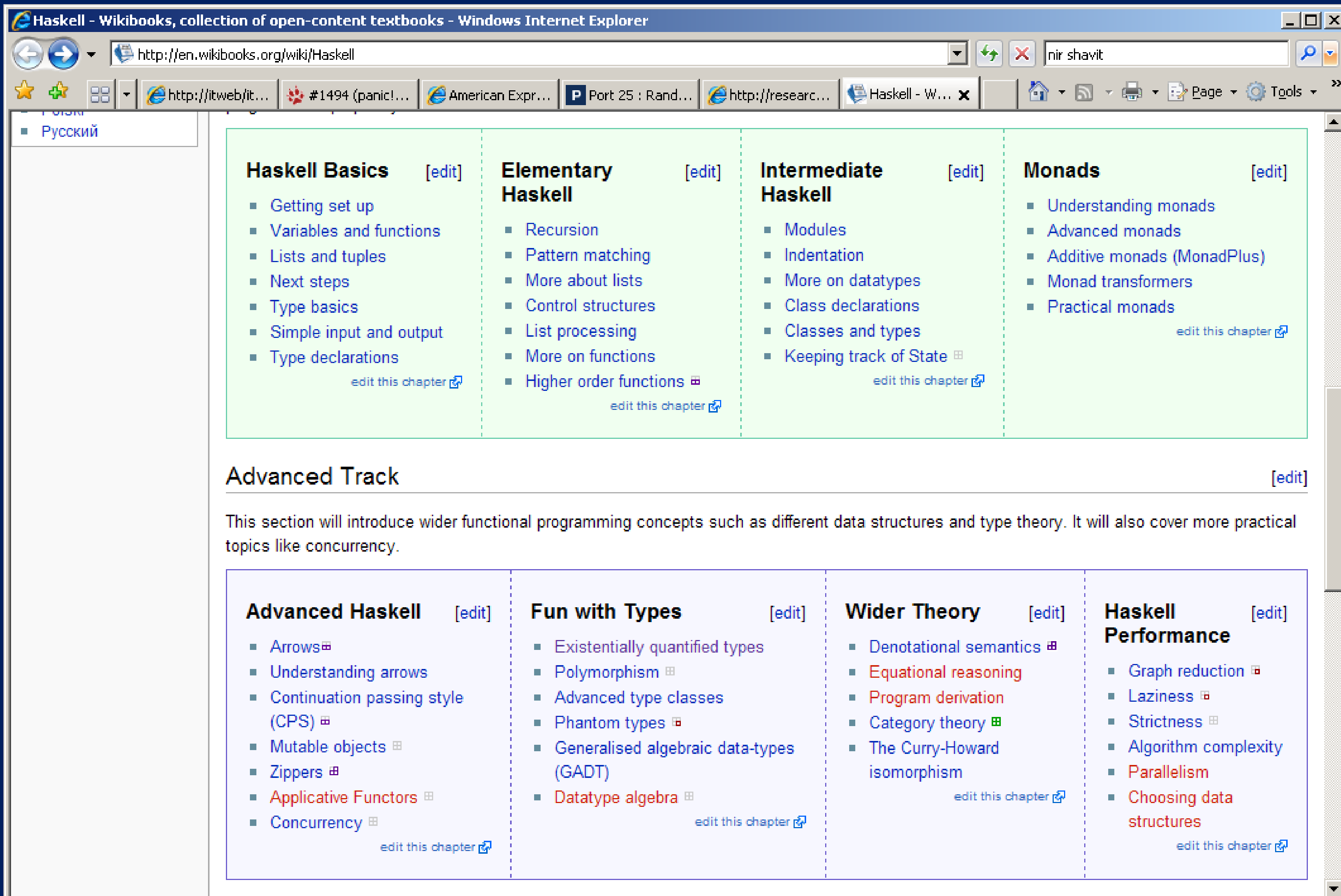
SLPJ conclusions

- One of Haskell's most significant contributions is to take purity seriously, and relentlessly pursue Plan B
- Imperative languages will embody growing (and checkable) pure subsets
- Knowing functional programming makes you a better Java/C#/Perl/Python/Ruby programmer

More info: haskell.org

- The Haskell wikibook
  - http://en.wikibooks.org/wiki/Haskell
- All the Haskell bloggers, sorted by topic
  - http://haskell.org/haskellwiki/Blog_articles
- Collected research papers about Haskell
  - http://haskell.org/haskellwiki/Research_papers
- Wiki articles, by category
  - http://haskell.org/haskellwiki/Category:Haskell
- Books and tutorials
  - http://haskell.org/haskellwiki/Books_and_tutorials

Wikibook



Haskell - Wikibooks, collection of open-content textbooks - Windows Internet Explorer

http://en.wikibooks.org/wiki/Haskell

- Русский

Haskell Basics [edit]

- Getting set up
- Variables and functions
- Lists and tuples
- Next steps
- Type basics
- Simple input and output
- Type declarations

edit this chapter

Elementary Haskell [edit]

- Recursion
- Pattern matching
- More about lists
- Control structures
- List processing
- More on functions
- Higher order functions

edit this chapter

Intermediate Haskell [edit]

- Modules
- Indentation
- More on datatypes
- Class declarations
- Classes and types
- Keeping track of State

edit this chapter

Monads [edit]

- Understanding monads
- Advanced monads
- Additive monads (MonadPlus)
- Monad transformers
- Practical monads

edit this chapter

Advanced Track [edit]

This section will introduce wider functional programming concepts such as different data structures and type theory. It will also cover more practical topics like concurrency.

Advanced Haskell [edit]

- Arrows
- Understanding arrows
- Continuation passing style (CPS)
- Mutable objects
- Zippers
- Applicative Functors
- Concurrency

edit this chapter

Fun with Types [edit]

- Existentially quantified types
- Polymorphism
- Advanced type classes
- Phantom types
- Generalised algebraic data-types (GADT)
- Datatype algebra

edit this chapter

Wider Theory [edit]

- Denotational semantics
- Equational reasoning
- Program derivation
- Category theory
- The Curry-Howard isomorphism

edit this chapter

Haskell Performance [edit]

- Graph reduction
- Laziness
- Strictness
- Algorithm complexity
- Parallelism
- Choosing data structures

edit this chapter

More info: haskell.org