Wearing the hair shirt
A retrospective on Haskell

Simon Peyton Jones
Microsoft Research, Cambridge

The primoridal soup

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

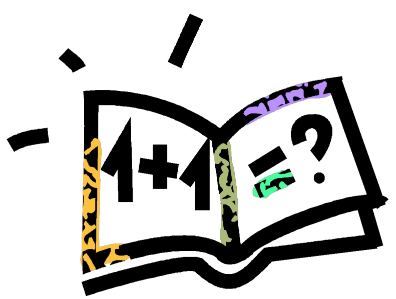
Led to...a succession of face-to-face meetings

April 1990: **Haskell 1.0** report released (editors: Hudak, Wadler)

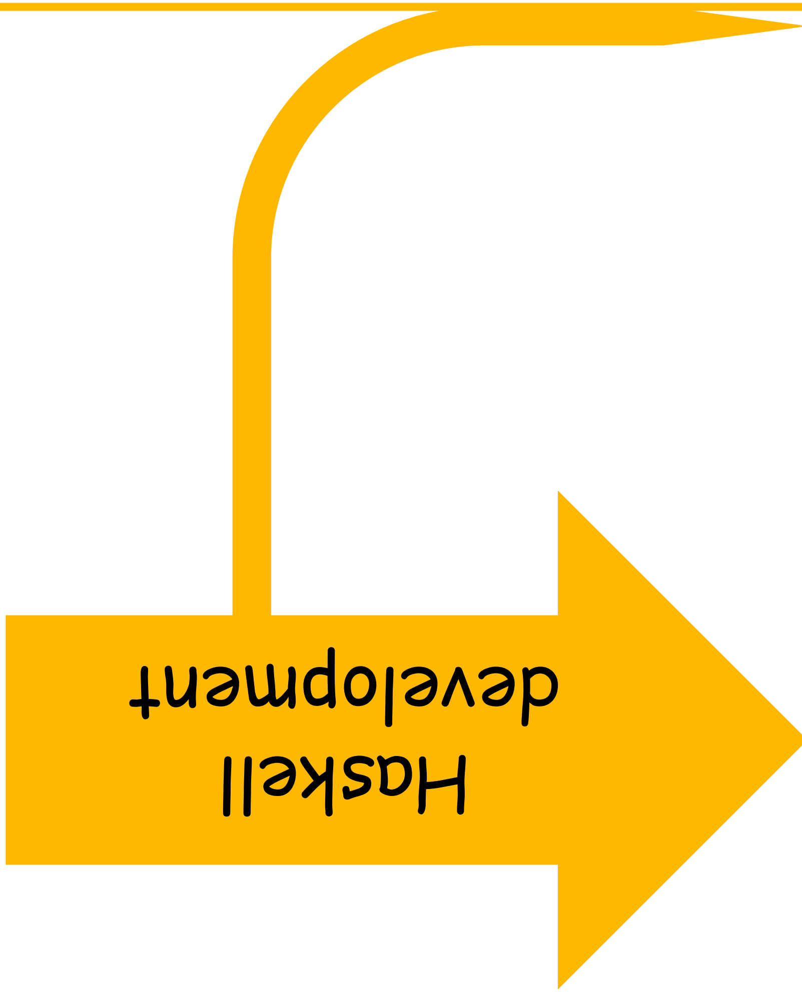
Timeline

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

The Book!



Haskell 98



Haskell 98

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

Haskell + extensions

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

Reflections on the process

- The idea of having a fixed standard (Haskell 98) in parallel with an evolving language, has worked really well
- Formal semantics only for fragments (but see [Faxen2002])
- A smallish, rather pointy-headed user-base makes Haskell nimble. Haskell has evolved rapidly and continues to do so.

Motto: **avoid success at all costs**

The price of usefulness

- Libraries increasingly important:
  - 1996: Separate libraries Report
  - 2001: Hierarchical library naming structure, increasingly populated
- Foreign-function interface increasingly important
  - 1993 onwards: a variety of experiments
  - 2001: successful effort to standardise a FFI across implementations
- Any language large enough to be useful is dauntingly complex

Syntax

Syntax

Syntax is not important

Parsing is the easy bit of a compiler

Syntax

~~Syntax is not important~~

Syntax is the user interface of a language

~~Parsing is the easy bit of a compiler~~

The parser is often the trickiest bit of a compiler

Good ideas from other languages

List comprehensions

```
[(x,y) | x <- xs, y <- ys, x+y < 10]
```

Separate type signatures

```
head :: [a] -> a
head (x:xs) = x
```

Upper case constructors

```
f True true = true
```

DIY infix operators

```
f `map` xs
```

Optional layout

```
let x = 3
    y = 4
in x+y
```

```
let { x = 3; y = 4} in x+y
```

Fat vs thin

Expression style

- Let
- Lambda
- Case
- If

Declaration style

- Where
- Function arguments on lhs
- Pattern-matching
- Guards

SLPJ's conclusion
syntactic redundancy is a big win

Tony Hoare's comment "I fear that Haskell is doomed to succeed"

“Declaration style“

Define a function as a series of independent equations

```
map f []     = []
map f (x:xs) = f x : map f xs
```

```
sign x   | x>0 = 1
   | x==0   = 0
   | x<0 = -1
```

"Expression style"

Define a function as an expression

```
map = \f xs -> case xs of
             []     -> []
             (x:xs) -> map f xs
```

```
sign = \x -> if x>0 then 1
   else if x==0 then 0
   else -1
```

Example (ICFP02 prog comp)

Pattern match

Guard

Pattern guard

Conditional

Where clause

```
sp_help item@(Item cur_loc cur_link _) wq vis
  | cur_length > limit     -- Beyond limit
  = sp wq vis
  | Just vis_link <- lookupVisited vis cur_loc
  =      -- Already visited; update the visited
         -- map if cur_link is better
    if cur_length >= linkLength vis_link then
         -- Current link is no better
          sp wq vis
    else
         -- Current link is better
          emit vis item ++ sp wq vis'

  | otherwise   -- Not visited yet
  = emit vis item ++ sp wq' vis'
  where
    vis' = ...
    wq   = ...
```

So much for syntax...

What is important or interesting about Haskell?

What really matters?

Laziness

Type classes

Sexy types

Laziness

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

But...

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

So why wear the hair shirt of laziness?

In favour of laziness

Laziness is jolly convenient

```
sp_help item@(Item cur_loc cur_link _) wq vis
  | cur_length > limit     -- Beyond limit
  = sp wq vis
  | Just vis_link <- lookupVisited vis cur_loc
  = if cur_length >= linkLength vis_link then
          sp wq vis
    else
          emit vis item ++ sp wq vis'

  | otherwise
  = emit vis item ++ sp wq' vis'
  where
    vis' = ...
    wq'  = ...
```

Used in two cases

Used in one case

Combinator libraries

Recursive values are jolly useful

```
type Parser a = String -> (a, String)

exp :: Parser Expr
exp = lit "let" <+> decls <+> lit "in" <+> exp
   ||| exp <+> aexp
   ||| ...etc...
```

This is illegal in ML, because of the value restriction

Can only be made legal by eta expansion.

But that breaks the Parser abstraction,
and is extremely gruesome:

```
exp x = (lit "let" <+> decls <+> lit "in" <+> exp
     ||| exp <+> aexp
    ||| ...etc...) x
```

The big one....

Laziness keeps you honest

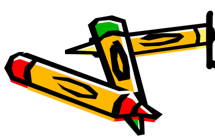
- Every call-by-value language has given into the siren call of side effects
- But in Haskell

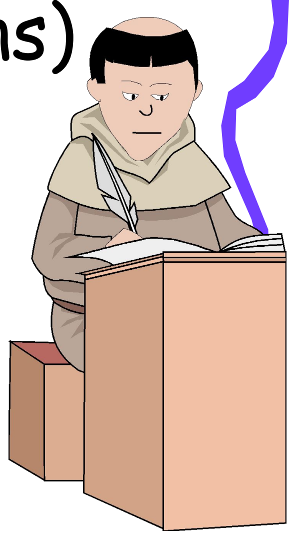
  ```
  (print "yes") + (print "no")
  ```

  just does not make sense. Even worse is

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

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





Monadic I/O

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

eg.

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

Performing I/O

```
main :: IO a
```

- A program is a single I/O action
- Running the program performs the action
- Can't do I/O from pure code.
- Result: clean separation of pure code from imperative code

Connecting I/O operations

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

eg.

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

The do-notation

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

==

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

- Syntactic sugar only
- Easy translation into (>>=), return
- Deliberately imperative “look and feel”

Control structures

Values of type (IO t) are first class

So we can define our own "control structures"

```
forever :: IO () -> IO ()
forever a = do { a; forever a }

repeatN :: Int -> IO () -> IO ()
repeatN 0 a = return ()
repeatN n a = do { a; repeatN (n-1) a }
```

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

Generalising the idea

A monad consists of:

- A type constructor M
- bind :: M a -> (a -> M b) -> M b
- unit :: a -> M a
- PLUS some per-monad operations (e.g. getChar :: IO Char)

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

Monads

- Exceptions

```
type Exn a = Either String a
fail :: String -> Exn a
```

- Unique supply

```
type Uniq a = Int -> (a, Int)
new :: Uniq Int
```

- Parsers

```
type Parser a = String -> [(a,String)]
alt :: Parser a -> Parser a -> Parser a
```

Monad combinators (e.g. sequence, fold, etc), and do-notation, work over all monads

Example: a type checker

```
tcExpr :: Expr -> Tc Type
tcExpr (App fun arg)
    = do { fun_ty <- tcExpr fun
         ; arg_ty <- tcExpr arg
         ; res_ty <- newTyVar
         ; unify fun_ty (arg_ty --> res_ty)
         ; return res_ty }
```

Tc monad hides all the plumbing:

- Exceptions and failure
- Current substitution (unification)
- Type environment
- Current source location
- Manufacturing fresh type variables

Robust to changes in plumbing

The IO monad

The IO monad allows controlled introduction of other effect-ful language features (not just I/O)

- State

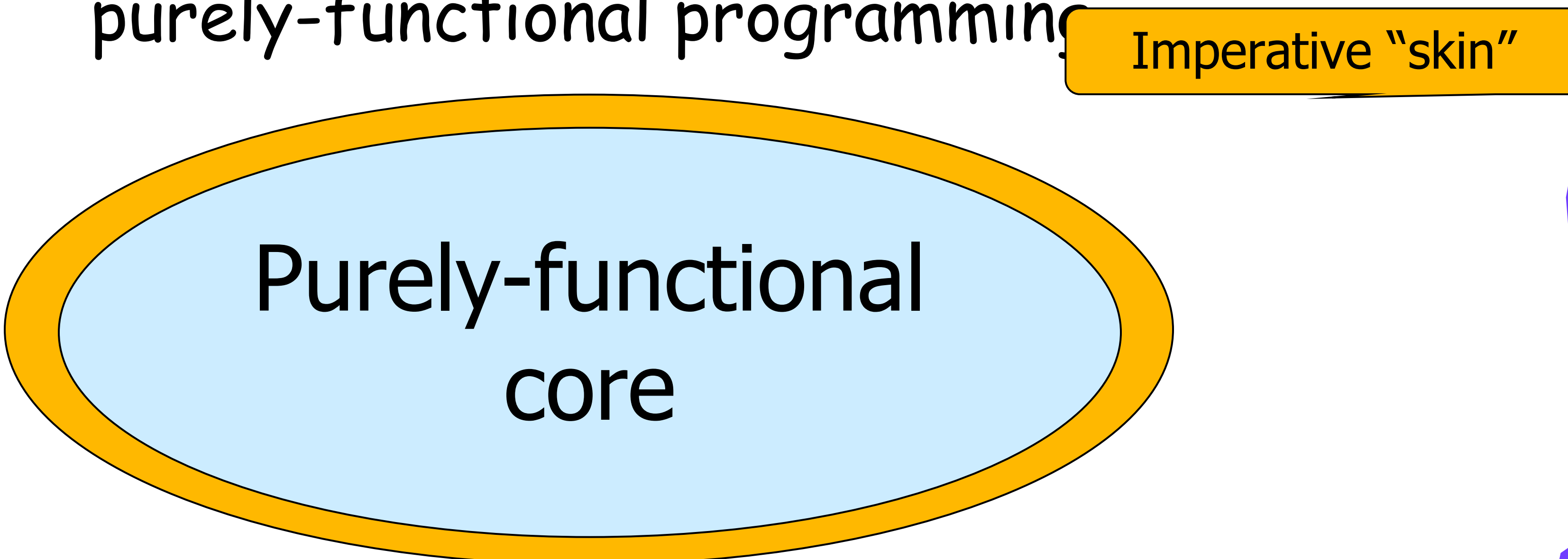
```
newRef :: IO (IORef a)
read   :: IORef s a -> IO a
write  :: IORef s a -> a -> IO ()
```

- Concurrency

```
fork      :: IO a -> IO ThreadId
newMVar   :: IO (MVar a)
takeMVar :: MVar a -> IO a
putMVar  :: MVar a -> a -> IO ()
```

What have we achieved?

- The ability to mix imperative and purely-functional programming



What have we achieved?

- ...without ruining either
- All laws of pure functional programming remain unconditionally true, even of actions

e.g. let x=e in ...x....x...

=

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

What we have not achieved

- Imperative programming is no easier than it always was

e.g. `do { ...; x <- f 1; y <- f 2; ...}`

?=?

`do { ...; y <- f 2; x <- f 1; ...}`

- ...but there's less of it!
- ...and actions are first-class values

Open challenge 1

Open problem: the IO monad has become Haskell's sin-bin. (Whenever we don't understand something, we toss it in the IO monad.)

Festering sore:

```
unsafePerformIO :: IO a -> a
```

Dangerous, indeed type-unsafe, but occasionally indispensable.

Wanted: finer-grain effect partitioning

e.g. IO {read x, write y} Int

Open challenge 2

Which would you prefer?

```
do { a <- f x;
     b <- g y;
     h a b }
```

```
h (f x) (g y)
```

In a commutative monad, it does not matter whether we do `(f x)` first or `(g y)`.

Commutative monads are very common. (Environment, unique supply, random number generation.) For these, monads over-sequentialise.

Wanted: theory and notation for some cool compromise.

Monad summary

- Monads are a beautiful example of a theory-into-practice (more the thought pattern than actual theorems)
- Hidden effects are like hire-purchase: pay nothing now, but it catches up with you in the end
- Enforced purity is like paying up front: painful on Day 1, but usually worth it
- But we made one big mistake...

Our biggest mistake

Using the scary term "monad"

rather than

"warm fuzzy thing"

What really matters?

~~Laziness~~

Purity and monads

Type classes

Sexy types

SLPJ conclusions

- Purity is more important than, and quite independent of, laziness
- The next ML will be pure, with effects only via monads. The next Haskell will be strict, but still pure.
- Still unclear exactly how to add laziness to a strict language. For example, do we want a type distinction between (say) a lazy Int and a strict Int?

Type classes

Type classes

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

instance Eq Int where
  i1 == i2 = eqInt i1 i2

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


member :: Eq a => a -> [a] -> Bool
member x []      = False
member x (y:ys)| x==y    = True
   | otherwise = member x ys
```

Initially, just a neat way to get systematic overloading of (==), read, show.

Implementing type classes

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

dEqInt :: Eq Int
dEqInt = MkEq eqInt

dEqList :: Eq a -> Eq [a]
dEqList (MkEq e) = MkEq el
  where el []     []     = True
        el (x:xs) (y:ys) = x `e` y && xs `el` ys

member :: Eq a -> a -> [a] -> Bool
member d x []              = False
member d x (y:ys) | eq d x y  = True
   | otherwise = member d x ys
```

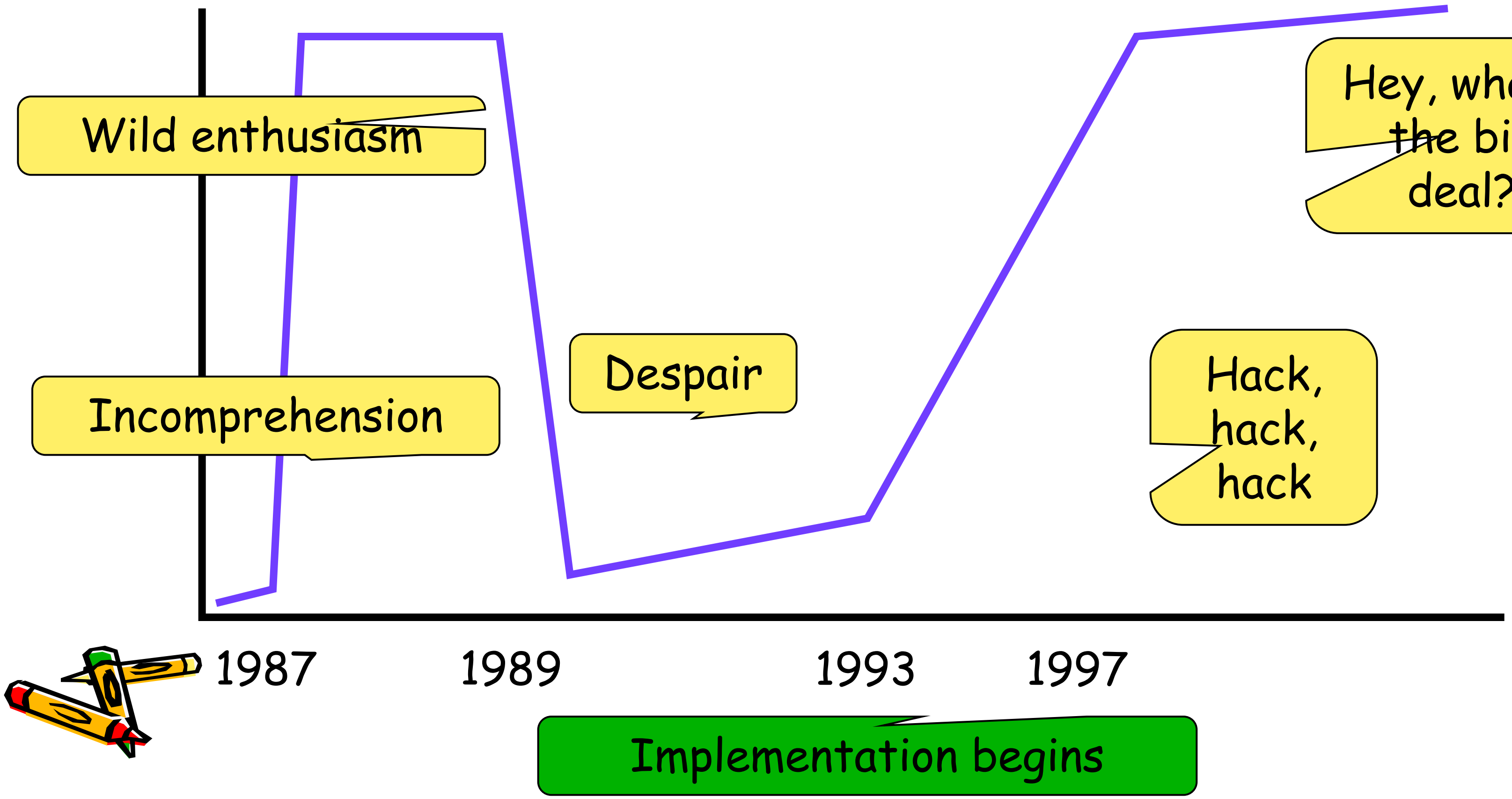
Class witnessed by a “dictionary” of methods

Instance declarations create dictionaries

Overloaded functions take extra dictionary parameter(s)

Type classes over time

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



Type classes are useful

Type classes have proved extraordinarily convenient in practice

- Equality, ordering, serialisation, numerical operations, *and not just the built-in ones (e.g. pretty-printing, time-varying values)*
- Monadic operations

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
  fail   :: String -> m a
```

Note the higher-kinded type variable, m

Quickcheck

```
propRev :: [Int] -> Bool
propRev xs = reverse (reverse xs) == xs

propRevApp :: [Int] -> [Int] -> Bool
propRevApp xs ys = reverse (xs++ys) ==
        reverse ys ++ reverse xs
```

```
ghci> quickCheck propRev
 OK: passed 100 tests

ghci> quickCheck propRevApp
OK: passed 100 tests
```

Quickcheck (which is just a Haskell 98 library)

- Works out how many arguments
- Generates suitable test data
- Runs tests

Quickcheck

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

class Test a where
  test :: a -> Rand -> Bool

class Arby a where
  arby :: Rand -> a

instance (Arby a, Test b) => Test (a->b) where
  test f r = test (f (arby r1)) r2
           where (r1,r2) = split r

instance Test Bool where
  test b r = b
```

Extensiblity

- Like OOP, one can add new data types "later". E.g. QuickCheck works for your new data types (provided you make them instances of **`Arby`**)
- ...but also not like OOP

Type-based dispatch

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

- A bit like OOP, except that method suite passed separately?

```
double :: Num a => a -> a
double x = x+x
```

- No: type classes implement type-based dispatch, not value-based dispatch

Type-based dispatch

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

```
double :: Num a => a -> a
  double x = 2*x
```

means

```
double :: Num a -> a -> a
  double d x = mul d (fromInteger d 2)
  x
```

The overloaded value is *returned by* ***fromInteger***, not passed to it. It is the dictionary (and type) that are passed as argument to **fromInteger**

Type-based dispatch

So the links to intensional polymorphism are much closer than the links to OOP.

The dictionary is like a proxy for the (interesting aspects of) the type argument of a polymorphic function.

Intensional polymorphism

```
f :: forall a. a -> Int
f t (x::t) = ...typecase t...
```

Haskell

```
f :: forall a. C a => a -> Int
f x = ...(call method of C)...
```

C.f. Crary et al λR (ICFP98), Baars et al (ICFP02)

Cool generalisations

- Multi-parameter type classes
- Higher-kinded type variables (a.k.a. constructor classes)
- Overlapping instances
- Functional dependencies (Jones ESOP'00)
- Type classes as logic programs (Neubauer et al POPL'02)

Qualified types

- Type classes are an example of qualified types [Jones thesis]. Main features
  - types of form $\forall\alpha.Q \Rightarrow \tau$
  - qualifiers Q are witnessed by run-time evidence
- Known examples
  - type classes (evidence = tuple of methods)
  - implicit parameters (evidence = value of implicit param)
  - extensible records (evidence = offset of field in record)
- Another unifying idea: Constraint Handling Rules (Stucky/Sulzmann ICFP'02)

Type classes summary

- A much more far-reaching idea than we first realised
- Many interesting generalisations
- Variants adopted in Isabel, Clean, Mercury, Hal, Escher
- Open questions:
  - tension between desire for overlap and the open-world goal
  - danger of death by complexity

Sexy types

Sexy types

Haskell has become a laboratory and playground for advanced type hackery

- Polymorphic recursion
- Higher kinded type variables
```
data T k a = T a (k (T k a))
```
- Polymorphic functions as constructor arguments
```
data T = MkT (forall a. [a] -> [a])
```
- Polymorphic functions as arbitrary function arguments (higher ranked types)
```
f :: (forall a. [a]->[a]) -> ...
```
- Existential types
```
data T = exists a. Show a => MkT a
```

Is sexy good? Yes!

- Well typed programs don't *go wrong*
- Less mundanely (but more allusively) sexy types let you think higher thoughts and still stay [almost] sane:
  - deeply higher-order functions
  - functors
  - folds and unfolds
  - monads and monad transformers
  - arrows (now finding application in real-time reactive programming)
  - short-cut deforestation
  - bootstrapped data structures

How sexy?

- Damas-Milner is on a cusp:
  - Can infer most-general types without any type annotations at all
  - But virtually any extension destroys this property
- Adding type quite modest type annotations lets us go a LOT further (as we have already seen) without losing inference for most of the program.
- Still missing from even the sexiest Haskell impls
  - λ at the type level
  - Subtyping
  - Impredicativity

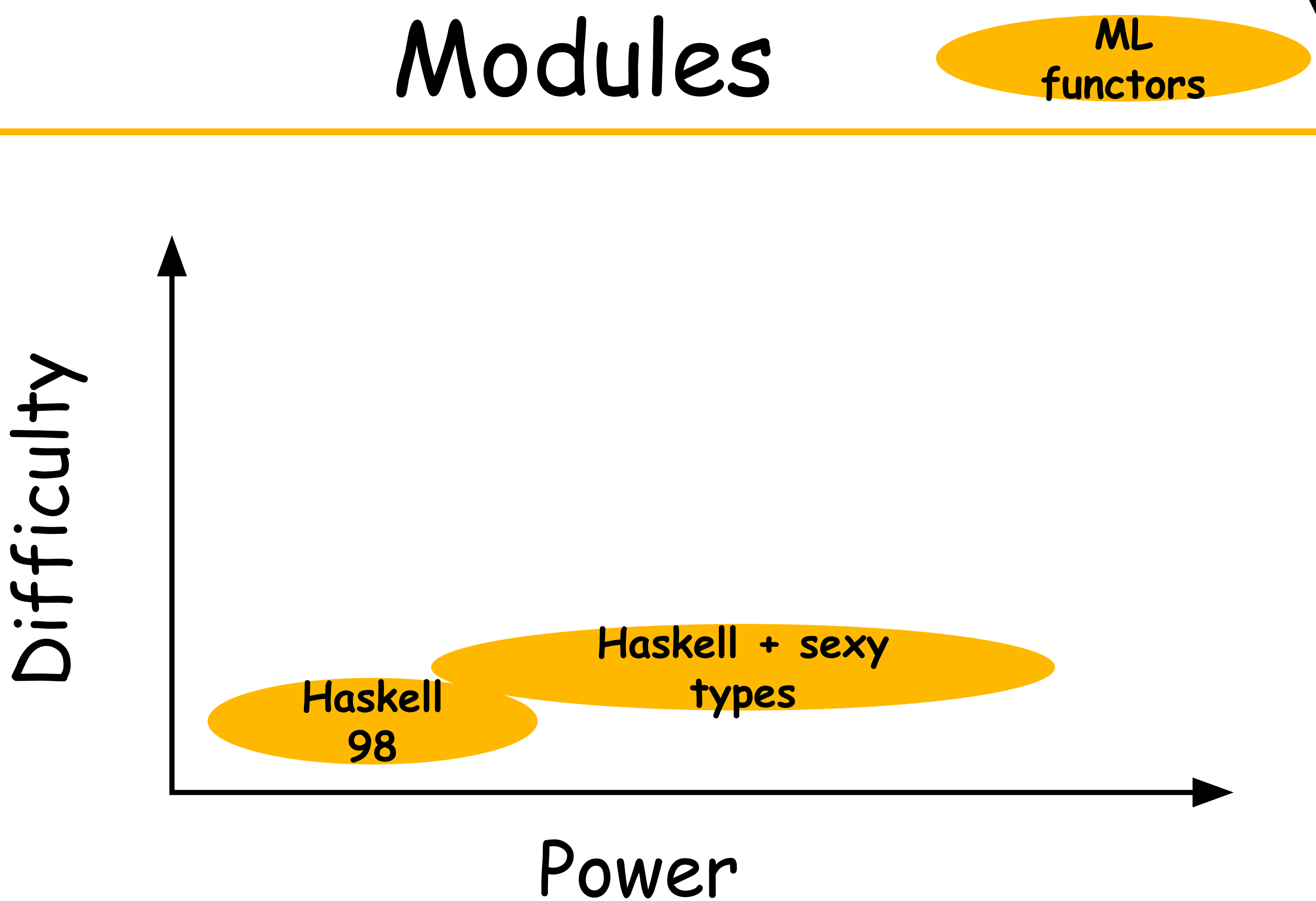
Destination = $F^{\omega}_{<:}$

Open question

What is a good design for user-level type annotation that exposes the power of F^{ω} or $F^{\omega}_{<:}$, but co-exists with type inference?

C.f. Didier & Didier's MLF work

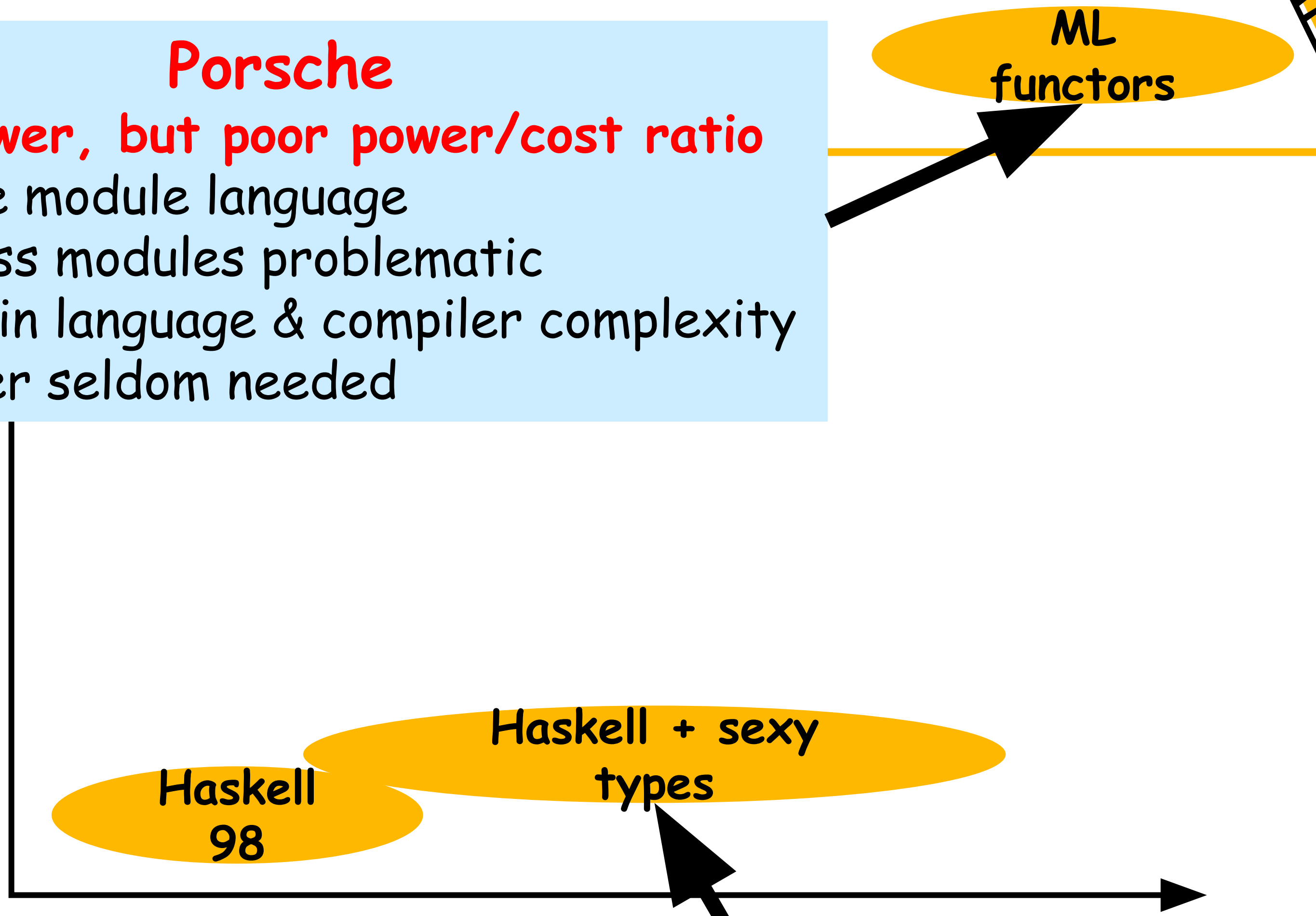
Modules


ML
functors
Haskell 98
Haskell + sexy types
Difficulty
Power

Porsche

High power, but poor power/cost ratio

- Separate module language
- First class modules problematic
- Big step in language & compiler complexity
- Full power seldom needed



Ford Cortina with alloy wheels

Medium power, with good power/cost

- Module parameterisation too weak
- No language support for module signatures

Modules

- Haskell has many features that overlap with what ML-style modules offer:
  - type classes
  - first class universals and existentials
- Does Haskell need functors anyway? No: one seldom needs to instantiate the same functor at different arguments
- But Haskell lacks a way to distribute "open" libraries, where the client provides some base modules; need module signatures and type-safe linking (e.g. PLT,Knit?). π not λ!
- **Wanted: a design with better power, but good power/weight.**

Encapsulating it all

```
data ST s a -- Abstract
newRef :: a -> ST s (STRef s a)
read   :: STRef s a -> ST s a
write  :: STRef s a -> a -> ST s ()
```

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

Stateful computation

Pure result

```
sort :: Ord a => [a] -> [a]
sort xs = runST (do { ..in-place sort.. })
```

Encapsulating it all

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

Higher rank type

Security of encapsulation depends on parametricity

Monads

And that depends on type classes to make non-parametric operations explicit (e.g. f :: Ord a => a -> a)

Parametricity depends on there being few polymorphic functions (e.g.. f:: a->a means f is the identity function or bottom)

And it also depends on purity (no side effects)

The Haskell committee

Arvind
Lennart Augustsson
Dave Barton
Brian Boutel
Warren Burton
Jon Fairbairn
Joseph Fasel
Andy Gordon
Maria Guzman
Kevin Hammond
Ralf Hinze
Paul Hudak [editor]
John Hughes [editor]

Thomas Johnsson
Mark Jones
Dick Kieburtz
John Launchbury
Erik Meijer
Rishiyur Nikhil
John Peterson
Simon Peyton Jones [editor]
Mike Reeve
Alastair Reid
Colin Runciman
Philip Wadler [editor]
David Wise
Jonathan Young