Little and not so big bits of Haskell

D. Renault

The Tuesday's Geekeries
ENSEIRB-MATMECA

July 2013, v. 1.1.1

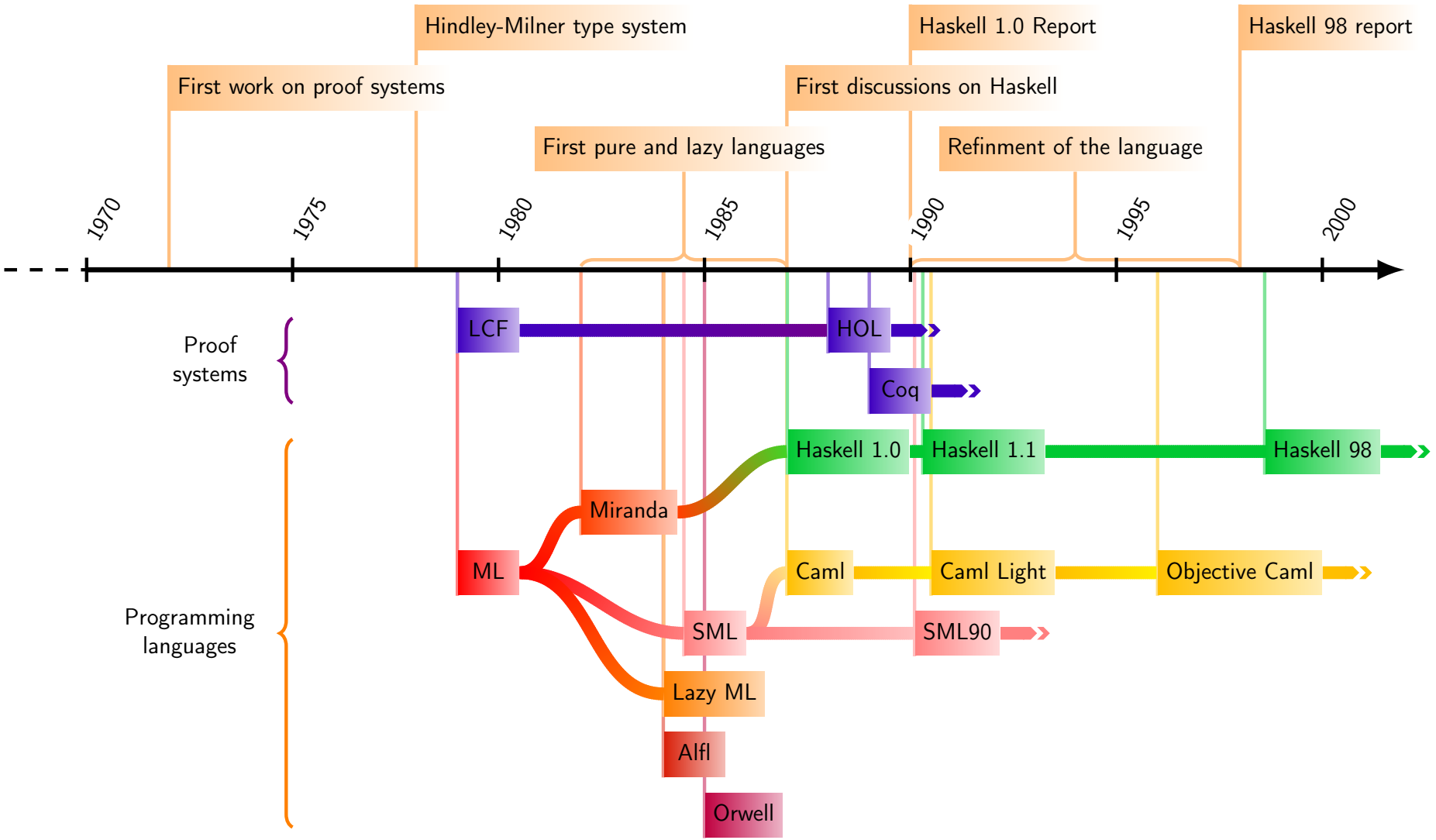
1990 - A committee formed by Simon **Peyton-Jones**, Paul **Hudak**, Philip **Wadler**, Ashton Kutcher, and People for the Ethical Treatment of Animals creates Haskell, a pure, non-strict, functional language.

Haskell gets some resistance due to the complexity of using monads to control side effects.

Wadler tries to appease critics by explaining that "a monad is a monoid in the category of endofunctors, what's the problem ?"

J.Iry, *"A Brief, Incomplete, and Mostly Wrong History of Programming Languages"*

A brief history of Haskell



A brief history of Haskell

- Essentially a **pure functional** language, meaning :
  declarative style, no side effects, no assignment, no sequence.
- With a **call-by-need** evaluation strategy (also named lazy evaluation).
- No formal definition, the current standard is the *Haskell 98 report* which gives an informal (albeit precise) definition of the language.
- Used for research in programming **new language features** (monads, type classes, zippers, lenses . . .)
- Now features an active community and several libraries in ongoing development (parallelism, web frameworks)

1. Quick introduction
2. The Haskell type system
3. Type classes
4. Lazy evaluation
5. The IO Monad
6. Haskell for programmers

Training wheels

Let us begin with the usual very simple examples :

```
1 + 2    ⟶ 3
6/3      ⟶ 2.0
sqrt 49  ⟶ 7.0
```

. . . or perhaps more traditionally :

```
putStrLn "Hello World" -- Prints "Hello World"
```

And it is quite possible to go some time † without realizing that even simple operations correspond to very complex representations under the hood.

†. Let's admit it, it would be quite a small amount of time, say 4 slides.

Examples of functions

Consider the definition of the factorial function :

```
fact x = if (x <= 1) then 1 else x * fact(x-1)
```

The same, but defined in an equational way :

```
fact 1 = 1
fact n = n * fact (n-1)
```

Another example with the greatest common divisor function :

```
gcd x y | y == 0     = x
        | y < 0      = gcd (-y) x
        | otherwise = gcd y (x `rem` y)
```

Playing with lists

In Haskell, lists are defined in the following way :

```
x = [9, 4, 3] -- a list of Ints, in short [Int]
```

On these you can do the usual stuff :

```
head x ---> 9
tail x ---> [4,3]
x ++ x ---> [9,4,3,9,4,3]
```

And also apply some classical functional programming :

```
map (+ 1) x  ---> [10,5,4]
filter odd x ---> [9,3]
fold (+) 0 x ---> 16 = 9+4+3, same as reduce in Lisp
```

Sorting lists

The quicksort algorithm in an elegant way :

```
qsort []     = []
qsort (x:xs) = qsort small ++ mid ++ qsort large
 where
    small = filter (< x) xs
    mid   = filter (==x) (x:xs)
    large = filter (> x) xs
```

. . . with a pattern-matching on a list parameter.

The Haskell Type system

Haskell has a strong type system, meaning that the system :

- determines the type of every expression statically (**type inference**) ;
- checks the coherence of the types of every expression.

Moreover, this type is accessible at the toplevel for inspection :

```
:t True
→ True :: Bool
:t "abcde"
→ "abcde" :: [Char] -- represents a list of Char
```

Even though some "routine" values have a rather strange type :

```
:t 3
→ 3 :: Num a ⇒ a
:t 2.0
→ 2.0 :: Fractional a ⇒ a
```

What is a type variable ?

Some types contain **type variables**, *i.e* placeholders that could be replaced by any concrete type.

For example, the function **head** returns the first element of a list :

```
:t head
→ head :: [a] → a -- a is a type variable
```

This function may be applied to a list containing any kind of values.
It is **generic** :

```
head [1,2,3]
→ 1
head "Hello"
→ 'H' -- recall that String = [Char]
```

Examples of genericity

Genericity appears quite naturally when dealing with abstract values.
The operator (.) corresponds to the composition of functions :

```
compose g h x = h(g(x)) -- (a → b) → (b → c) → a → c

(.)     g h x = g(h(x)) -- (b → c) → (a → b) → a → c
```

Usually, this operator is used to write concisely simple functions :

```
-- Compute the sum of the squares
-- of the odd integers < 10000
(sum . map square . filter odd) [1..10000]
```

Also, **map** and **fold** are examples of generic functions on lists.

```
:t map  ⟶ map  :: (a → b) → [a] → [b]
:t fold ⟶ fold :: (a → b → a) → a → [b] → a
```

Constrained genericity

The following function tests the existence of an element inside a list :

```
belongs y []     = False
belongs y (x:xs) = if (y == x) then True
                               else find y xs
```

What is the type of the belongs function ?

```
belongs :: Eq a ⇒ a → [a] → Boolean
```

This notation expresses the fact that the genericity is **constrained** :
the type variable a can be instantiated/replaced by a concrete type t if and only if t possesses the property **Eq** t.

Type classes

Type class

A type class **TC** is a set of types implementing a specification.
The fact that the concrete type t $\in$ **TC** is noted **TC** t.

For example, a possible definition for the **Eq** type class :

```
class Eq a where
  (==)          :: a → a → Bool
  (/=)          :: a → a → Bool
  x /= y        =  not (x == y) -- default definition
```

It is equivalent to the set of types possessing an equality function.

Type classes

Type class

A type class **TC** is a set of types implementing a specification.
The fact that the concrete type t $\in$ **TC** is noted **TC** t.

For a type to be in **Eq** t, it suffices to provide an implementation for (==) :

```
instance Eq Integer where
  x == y     =  specificIntegerEq x y
```

. . . where specificIntegerEq is an integer-customized equality function.
In practice, the (==) function is **overloaded** for all values typed in **Eq**.

This shares many similarities with Java interfaces, without the OO flavor.

Type classes - Examples

What kind of types belong to the **Eq** type class ?

```
      1 == 2          ⟶ False , a = Int
"Hello" == "Toto"     ⟶ False , a = [Char]
[1,2,3] == [1,2,3]    ⟶ True ,  a = [Int]
```

On the other hand, there exists types that do not belong to this type class, for example functions :

```
sin == cos ⟶ No instance for (Eq (a0 → a0))
             --   arising from a use of '=='
```

Comparison with OCaml

In OCaml, the belongs function looks like :

```
let rec belongs x l = match l with
  | []    → false
  | y::ys → (x=y) || (belongs x ys);;
```

The compiler claims that this function is completely generic.

```
val belongs : 'a → 'a list → bool
```

Nevertheless, in OCaml :

```
let _ = belongs sin [cos]
```

⇒ Compiles, but yields an exception at runtime.

```
(* Exception: Invalid_argument
   "equal: functional value". *)
```

Whereas in Haskell :

```
let _ = belongs sin [cos]
```

⇒ Yields a compilation error.

```
No instance for (Eq (a0 → a0))
arising from a use of 'belongs'
```

Type classes - Inheritance

It is possible to refine type classes, through **inheritance**.

In this example, the **Ord** type class inherits from the **Eq** type class :

```
class (Eq a) ⇒ Ord a where
  (<), (<=), (>=), (>)  :: a → a → Bool
  max, min              :: a → a → a
```

Considered as set of types, **Ord** $\subset$ **Eq**.

The language Haskell allows multiple inheritance.

Numerical types

Now it becomes possible to understand the types seen in the previous slides.

```
:t 3
→ 3 :: Num a ⇒ a
```

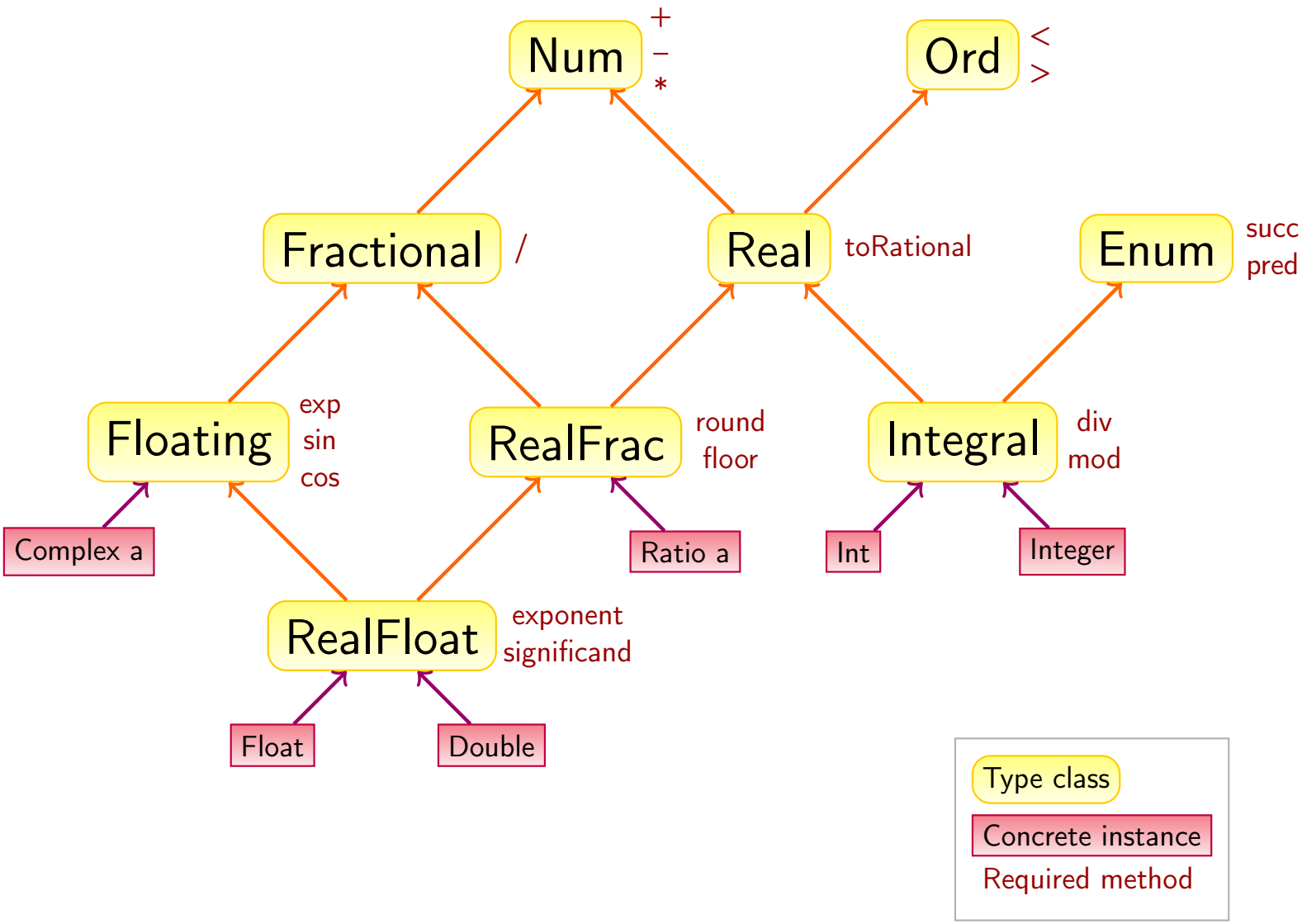
This means that the constant 3 has a type that is generic, but constrained by the **Num** type class (compare that to the behavior of C).

Precisely, the **Num** type class contains the following concrete types :

- **Int** and **Integer**, for the usual 32-bit and arbitrarily long integers,
- **Float** and **Double**, the single/double precision floating point numbers.

The **Num** type class requires the operations : $+$, $*$,$-$,**abs**,**signum**.

Haskell numerical types hierarchy



Type inference

And now comes the most interesting part :

```
3 + (7::Int)   ⟶ 10 :: Int
3 + (7::Float) ⟶ 10.0 :: Float
```

And both expressions are well-typed.

What the compiler does is analyze the expression and resolve all the constraints for all the types. This is called **type inference**.

- 3 :: **Num** a ⇒a
- 7 :: **Float**
- (+) :: **Num** a ⇒a →a →a

In this case, the only valid solution 3 + (7::**Float**) is a = **Float**.

In OCaml, operations for integers and floats are clearly separated :

```
3 + 7.0;;(* Error: This expression has type float *)
         (*        but was expected of type int   *)
(+)  : int → int → int
(+.) : float → float → float
```

... whereas in Haskell, the (+) function is **overloaded** and may be applied to different types.

```
(+) :: Num a ⇒ a → a → a
```

Nevertheless, this system still can prevent type errors such as :

```
(3::Int) + (7::Float)
-- Error : Couldn't match expected
-- type `Int' with actual type `Float'
```

Limits of type inference

However, type inference is not completely effective : it cannot always find a solution to the type constraints.

```
:t read ---> read :: Read a => String -> a
:t show ---> show :: Show a => a -> String
id x = show (read x) -- Ambiguous type variable
                     --'a0' in the constraints
```

The compiler is not able to determine the type of the expression **read** x.
In this example, **read** "4" could be typed with either **Int** or **Float**, but the type cannot be determined statically.

Lazy evaluation

Basic idea : some evaluations of expressions can terminate, even if their sub-expressions do not terminate.

```
fst (1, 1/0) ⟶ returns 1 despite the 1/0
[1..] !! 4   ⟶ returns 5 despite the infinite list
```

Call-by-need evaluation rule

An expression is evaluated only when necessary.
Typically, pattern matches and I/O force the evaluation of their arguments.

- The idea already exists in other languages such as C whose specification contains special rules for && and || .
- The if-then-else construct (which may or may not be an expression of the language) does not evaluate all its sub-expressions.

How does it work in practice ?

The process of evaluation of an expression usually reduces the expression to a normal form called a value.
Haskell authorizes the partial reduction of the sub-expressions (of a given expression) into pieces that are yet to be evaluated.

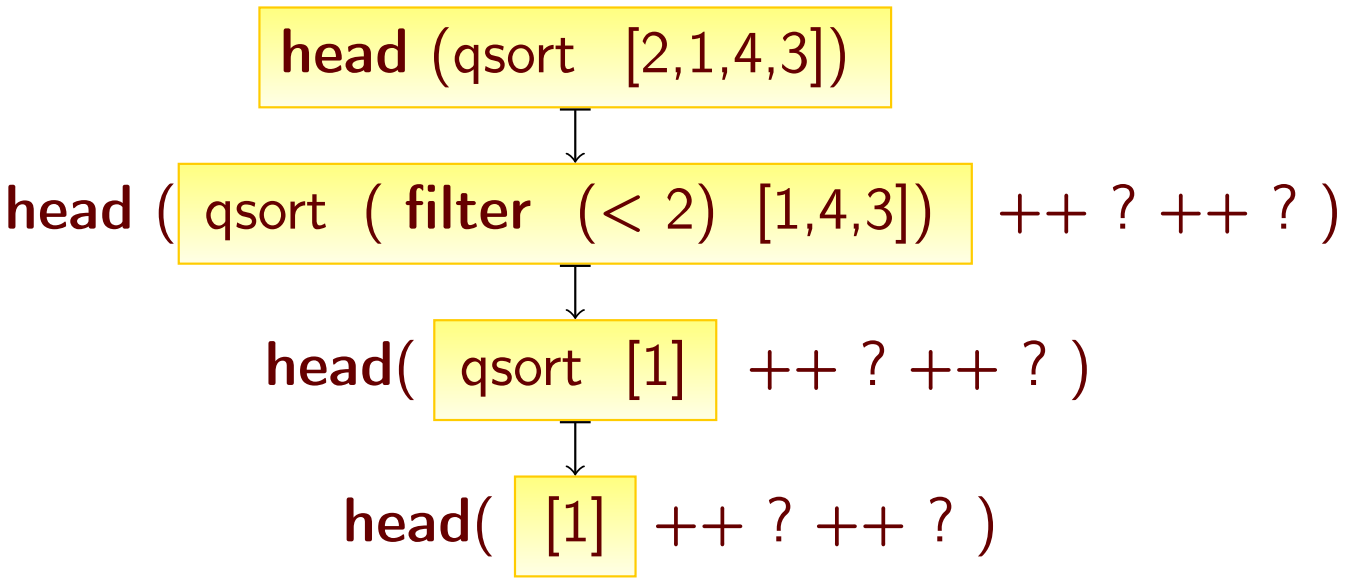
For example, reusing the quicksort algorithm :

```
head (qsort  [2,1,4,3])
```

How does it work in practice ?

The process of evaluation of an expression usually reduces the expression to a normal form called a **value**.
Haskell authorizes the partial reduction of the sub-expressions (of a given expression) into pieces that are yet to be evaluated.

For example, reusing the quicksort algorithm :



. . . returns 1 without evaluating the rest of the expression.

What kind of lazy computations ?

- Computations with infinite lists (ex. : Erathostenes sieve)

```
primes = 2 : 3 : ([5,7..] `minus`
    unionAll [[p*p, p*p+2*p..] | p ← tail primes])
```

- Streams and *in-place* computations (ex. : database access)

```
query  = do { x ← people -- possibly very long
            ; restrict (length x < 5)
            ; order [asc x!name] ; return x }
```

- Delayed computations
  (ex. : *copy-on-write* strategy for page allocation in virtual memory)
- Elegant writing of unbounded computations

```
msum $ repeat getValidPassword
-- equivalent to while loop
```

Limits of call-by-need semantics

- Slight memory overhead for storing expressions instead of values ;
- Important **memory leaks** may occur when rewriting expressions without fully evaluating them. For instance :

  foldl (+) 0 [1..1000000:: **Integer**] $\rightarrow$ ((((1+2)+3)+4)+$\cdots$

  Expressions are reduced only when they are actually needed.
  Solution : replace **foldl** by a strict version **foldl'**
- The garbage collection overhead due to leaks incurs a performance hit.

Most of the time, the optimization of a Haskell program consists in balancing strict and lazy evaluation.

The GHC compiler makes an optimization pass called **strictness analysis**.

C++ Pop quizz

Can you guess the output of the following code ?

```
#include <iostream>
using namespace std;
int main(void){
  int i = 0;
  cout << i++ << i++ << i++ << i++ << endl;
  i = 0;
  cout << ++i << ++i << ++i << ++i << endl;
  return 0;
}
```

C++ Pop quizz

Can you guess the output of the following code ?

```
#include <iostream>
using namespace std;
int main(void){
  int i = 0;
  cout << i++ << i++ << i++ << i++ << endl;
  i = 0;
  cout << ++i << ++i << ++i << ++i << endl;
  return 0;
}
```

With −O0 :
3210
4321

With −O2 :
3210
4444

How is it possible to do I/O in a pure functional language ?
$\rightarrow$ since I/O actions all contain side effects, this is really a problem.

What is the important property here ?

Referential transparency

The value of an expression must be independent of the moment of its evaluation.

Example : in the last example, i++ was not referentially transparent.

This property is *fundamental* with a call-by-need evaluation strategy.

How do you make an action such as **getChar** behave in a referentially transparent manner ?

The IO Monad

Consider the following C program :

```c
int main(void) {
  int i,j,s;
  scanf("%d", &i);    // read i
  scanf("%d", &j);    // read j
  s = i+j;
  printf("%d\n", s); // print the sum
  return 0;
}
```

- Composed of 3 side-effects : 2 inputs, 1 output.
- Highly reliant on imperative features.

Let us transform this example in the Haskell world, called the IO **monad**.

The IO Monad

We obtain the following Haskell program :

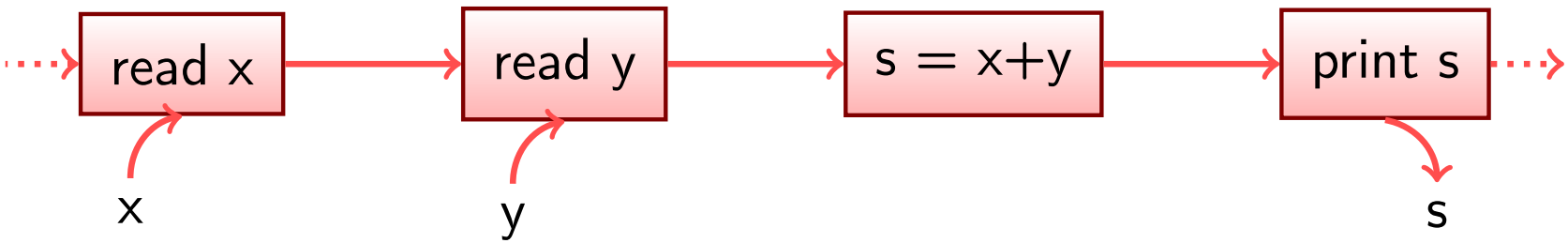
```c
int main(void) {
  int i,j,s;
  scanf("%d", &i);
  scanf("%d", &j);
  s = i+j;
  printf("%d\n", s);
  return 0;
}
```

```haskell
main = do
     x ← getLine
     y ← getLine
     let s = x+y
     putStr (show s)
```

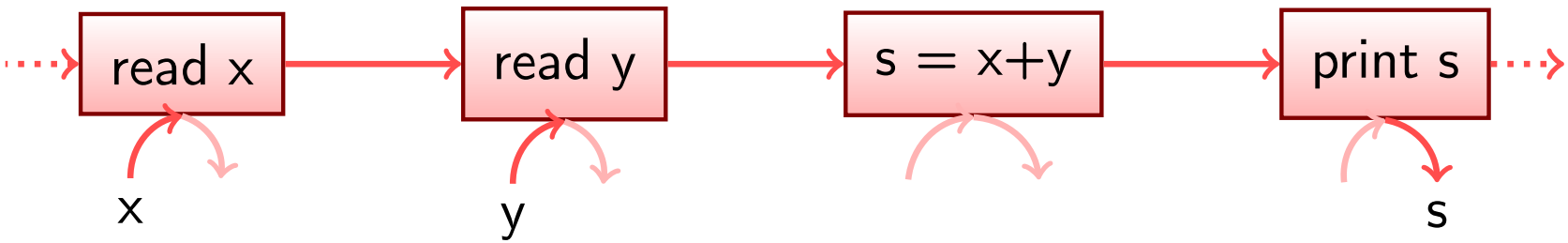
. . . in a very straightforward manner†.

†. That will take the 7 next slides to explain.

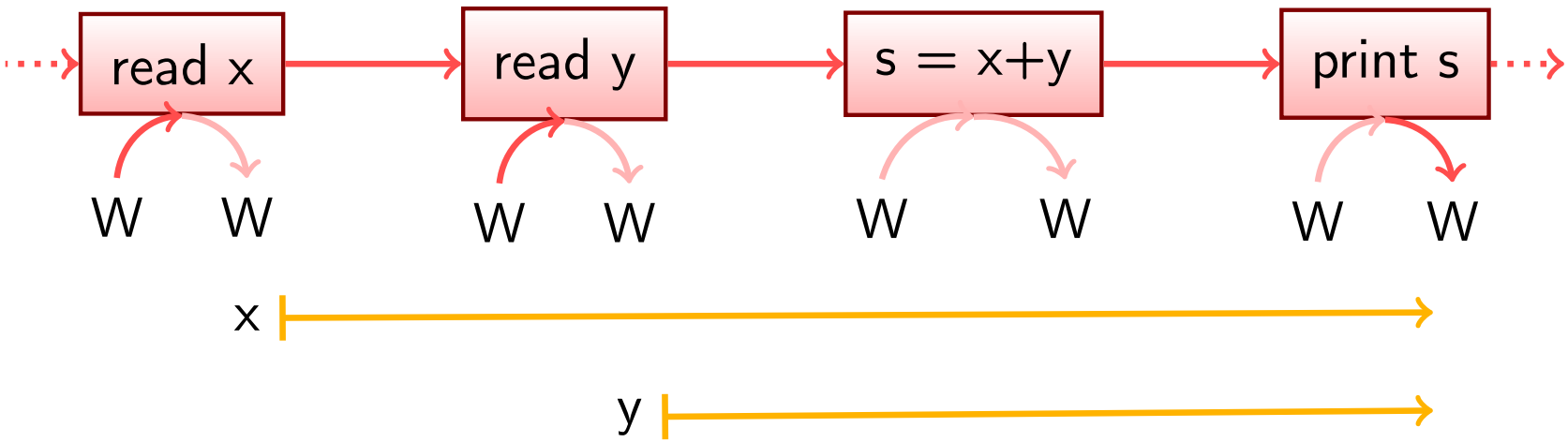
Decompose (sequentially) the operations performed in this example :



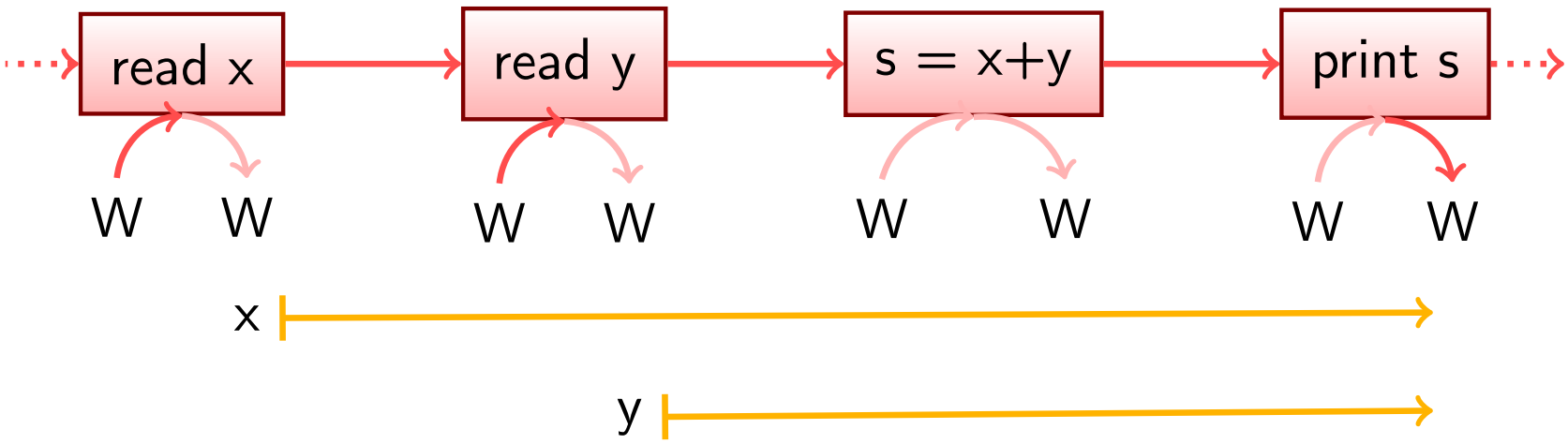
Decompose (sequentially) the operations performed in this example :



Decompose (sequentially) the operations performed in this example :



Decompose (sequentially) the operations performed in this example :



Consider the following type : **type IO** a = World → (World, a)

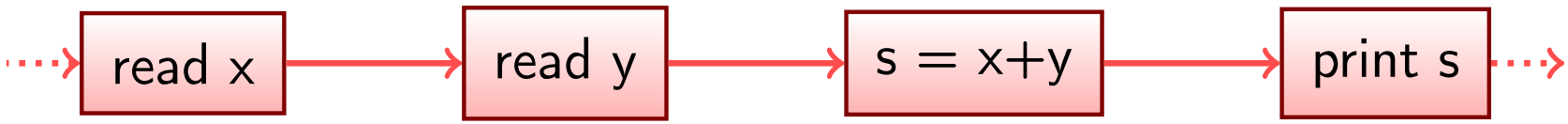
Namely : a function taking **some world**, returning { **a new world** and / a value of type a.

Example : the World contains handles attached to the stdin and stdout.

In this framework :

- **getLine** :: **IO String** $\equiv$ World $\rightarrow$ (World, **String**)
- **putStr** :: **String** $\rightarrow$**IO** () $\equiv$ **String** $\rightarrow$World $\rightarrow$(World, ())

In a functional language, it becomes natural to compose these actions :




```
main w0 =
    let (x, w1)  = getLine w0 in
    let (y, w2)  = getLine w1 in
    let (s, w3)  = (x+y, w2)  in
    let ((), w4) = putStr (show s) w3
```

```
main w0 =
    let (x, w1) = getLine w0 in
    let (y, w2) = getLine w1 in
    let (s, w3) = (x+y, w2) in
    let ((), w4) = putStr (show s) w3
```

Let us rewrite this code using a composition of functions :

```
main w0 =
 (\(x,w1) →
            (\(y,w2) →
                       (\(s,w3) → putStr (show s) w3 )
                        (x+y,w2)
            ( getLine w1 ))
 ( getLine w0 )
```

```
main w0 =
  (\(x,w1) ->
              (\(y,w2) ->
                          (\(s,w3) -> putStr (show s) w3 )
                           (x+y,w2)
              ( getLine w1 ))
  ( getLine w0 )
```

Rewrite and reorder the function application using an infix operator `>>=` :

```
main w0 =
  getLine w0 >>= (\(x,w1) ->
                  getLine w1 >>= (\(y,w2) ->
                                  (x+y,w2) >>= (\(s,w3) ->
                                                putStr (show s) w3 )))
```

```
main w0 =
    getLine w0 >>= (\(x,w1) ->
        getLine w1 >>= (\(y,w2) ->
            (x+y,w2)) >>= (\(s,w3) ->
                putStr (show s) w3)))
```

Now consider that the "w_i" variables are implicitely transmitted :

```
main =
    getLine >>= (\x ->
        getLine >>= (\y ->
            return (x+y) >>= (\s ->
                putStr (show s))))
```

Notice the apparition of the **return** function.

```
main =
    getLine >>= (\x ->
        getLine >>= (\y ->
            return (x+y) >>= (\s ->
                putStr (show s))))
```

Finally, rewrite this using a bit of syntactic sugar :

```
main = do
    x <- getLine
    y <- getLine
    let s = x+y
    putStr (show s)
```

Final comparison

C code (imperative) :

```c
int main(void) {
  int i,j,s;
  scanf("%d", &i);
  scanf("%d", &j);
  s = i+j;
  printf("%d\n", s);
  return 0;    }
```

Haskell code (functional) :

```haskell
main = do
    x ← getLine
    y ← getLine
    let s = (read x)+(read y) -- convert to ints
    putStr (show s)
```

So what ?

- **Pipelined computation** encapsulating a **global state** : the monad imposes a sequence of actions with a transmission of the state of the world, with no direct access, no possibility of reference or duplication.
- Clear **type separation** : all IO-dependent computations must be done inside the IO monad (and are tagged with the **IO** type), all pure (*i.e* IO-independent) computations can be done outside.
- **Referential transparency** : the computation done within the IO monad is dependent of an exterior World variable. In this regard, it is referentially transparent.

But :

- Difficulty to write in a monadic style : the construction possesses a **complex semantics**, and the type errors rapidly become pretty scary.

Monads in the category of endofunctors

Monads represent a generic data type for arranging the composition of operations. For example :

- on option types : → error handling

```
newUser login pass = do
    validateUsrString login  ---> Just String  || ∅
    validatePwdString pass   ---> Just String  || ∅
    hash <- computeHash pass ---> Just Hash    || ∅
    insertDB login hash      ---> Just ()      || ∅
```

- on lists : → list comprehensions

```
dubiousList = do
  x <- [1,2,3]; y <- [4,5]
  let s = x+y                 -- s = [5,6,6,7,7,8]
  guard (even s)              -- s = [6,6,8]
  return s                    -- return s
```

Haskell in the Real World™

Some links for beginners in the language :

- Hackage, a **package library**
  Hackage : `http://hackage.haskell.org`
  Features around 5000 packages and 450000 functions.
- Several **documentation** search engines
  Hayoo : `http://holumbus.fh-wedel.de/hayoo/hayoo.html`
  Hoogle : `http://www.haskell.org/hoogle`
- Cabal, a **build system** for Haskell
  `http://www.haskell.org/cabal`
- The Haskell (semestrial) community report
  `http://www.haskell.org/communities`
- Live open source projects (darcs, pandoc, xmonad, ...)

Haskell and parallel algorithms

Eden library (http://www.mathematik.uni-marburg.de/~eden)

Example : approximation of π by a midpoint-integration scheme :

```
ncpi n = mapRedr (+) 0 (f . index) [1..n]
                 / fromInteger n
    where f x = 4 / (1 + x*x)
          index i = (fromInteger i - 0.5)
                 / fromInteger n
```

Replace mapRedr by offlineParMapRedr to get a parallel algorithm.

Provides strategies for Map-Reduce and Divide-and-Conquer algorithms.
Contains helpers simplifying the communications between parallel processes.

Haskell and the web

Yesod framework (http://www.yesodweb.com)

- Type-checked DSLs for templating and routing :

```
<div .message>
$maybe (info,msg) <- submission
    File type : <em> #fileContentType info </em>.
    Message   : <em> #msg </em>
  <form method=post action= @{HomeR}#form
        enctype= #{formEnctype} >
   ^formWidget
  <input type="submit" value="Send it !">
```

Snap framework (http://snapframework.com)

- Modular system called *snaplets* separating concerns (authentication, authorization, database ...);

Good reading

- Miran Lipovača : *Learn you a Haskell for great good !*
  http://learnyouahaskell.com
- B. O'Sullivan, D. Stewart, and J. Goerzen : *Real World Haskell*
  http://book.realworldhaskell.org
- The Haskell wiki
  http://www.haskell.org/haskellwiki/Category:Haskell