# MGS 2011: FUN Lecture 1

*Lazy Functional Programming*

Henrik Nilsson

University of Nottingham, UK



## What Is a Functional Language? (1)

- ***Imperative Languages***:
  - Implicit state.
  - Computation essentially a sequence of side-effecting actions.
- ***Declarative Languages*** (Lloyd 1994):
  - ***No*** implicit state.
  - A program can be regarded as a theory.
  - Computation can be seen as deduction from this theory.
  - Examples: Logic and Functional Languages.



## What Is a Functional Language? (2)

Another perspective:

- ***Algorithm = Logic + Control***
- Declarative programming emphasises the logic ("what") rather than the control ("how").
- Examples:
  - Resolution (logic programming)
  - Lazy evaluation (found in some functional and logic languages)



## What Is a Functional Language? (3)

Declarative languages for practical use tend to be only ***weakly declarative***; i.e., not totally free of control aspects. For example:

- Equations in functional languages are directed.
- Order of patterns often matters for pattern matching.
- Constructs for taking control over the order of evaluation. (E.g. `cut` in Prolog, `seq` in Haskell.)

## What Is a Functional Language? (4)

Exactly what constitute a functional language is somewhat contentious.

Pragmatically, a functional language is one that encourages a mostly declarative, ***functional style*** of programming.

Typical features/characteristics:

- Functions are first-class entities.
- Computation expressed through function application.
- Recursive (and co-recursive) definitions.



## What Is a Functional Language? (5)

This "definition" covers both:

- ***Pure*** functional languages: no side effects
  - (Weakly) declarative: equational reasoning valid (with care); ***referentially transparent***.
  - Example: Haskell
- ***Mostly*** functional languages: some side effects, e.g. for I/O.
  - Equational reasoning with care.
  - Examples: ML, OCaml, Scheme, Erlang

(Real purists would point out that non-termination is also a side effect.)



## This and the Following Lectures

- In this and the following lectures we will explore ***Purely Functional Programming*** through the use of ***Haskell***.
- Theme of today: ***Relinquishing control: exploiting lazy evaluation***

Will assume some familiarity with functional programming in a language like Haskell or ML. Will explain Haskell syntax and other points as needed: ***Just ask!***



## Evaluation Orders (1)

Consider:

```
sqr x = x * x
dbl x = x + x
main = sqr (dbl (2 + 3))
```

Many possible reduction orders. Innermost, leftmost ***redex*** first is called ***Applicative Order Reduction*** (AOR):

```
main ⇒ sqr (dbl (2 + 3)) ⇒ sqr (dbl 5)
⇒ sqr (5 + 5) ⇒ sqr 10 ⇒ 10 * 10 ⇒ 100
```

This is just ***Call-By-Value***.

## Evaluation Orders (2)

Outermost, leftmost redex first is called ***Normal Order Reduction*** (NOR):

```
main ⇒ sqr (dbl (2 + 3))
⇒ dbl (2 + 3) * dbl (2 + 3)
⇒ ((2 + 3) + (2 + 3)) * dbl (2 + 3)
⇒ (5 + (2 + 3)) * dbl (2 + 3)
⇒ (5 + 5) * dbl (2 + 3) ⇒ 10 * dbl (2 + 3)
⇒ ... ⇒ 10 * 10 ⇒ 100
```

(Applications of arithmetic operations only considered redexes once arguments are numbers.)
Demand-driven evaluation or ***Call-By-Need***



## Why Normal Order Reduction? (1)

NOR seems rather inefficient. Any use?

- Best possible termination properties. Two important theorems from the $\lambda$-calculus:
  - Church-Rosser Theorem I:
    No term has more than one normal form.
  - Church-Rosser Theorem II:
    If a term has a normal form, then NOR will find it.



## Why Normal Order Reduction? (2)

- More expressive power; e.g.:
  - "Infinite" data structures
  - Circular programming
- More declarative code as control aspects (order of evaluation) left implicit.



## Strict vs. Non-strict Semantics (1)

- $\perp$, or "bottom", the ***undefined value***, representing ***errors*** and ***non-termination***.
- A function $f$ is ***strict*** iff:

$$f\ \perp = \perp$$

For example, $+$ is strict in both its arguments:

$$(0/0) + 1 = \perp + 1 = \perp$$
$$1 + (0/0) = 1 + \perp = \perp$$

## Strict vs. Non-strict Semantics (2)

Consider:

```
foo x = 1
```

What is the value of `foo (0/0)`?

- AOR: `foo (0/0)` $\Rightarrow \bot$
  Conceptually, `foo` $\bot = \bot$; i.e., `foo` is strict.
- NOR: `foo (0/0)` $\Rightarrow$ `1`
  Conceptually, `foo` $\bot = 1$; i.e., `foo` is non-strict.

Thus, NOR results in non-strict semantics.
Note: NOR gave well-defined result, AOR did not.



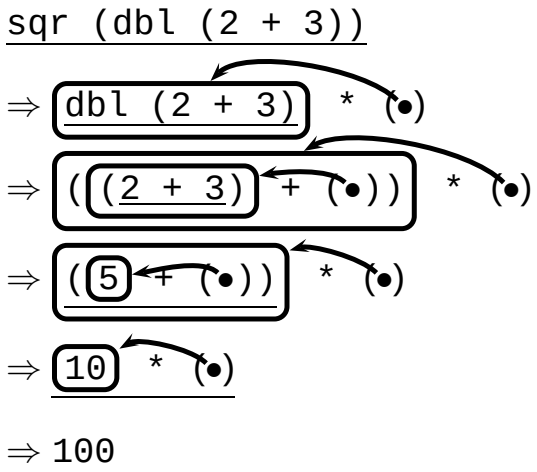
## Lazy Evaluation (1)

Lazy evaluation is an ***technique for implementing NOR*** more efficiently:

- An expression is evaluated ***only if needed***.
- ***Sharing*** employed to ensure any one expression evaluated at most once.



## Lazy Evaluation (2)






## Infinite Data Structures (1)

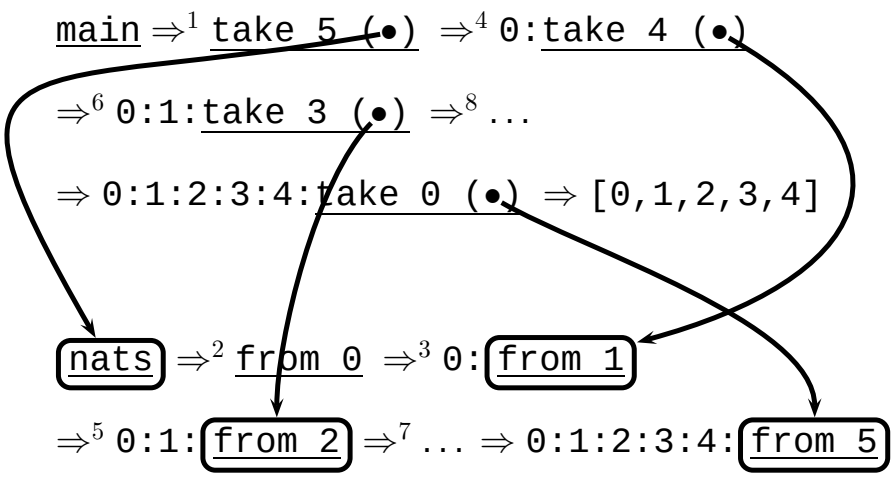
```
take 0 xs     = []
take n []     = []
take n (x:xs) = x : take (n-1) xs

from n = n : from (n+1)

nats = from 0

main = take 5 nats
```

## Infinite Data Structures (2)






## Circular Data Structures (2)

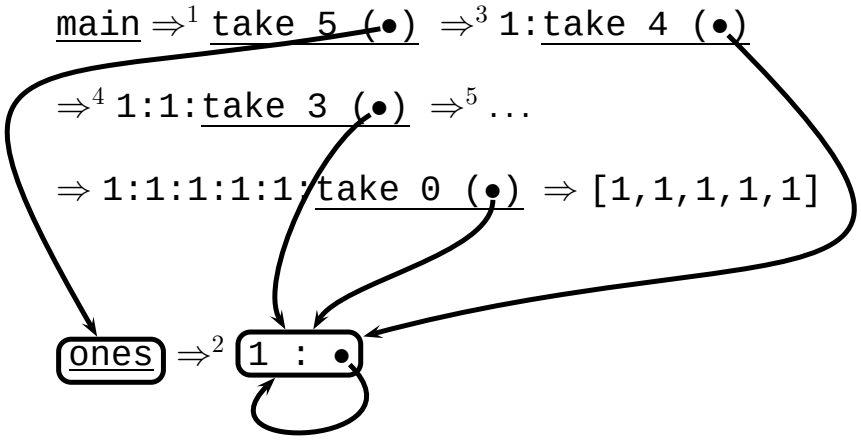
```
take 0 xs     = []
take n []     = []
take n (x:xs) = x : take (n-1) xs

ones = 1 : ones

main = take 5 ones
```



## Circular Data Structures (2)






## Circular Programming (1)

A non-empty tree type:

```
data Tree = Leaf Int | Node Tree Tree
```

Suppose we would like to write a function that replaces each leaf integer in a given tree with the ***smallest*** integer in that tree.

How many passes over the tree are needed?

***One!***

## Circular Programming (2)

Write a function that replaces all leaf integers by a given integer, and returns the new tree along with the smallest integer of the given tree:

```
fmr :: Int -> Tree -> (Tree, Int)
fmr m (Leaf i) = (Leaf m, i)
fmr m (Node tl tr) =
    (Node tl' tr', min ml mr)
    where
        (tl', ml) = fmr m tl
        (tr', mr) = fmr m tr
```



## Circular Programming (3)

For a given tree `t`, the desired tree is now obtained as the ***solution*** to the equation:

```
(t', m) = fmr m t
```

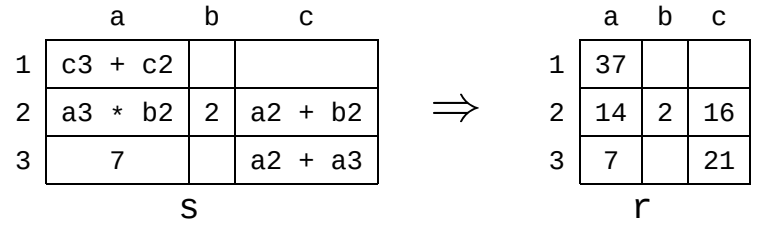
Thus:

```
findMinReplace t = t'
    where
        (t', m) = fmr m t
```

Intuitively, this works because `fmr` can compute its result without needing to know the ***value*** of `m`.



## A Simple Spreadsheet Evaluator



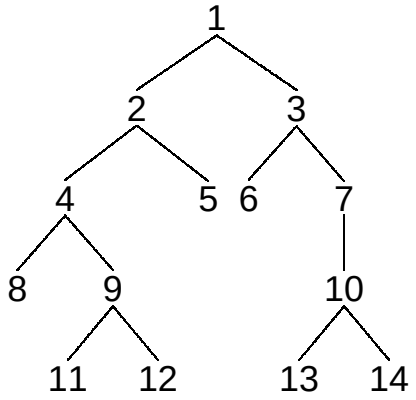

```
r = array (bounds s)
          [ ((i,j), eval r (s!(i,j)))
          | (i,j) <- indices s ]
```

The evaluated sheet is again simply the ***solution*** to the stated equation. No need to worry about evaluation order. Any caveats?



## Breadth-first Numbering (1)

Consider the problem of numbering a possibly infinitely deep tree in breadth-first order:



## Breadth-first Numbering (2)

The following algorithm is due to G. Jones and J. Gibbons (1992), but the presentation differs.

Consider the following tree type:

```
data Tree a = Empty
            | Node (Tree a) a (Tree a)
```

Define:

| | |
|---|---|
| $\text{width}\ t\ i$ | The width of a tree $t$ at level $i$ (0 origin). |
| $\text{label}\ t\ i\ j$ | The $j$th label at level $i$ of a tree $t$ (0 origin). |



## Breadth-first Numbering (3)

The following system of equations defines breadth-first numbering:

$$\begin{aligned} \text{label}\ t\ 0\ 0 &= 1 && (1)\\ \text{label}\ t\ (i+1)\ 0 &= \text{label}\ t\ i\ 0 + \text{width}\ t\ i && (2)\\ \text{label}\ t\ i\ (j+1) &= \text{label}\ t\ i\ j + 1 && (3) \end{aligned}$$

Note that $\text{label}\ t\ i\ 0$ is defined for ***all*** levels $i$ (as long as the widths of all tree levels are finite).



## Breadth-first Numbering (4)

The code that follows sets up the defining system of equations:

- ***Streams*** (infinite lists) of labels are used as a ***mediating data structure*** to allow equations to be set up between adjacent nodes within levels and between the last node at one level and the first node at the next.
- Idea: the tree numbering function for a subtree takes a stream of labels for the ***first node*** at each level, and returns a stream of labels for the ***node after the last node*** at each level.



## Breadth-first Numbering (5)

- As there manifestly are ***no cyclic dependences*** among the equations, we can entrust the details of solving them to the lazy evaluation machinery in the safe knowledge that a solution will be found.

## Breadth-first Numbering (6)

```
bfn :: Tree a -> Tree Integer
bfn t = t'
    where
        (ns, t') = bfnAux (1 : ns) t

bfnAux :: [Integer] -> Tree a
          -> ([Integer], Tree Integer)
bfnAux ns       Empty          = (ns, Empty)
bfnAux (n : ns) (Node tl _ tr) = ((n + 1) : ns'',
                                  Node tl' n tr')
    where
        (ns',  tl') = bfnAux ns tl
        (ns'', tr') = bfnAux ns' tr
```

*Eqns (1) & (2)*

*Eqn (3)*



## Dynamic Programming

***Dynamic Programming***:

- Create a ***table*** of all subproblems that ever will have to be solved.
- Fill in table without regard to whether the solution to that particular subproblem will be needed.
- Combine solutions to form overall solution.

***Lazy Evaluation*** is a perfect match as saves us from having to worry about finding a suitable evaluation order.



## The Triangulation Problem (1)

Select a set of ***chords*** that divides a convex polygon into triangles such that:

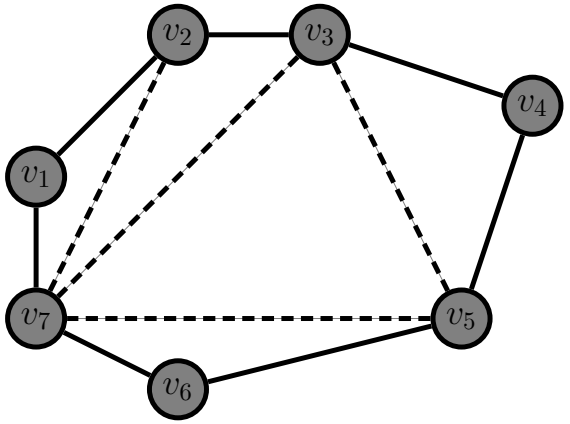
- no two chords cross each other
- the sum of their length is minimal.

We will only consider computing the minimal length.

See Aho, Hopcroft, Ullman (1983) for details.



## The Triangulation Problem (2)

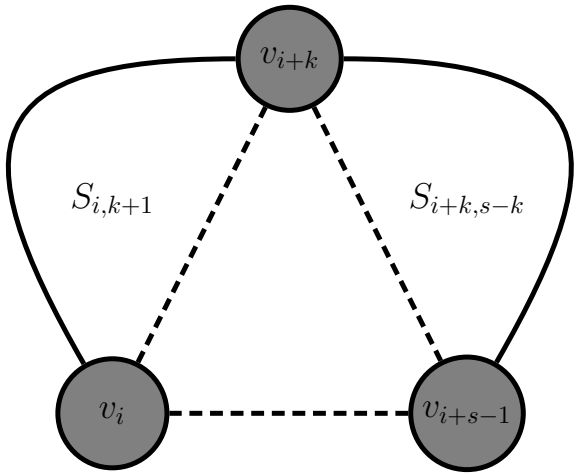


## The Triangulation Problem (3)

- Let $S_{is}$ denote the subproblem of size $s$ starting at vertex $v_i$ of finding the minimum triangulation of the polygon $v_i$, $v_{i+1}$, ..., $v_{i+s-1}$ (counting modulo the number of vertices).
- Subproblems of size less than 4 are trivial.
- Solving $S_{is}$ is done by solving $S_{i,k+1}$ and $S_{i+k,s-k}$ for all $k$, $1 \leq k \leq s-2$
- The obvious recursive formulation results in $3^{s-4}$ (non-trivial) calls.
- But for $n \geq 4$ vertices there are only $n(n-3)$ non-trivial subproblems!



## The Triangulation Problem (4)






## The Triangulation Problem (5)

- Let $C_{is}$ denote the minimal triangulation cost of $S_{is}$.
- Let $D(v_p, v_q)$ denote the length of a chord between $v_p$ and $v_q$ (length is 0 for non-chords; i.e. adjacent $v_p$ and $v_q$).
- For $s \geq 4$:

$$C_{is} = \min_{k \in [1,s-2]} \left\{ \begin{array}{l} C_{i,k+1} + C_{i+k,s-k} \\ +D(v_i, v_{i+k}) + D(v_{i+k}, v_{i+s-1}) \end{array} \right\}$$

- For $s < 4$, $S_{is} = 0$.



## The Triangulation Problem (6)

These equations can be transliterated straight into Haskell:

```
triCost :: Polygon -> Double
triCost p = cost!(0,n) where
    cost = array ((0,0), (n-1,n))
                 ([ ((i,s),
                     minimum [ cost!(i, k+1)
                               + cost!((i+k) `mod` n, s-k)
                               + dist p i ((i+k) `mod` n)
                               + dist p ((i+k) `mod` n)
                                        ((i+s-1) `mod` n)
                             | k <- [1..s-2] ])
                  | i <- [0..n-1], s <- [4..n] ] ++
                  [ ((i,s), 0.0)
                  | i <- [0..n-1], s <- [0..3] ])
    n = snd (bounds b) + 1
```

## Attribute Grammars (1)

Lazy evaluation is also very useful for evaluation of ***Attribute Grammars***:

- The attribution function is defined recursively over the tree:
  - takes inherited attributes as extra arguments;
  - returns a tuple of all synthesised attributes.
- As long as there exists ***some*** possible attribution order, lazy evaluation will take care of the attribute evaluation.



## Attribute Grammars (2)

- The earlier examples on Circular Programming and Breadth-first Numbering can be seen as instances of this idea.



## Reading


- John W. Lloyd. Practical advantages of declarative programming. In *Joint Conference on Declarative Programming, GULP-PRODE'94*, 1994.
- John Hughes. Why Functional Programming Matters. *The Computer Journal*, 32(2):98–197, April 1989.
- Thomas Johnsson. Attribute Grammars as a Functional Programming Paradigm. In *Functional Programming Languages and Computer Architecture, FPCA'87*, 1987




## Reading


- Geraint Jones and Jeremy Gibbons. *Linear-time breadth-first tree algorithms: An exercise in the arithmetic of folds and zips.* Technical Report TR-31-92, Oxford University Computing Laboratory, 1992.
- Alfred Aho, John Hopcroft, Jeffrey Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

# MGS 2011: FUN Lecture 2

*Purely Functional Data Structures*

Henrik Nilsson

University of Nottingham, UK



## Purely Functional Data structures (1)

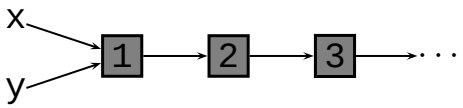
Why is there a need to consider purely functional data structures?

- The standard implementations of many data structures assume imperative update. To what extent truly necessary?
- Purely functional data structures are ***persistent***, while imperative ones are ***ephemeral***:
  - Persistence is a useful property in its own right.
  - Can't expect added benefits for free.

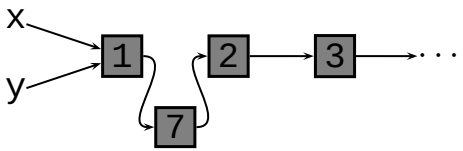


## Purely Functional Data structures (2)

Linked list:



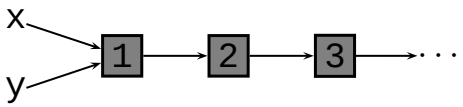

After insert, if ephemeral:



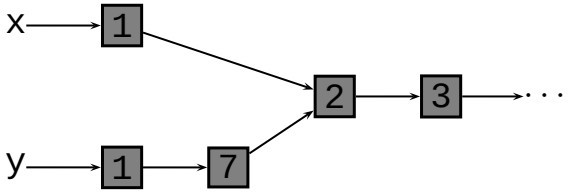



## Purely Functional Data structures (3)

Linked list:




After insert, if persistent:



## Purely Functional Data structures (4)

This lecture draws from:

> Chris Okasaki. *Purely Functional Data Structures*. Cambridge University Press, 1998.

We will look at some examples of how ***numerical representations*** can be used to derive purely functional data structures.



## Numerical Representations (1)

Strong analogy between lists and the usual representation of natural numbers:

```
data List a =                    data Nat =
      Nil                              Zero
    | Cons a (List a)                | Succ Nat

tail (Cons _ xs) = xs            pred (Succ n) = n

append Nil         ys = ys       plus Zero n     = n
append (Cons x xs) ys =          plus (Succ m) n =
    Cons x (append xs ys)            Succ (plus m n)
```



## Numerical Representations (2)

This analogy can be taken further for designing container structures because:

- inserting an element resembles incrementing a number
- combining two containers resembles adding two numbers

etc.

Thus, representations of natural numbers with certain properties induce container types with similar properties. Called ***Numerical Representations***.



## Random Access Lists

We will consider ***Random Access Lists*** in the following. Signature:

```
data RList a

empty   :: RList a
isEmpty :: RList a -> Bool
cons    :: a -> RList a -> RList a
head    :: RList a -> a
tail    :: RList a -> RList a
lookup  :: Int -> RList a -> a
update  :: Int -> a -> RList a -> RList
```

## Positional Number Systems (1)

- A number is written as a ***sequence*** of ***digits*** $b_0 b_1 \ldots b_{m-1}$, where $b_i \in D_i$ for a fixed family of digit sets given by the positional system.
- $b_0$ is the ***least significant*** digit, $b_{m-1}$ the ***most significant*** digit (note the ordering).
- Each digit $b_i$ has a ***weight*** $w_i$. Thus:

$$\text{value}(b_0 b_1 \ldots b_{m-1}) = \sum_{0}^{m-1} b_i w_i$$

where the fixed sequence of weights $w_i$ is given by the positional system.



## Positional Number Systems (2)

- A number is written written in ***base*** $B$ if $w_i = B^i$ and $D_i = \{0, \ldots, B-1\}$.
- The sequence $w_i$ is usually, but not necessarily, increasing.
- A number system is ***redundant*** if there is more than one way to represent some numbers (disallowing trailing zeroes).
- A representation of a positional number system can be ***dense***, meaning including zeroes, or ***sparse***, eliding zeroes.



## Exercise 1: Positional Number Systems

Suppose $w_i = 2^i$ and $D_i = \{0, 1, 2\}$. Give three different ways to represent 17.



## Exercise 1: Solution

- 10001, since $\text{value}(10001) = 1 \cdot 2^0 + 1 \cdot 2^4$
- 1002, since $\text{value}(1002) = 1 \cdot 2^0 + 2 \cdot 2^3$
- 1021, since $\text{value}(1021) = 1 \cdot 2^0 + 2 \cdot 2^2 + 1 \cdot 2^3$
- 1211, since $\text{value}(1211) = 1 \cdot 2^0 + 2 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3$

## From Positional System to Container

Given a positional system, a numerical representation may be derived as follows:

- for a container of size $n$, consider a representation $b_0 b_1 \ldots b_{m-1}$ of $n$,
- represent the collection of $n$ elements by a ***sequence of trees*** of size $w_i$ such that there are $b_i$ trees of that size.

For example, given the positional system of exercise 1, a container of size 17 might be represented by 1 tree of size 1, 2 trees of size 2, 1 tree of size 4, and 1 tree of size 8.



## What Kind of Trees?

The kind of tree should be chosen depending on needed sizes and properties. Two possibilities:

- ***Complete Binary Leaf Trees***

```
data Tree a = Leaf a
            | Node (Tree a) (Tree a)
```

  Sizes: $2^n, n \geq 0$

- ***Complete Binary Trees***

```
data Tree a = Leaf a
            | Node (Tree a) a (Tree a)
```

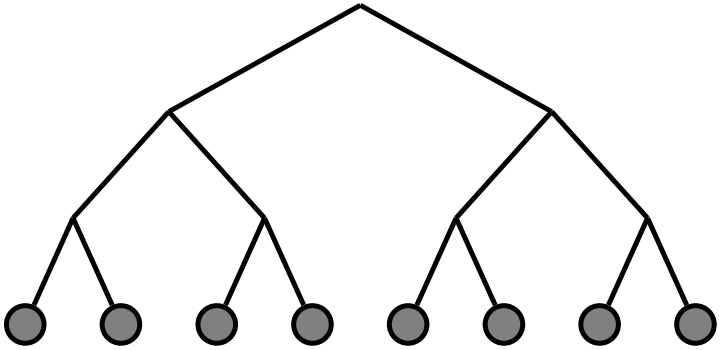
  Sizes: $2^{n+1} - 1, n \geq 0$

(Balance has to be ensured separately.)



## Example: Complete Binary Leaf Tree

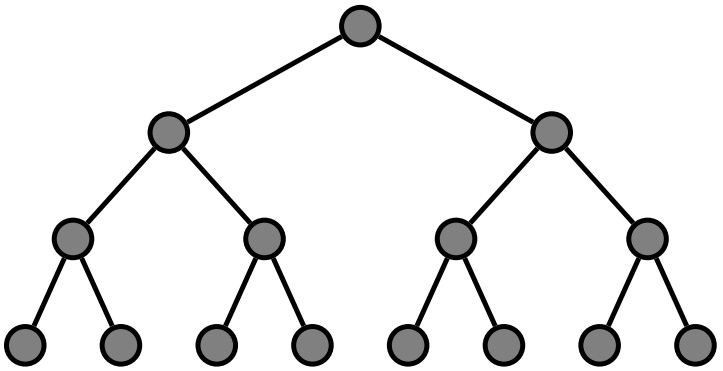
Size $2^3 = 8$:





## Example: Complete Binary Tree

Size $2^4 - 1 = 15$:



## Binary Random Access Lists (1)

***Binary Random Access Lists*** are induced by

- the usual binary representation, i.e. $w_i = 2^i$, $D_i = \{0, 1\}$
- complete binary leaf trees

Thus:

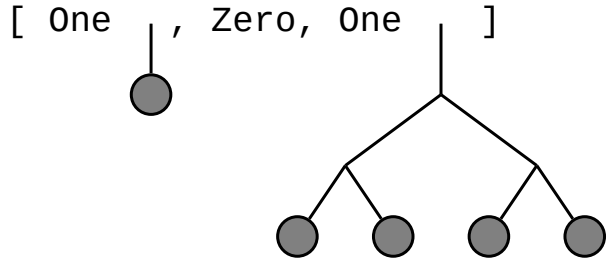
```
data Tree a = Leaf a
            | Node Int (Tree a) (Tree a)
data Digit a = Zero | One (Tree a)
type RList a = [Digit a]
```

The `Int` field keeps track of tree size for speed.



## Binary Random Access Lists (2)

Example: Binary Random Access List of size 5:






## Binary Random Access Lists (3)

The increment function on dense binary numbers:

```
inc [] = [One]
inc (Zero : ds) = One : ds
inc (One  : ds) = Zero : inc ds  -- Carry
```



## Binary Random Access Lists (4)

Inserting an element first in a binary random access list is analogous to `inc`:

```
cons :: a -> RList a -> RList a
cons x ts = consTree (Leaf x) ts

consTree :: Tree a -> RList a -> RList a
consTree t []            = [One t]
consTree t (Zero : ts)   = (One t : ts)
consTree t (One t' : ts) =
    Zero : consTree (link t t') ts
```

## Binary Random Access Lists (5)

The utility function `link` joins two equally sized trees:

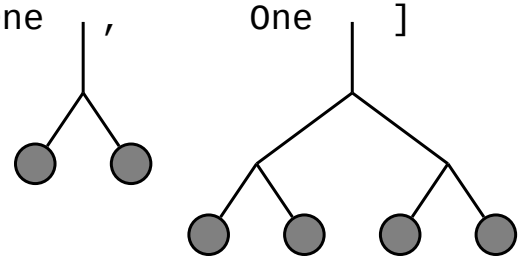
```
-- t1 and t2 are assumed to be the same size
link t1 t2 = Node (2 * size t1) t1 t2
```



## Binary Random Access Lists (6)

Example: Result of consing element onto list of size 5:

```
[ Zero, One   ,      One    ]
```





## Exercise 2: `unconsTree`

The decrement function on dense binary numbers:

```
dec [One] = []
dec (One  : ds) = Zero : ds
dec (Zero : ds) = One : dec ds  -- Borrow
```

Define `unconsTree` following the above pattern:

```
unconsTree :: RList a -> (Tree a, RList a)
```

And then `head` and `tail`:

```
head :: RList a -> a
tail :: RList a -> RList a
```



## Exercise 2: Solution (1)

```
unconsTree :: RList a -> (Tree a, RList a)
unconsTree [One t]      = (t, [])
unconsTree (One t : ts) = (t, Zero : ts)
unconsTree (Zero  : ts) = (t1, One t2 : ts')
    where
        (Node _ t1 t2, ts') = unconsTree ts
```

Note: partial operation.

## Exercise 2: Solution (2)

```
head :: RList a -> a
head ts = x
    where
        (Leaf x, _) = unconsTree ts


tail :: RList a -> RList a
tail ts = ts'
    where
        (_, ts') = unconsTree ts
```



## Binary Random Access Lists (7)

Lookup is done in two stages: first find the right tree, then lookup in that tree:

```
lookup :: Int -> RList a -> a
lookup i (Zero : ts) = lookup i ts
lookup i (One t : ts)
    | i < s       = lookupTree i t
    | otherwise   = lookup (i - s) ts
    where
        s = size t
```

Note: partial operation.



## Binary Random Access Lists (8)

```
lookupTree :: Int -> Tree a -> a
lookupTree _ (Leaf x) = x
lookupTree i (Node w t1 t2)
    | i < w `div` 2 =
        lookupTree i t1
    | otherwise =
        lookupTree (i - w `div` 2) t2
```

The operation `update` has exactly the same structure.



## Binary Random Access Lists (9)

Time complexity:

- `cons`, `head`, `tail`, perform $O(1)$ work per digit, thus $O(\log n)$ worst case.
- `lookup` and `update` take $O(\log n)$ to find the right tree, and then $O(\log n)$ to find the right element in that tree, so $O(\log n)$ worst case overall.

Time complexity for `cons`, `head`, `tail` disappointing: can we do better?

## Skew Binary Numbers (1)

Skew Binary Numbers:

- $w_i = 2^{i+1} - 1$ (rather than $2^i$)
- $D_i = \{0, 1, 2\}$

Representation is redundant. But we obtain a ***canonical form*** if we insist that only the least significant non-zero digit may be 2.

Note: The weights correspond to the sizes of ***complete*** binary trees.



## Skew Binary Numbers (2)

Theorem: Every natural number $n$ has a unique skew binary canonical form.
Proof sketch. By induction on $n$.

- Base case: the case for 0 is direct.



## Skew Binary Numbers (3)

- Inductive case. Assume $n$ has a unique skew binary representation $b_0 b_1 \ldots b_{m-1}$
  - If the least significant non-zero digit is smaller than 2, then $n + 1$ has a unique skew binary representation obtained by adding 1 to the least significant digit $b_0$.
  - If the least significant non-zero digit $b_i$ is 2, then note that $1 + 2(2^{i+1} - 1) = 2^{i+2} - 1$. Thus $n + 1$ has a unique skew binary representation obtained by setting $b_i$ to 0 and adding 1 to $b_{i+1}$.



## Exercise 3: Skew Binary Numbers

- Give the canonical skew binary representation for 31, 30, 29, and 28.
- Assume a ***sparse*** skew binary representation of the natural numbers

```
type Nat = [Int]
```

where the integers represent the ***weight*** of each ***non-zero*** digit. Assume further that the integers are stored in increasing order, except that the first two may be equal indicating that the smallest non-zero digit is 2.
Implement a function `inc` to increment a natural number.

## Exercise 3: Solution

- 00001, 0002, 0021, 0211
- ```
  inc :: Nat -> Nat
  inc (w1 : w2 : ws)
      | w1 == w2 = w1 * 2 + 1 : ws
  inc ws         = 1 : ws
  ```



## Skew Binary Random Access Lists (1)

```
data Tree a = Leaf a | Node (Tree a) a (Tree a)
type RList a = [(Int, Tree a)]

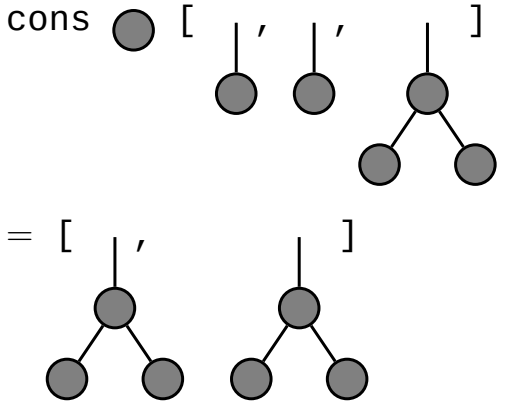
empty :: RList a
empty = []

cons :: a -> RList a -> RList a
cons x ((w1, t1) : (w2, t2) : wts) | w1 == w2 =
    (w1 * 2 + 1, Node t1 x t2) : wts
cons x wts = ((1, Leaf x) : wts)
```



## Skew Binary Random Access Lists (2)

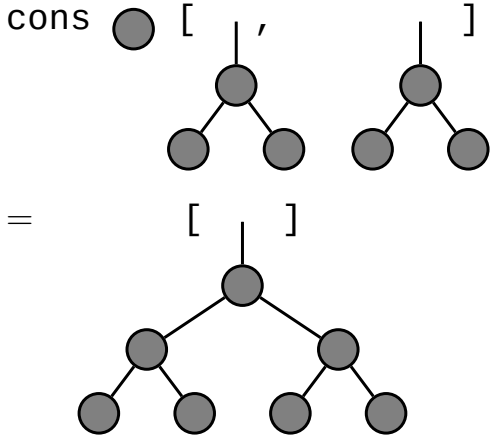
Example: Consing onto list of size 5:





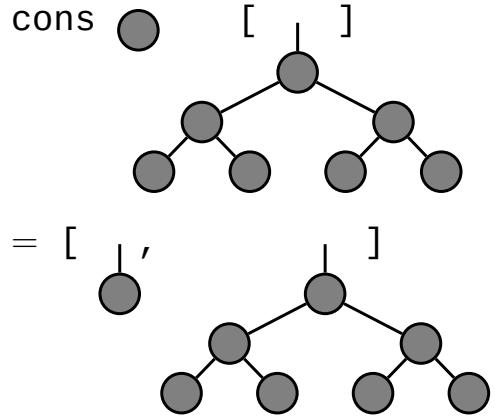

## Skew Binary Random Access Lists (3)

Example: Consing onto list of size 6:


```

## Skew Binary Random Access Lists (4)

Example: Consing onto list of size 7:






## Skew Binary Random Access Lists (5)

```
head :: RList a -> a
head ((_, Leaf x)     : _) = x
head ((_, Node _ x _) : _) = x

tail :: RList a -> RList a
tail ((_, Leaf _): wts) = wts
tail ((w, Node t1 _ t2) : wts) =
    (w', t1) : (w', t2) : wts
    where
        w' = w `div` 2
```

Note: again, partial operations.



## Skew Binary Random Access Lists (6)

```
lookup :: Int -> RList a -> a
lookup i ((w, t) : wts)
    | i < w      = lookupTree i w t
    | otherwise  = lookup (i - w) wts
lookupTree :: Int -> Int -> Tree a -> a
lookupTree _ _ (Leaf x) = x
lookupTree i w (Node t1 x t2)
    | i == 0    = x
    | i < w'    = lookupTree (i - 1) w' t1
    | otherwise = lookupTree (i - w' - 1) w' t2
    where
        w' = w `div` 2
```



## Skew Binary Random Access Lists (7)

Time complexity:

- `cons`, `head`, `tail`: $O(1)$.
- `lookup` and `update` take $O(\log n)$ to find the right tree, and then $O(\log n)$ to find the right element in that tree, so $O(\log n)$ worst case overall.

Okasaki:

> "Although there are better implementations of lists, and better implementations of (persistent) arrays, none are better at both."

# MGS 2011: FUN Lecture 3

*Monads*

Henrik Nilsson

University of Nottingham, UK



## A Blessing and a Curse

- The ***BIG*** advantage of ***pure*** functional programming is

  **“everything is explicit;”**

  i.e., flow of data manifest, no side effects. Makes it a lot easier to understand large programs.

- The ***BIG*** problem with ***pure*** functional programming is

  **“everything is explicit.”**

  Can add a lot of clutter, make it hard to maintain code



## Conundrum

***“Shall I be pure or impure?”*** (Wadler, 1992)

- Absence of effects
  - facilitates understanding and reasoning
  - makes lazy evaluation viable
  - allows choice of reduction order, e.g. parallel
  - enhances modularity and reuse.
- Effects (state, exceptions, . . . ) can
  - help making code concise
  - facilitate maintenance
  - improve the efficiency.



## Example: A Compiler Fragment (1)

***Identification*** is the task of relating each applied identifier occurrence to its declaration or definition:

```
public class C {
    int x, n;
    void set(int n) { x = n; }
}
```

In the body of `set`, the one applied occurrence of

- x refers to the ***instance variable*** x
- n refers to the ***argument*** n.

## Example: A Compiler Fragment (2)

Consider an AST `Exp` for a simple expression language. `Exp` is a parameterized type: the ***type parameter*** **a** allows variables to be annotated with an attribute of type **a**.

```
data Exp a
    = LitInt   Int
    | Var      Id a
    | UnOpApp  UnOp (Exp a)
    | BinOpApp BinOp (Exp a) (Exp a)
    | If       (Exp a) (Exp a) (Exp a)
    | Let      [(Id, Type, Exp a)] (Exp a)
```



## Example: A Compiler Fragment (3)

Example: The following code fragment

```
let int x = 7 in x + 35
```

would be represented like this (before identification):

```
Let [("x", IntType, LitInt 7)]
    (BinOpApp Plus
              (Var "x" ())
              (LitInt 35))
```



## Example: A Compiler Fragment (4)

Goals of the ***identification*** phase:

- Annotate each applied identifier occurrence with attributes of the corresponding variable declaration.
  I.e., map unannotated ~~AST~~ **`Exp ()`** to annotated AST **`Exp Attr`**.
- Report conflicting variable definitions and undefined variables.

```
identification ::
    Exp () -> Exp Attr, [ErrorMsg])
```



## Example: A Compiler Fragment (5)

Example: Before Identification

```
Let [("x", IntType, LitInt 7)]
    (BinOpApp Plus
              (Var "x" ())
              (LitInt 35))
```

After identification:

```
Let [("x", IntType, LitInt 7)]
    (BinOpApp Plus
              (Var "x" (1, IntType))
              (LitInt 35))
```

## Example: A Compiler Fragment (6)

`enterVar` inserts a variable at the given scope level and of the given type into an environment.

- Check that no variable with same name has been defined at the same scope level.
- If not, the new variable is entered, and the ***resulting environment*** is returned.
- Otherwise an ***error message*** is returned.

```
enterVar :: Id -> Int -> Type -> Env
            -> Either  Env   ErrorMsg
```



## Example: A Compiler Fragment (7)

Functions that do the real work:

```
identAux ::
    Int -> Env -> Exp ()
    -> (Exp Attr, [ErrorMsg])

identDefs ::
    Int -> Env -> [(Id, Type, Exp ())]
    -> ([(Id, Type, Exp Attr)],
        Env,
        [ErrorMsg])
```



## Example: A Compiler Fragment (8)

```
identDefs l env [] = ([], env, [])
identDefs l env ((i,t,e) : ds) =
  ((i,t,e') : ds', env'', ms1++ms2++ms3)
  where
    (e', ms1) = identAux l env e
    (env', ms2) =
       case enterVar i l t env of
          Left env' -> (env', [])
          Right m   -> (env,  [m])
    (ds', env'', ms3) =
      identDefs l env' ds
```



## Example: A Compiler Fragment (9)

Error checking and collection of error messages arguably added a lot of ***clutter***. The ***core*** of the algorithm is this:

```
identDefs l env [] = ([], env)
identDefs l env ((i,t,e) : ds) =
  ((i,t,e') : ds', env'')
  where
    e'           = identAux l env e
    env'         = enterVar i l t env
    (ds', env'') = identDefs l env' ds
```

Errors are just a ***side effect***.

## Answer to Conundrum: Monads (1)

- Monads bridges the gap: allow effectful programming in a pure setting.
- Key idea: ***Computational types***: an object of type $MA$ denotes a ***computation*** of an object of type $A$.
- ***Thus we shall be both pure and impure, whatever takes our fancy!***
- Monads originated in Category Theory.
- Adapted by
  - Moggi for structuring denotational semantics
  - Wadler for structuring functional programs



## Answer to Conundrum: Monads (2)

Monads

- promote disciplined use of effects since the type reflects which effects can occur;
- allow great flexibility in tailoring the effect structure to precise needs;
- support changes to the effect structure with minimal impact on the overall program structure;
- allow integration into a pure setting of ***real*** effects such as
  - I/O
  - mutable state.



## This Lecture

Pragmatic introduction to monads:

- Effectful computations
- Identifying a common pattern
- Monads as a ***design pattern***



## Example 1: A Simple Evaluator

```
data Exp = Lit Integer
         | Add Exp Exp
         | Sub Exp Exp
         | Mul Exp Exp
         | Div Exp Exp

eval :: Exp -> Integer
eval (Lit n)     = n
eval (Add e1 e2) = eval e1 + eval e2
eval (Sub e1 e2) = eval e1 - eval e2
eval (Mul e1 e2) = eval e1 * eval e2
eval (Div e1 e2) = eval e1 `div` eval e2
```

## Making the Evaluator Safe (1)

```
data Maybe a = Nothing | Just a

safeEval :: Exp -> Maybe Integer
safeEval (Lit n) = Just n
safeEval (Add e1 e2) =
    case safeEval e1 of
        Nothing -> Nothing
        Just n1 ->
            case safeEval e2 of
                Nothing -> Nothing
                Just n2 -> Just (n1 + n2)
```



## Making the Evaluator Safe (2)

```
safeEval (Sub e1 e2) =
    case safeEval e1 of
        Nothing -> Nothing
        Just n1 ->
            case safeEval e2 of
                Nothing -> Nothing
                Just n2 -> Just (n1 - n2)
```



## Making the Evaluator Safe (3)

```
safeEval (Mul e1 e2) =
    case safeEval e1 of
        Nothing -> Nothing
        Just n1 ->
            case safeEval e2 of
                Nothing -> Nothing
                Just n2 -> Just (n1 * n2)
```



## Making the Evaluator Safe (4)

```
safeEval (Div e1 e2) =
    case safeEval e1 of
        Nothing -> Nothing
        Just n1 ->
            case safeEval e2 of
                Nothing -> Nothing
                Just n2 ->
                    if n2 == 0
                    then Nothing
                    else Just (n1 `div` n2)
```

## Any Common Pattern?

Clearly a lot of code duplication!
Can we factor out a common pattern?

We note:

- ***Sequencing*** of evaluations (or ***computations***).
- If one evaluation fails, fail overall.
- Otherwise, make result available to following evaluations.



## Sequencing Evaluations

```
evalSeq :: Maybe Integer
           -> (Integer -> Maybe Integer)
           -> Maybe Integer
evalSeq ma f =
    case ma of
        Nothing -> Nothing
        Just a  -> f a
```



## Exercise 1: Refactoring `safeEval`

Rewrite `safeEval`, case `Add`, using `evalSeq`:

```
safeEval (Add e1 e2) =
    case safeEval e1 of
        Nothing -> Nothing
        Just n1 ->
            case safeEval e2 of
                Nothing -> Nothing
                Just n2 -> Just (n1 + n2)
evalSeq ma f =
    case ma of
        Nothing -> Nothing
        Just a  -> f a
```



## Exercise 1: Solution

```
safeEval :: Exp -> Maybe Integer
safeEval (Add e1 e2) =
    evalSeq (safeEval e1)
            (\n1 -> evalSeq (safeEval e2)
                            (\n2 -> Just (n1+n2
```

or

```
safeEval :: Exp -> Maybe Integer
safeEval (Add e1 e2) =
    safeEval e1 `evalSeq` (\n1 ->
    safeEval e2 `evalSeq` (\n2 ->
    Just (n1 + n2)))
```

## Aside: Scope Rules of $\lambda$-abstractions

The scope rules of $\lambda$-abstractions are such that parentheses can be omitted:

```
safeEval :: Exp -> Maybe Integer
...
safeEval (Add e1 e2) =
    safeEval e1 ‘evalSeq‘ \n1 ->
    safeEval e2 ‘evalSeq‘ \n2 ->
    Just (n1 + n2)
...
```



## Refactored Safe Evaluator (1)

```
safeEval :: Exp -> Maybe Integer
safeEval (Lit n) = Just n
safeEval (Add e1 e2) =
    safeEval e1 ‘evalSeq‘ \n1 ->
    safeEval e2 ‘evalSeq‘ \n2 ->
    Just (n1 + n2)
safeEval (Sub e1 e2) =
    safeEval e1 ‘evalSeq‘ \n1 ->
    safeEval e2 ‘evalSeq‘ \n2 ->
    Just (n1 - n2)
```



## Refactored Safe Evaluator (2)

```
safeEval (Mul e1 e2) =
    safeEval e1 ‘evalSeq‘ \n1 ->
    safeEval e2 ‘evalSeq‘ \n2 ->
    Just (n1 * n2)
safeEval (Div e1 e2) =
    safeEval e1 ‘evalSeq‘ \n1 ->
    safeEval e2 ‘evalSeq‘ \n2 ->
    if n2 == 0
    then Nothing
    else Just (n1 ‘div‘ n2)
```



## Inlining `evalSeq` (1)

```
safeEval (Add e1 e2) =
  safeEval e1 ‘evalSeq‘ \n1 ->
  safeEval e2 ‘evalSeq‘ \n2 ->
  Just (n1 + n2)
=
safeEval (Add e1 e2) =
  case (safeEval e1) of
    Nothing -> Nothing
    Just a -> (\n1 -> safeEval e2 ...) a
```

## Inlining `evalSeq` (2)

=

```
safeEval (Add e1 e2) =
  case (safeEval e1) of
    Nothing -> Nothing
    Just n1 -> safeEval e2 ‘evalSeq‘ (\n2 -> ...)
```

=

```
safeEval (Add e1 e2) =
  case (safeEval e1) of
    Nothing -> Nothing
    Just n1 -> case safeEval e2 of
                 Nothing -> Nothing
                 Just a -> (\n2 -> ...) a
```



## Inlining `evalSeq` (3)

=

```
safeEval (Add e1 e2) =
  case (safeEval e1) of
    Nothing -> Nothing
    Just n1 -> case safeEval e2 of
                 Nothing -> Nothing
                 Just n2 -> (Just n1 + n2)
```

Good excercise: verify the other cases.



## `Maybe` Viewed as a Computation (1)

- Consider a value of type `Maybe a` as denoting a ***computation*** of a value of type a that ***may fail***.
- When sequencing possibly failing computations, a natural choice is to fail overall once a subcomputation fails.
- I.e. ***failure is an effect***, implicitly affecting subsequent computations.
- Let’s generalize and adopt names reflecting our intentions.



## `Maybe` Viewed as a Computation (2)

Successful computation of a value:

```
mbReturn :: a -> Maybe a
mbReturn = Just
```

Sequencing of possibly failing computations:

```
mbSeq :: Maybe a -> (a -> Maybe b) -> Maybe b
mbSeq ma f =
    case ma of
        Nothing -> Nothing
        Just a  -> f a
```

## `Maybe` Viewed as a Computation (3)

Failing computation:

```
mbFail :: Maybe a
mbFail = Nothing
```



## The Safe Evaluator Revisited

```
safeEval :: Exp -> Maybe Integer
safeEval (Lit n) = mbReturn n
safeEval (Add e1 e2) =
    safeEval e1 ‘mbSeq‘ \n1 ->
    safeEval e2 ‘mbSeq‘ \n2 ->
    mbReturn (n1 + n2)
...
safeEval (Div e1 e2) =
    safeEval e1 ‘mbSeq‘ \n1 ->
    safeEval e2 ‘mbSeq‘ \n2 ->
    if n2 == 0 then mbFail
    else mbReturn (n1 ‘div‘ n2)))
```



## Example 2: Numbering Trees

```
data Tree a = Leaf a | Node (Tree a) (Tree a)

numberTree :: Tree a -> Tree Int
numberTree t = fst (ntAux t 0)
    where
        ntAux :: Tree a -> Int -> (Tree Int,Int)
        ntAux (Leaf _)     n = (Leaf n, n+1)
        ntAux (Node t1 t2) n =
            let (t1’, n’)  = ntAux t1 n
            in let (t2’, n’’) = ntAux t2 n’
               in (Node t1’ t2’, n’’)
```



## Observations

- Repetitive pattern: threading a counter through a ***sequence*** of tree numbering ***computations***.
- It is very easy to pass on the wrong version of the counter!

Can we do better?

## Stateful Computations (1)

- A ***stateful computation*** consumes a state and returns a result along with a possibly updated state.
- The following type synonym captures this idea:

```
type S a = Int -> (a, Int)
```

  (Only `Int` state for the sake of simplicity.)
- A value (function) of type `S a` can now be viewed as denoting a stateful computation computing a value of type `a`.



## Stateful Computations (2)

- When sequencing stateful computations, the resulting state should be passed on to the next computation.
- I.e. ***state updating is an effect***, implicitly affecting subsequent computations.
  (As we would expect.)



## Stateful Computations (3)

Computation of a value without changing the state (For ref.: `S a = Int -> (a, Int)`):

```
sReturn :: a -> S a
sReturn a = \n -> (a, n)
```

Sequencing of stateful computations:

```
sSeq :: S a -> (a -> S b) -> S b
sSeq sa f = \n ->
    let (a, n') = sa n
    in f a n'
```



## Stateful Computations (4)

Reading and incrementing the state (For ref.: `S a = Int -> (a, Int)`):

```
sInc :: S Int
sInc = \n -> (n, n + 1)
```

## Numbering trees revisited

```
data Tree a = Leaf a | Node (Tree a) (Tree a)

numberTree :: Tree a -> Tree Int
numberTree t = fst (ntAux t 0)
    where
        ntAux :: Tree a -> S (Tree Int)
        ntAux (Leaf _)  =
            sInc ‘sSeq‘ \n -> sReturn (Leaf n)
        ntAux (Node t1 t2) =
            ntAux t1 ‘sSeq‘ \t1’ ->
            ntAux t2 ‘sSeq‘ \t2’ ->
            sReturn (Node t1’ t2’)
```



## Observations

- The “plumbing” has been captured by the abstractions.
- In particular:
  - counter no longer manipulated directly
  - no longer any risk of “passing on” the wrong version of the counter!



## Comparison of the examples

- Both examples characterized by sequencing of effectful computations.
- Both examples could be neatly structured by introducing:
  - A type denoting computations
  - A function constructing an effect-free computation of a value
  - A function constructing a computation by sequencing computations
- In fact, both examples are instances of the general notion of a ***MONAD***.



## Monads in Functional Programming

A monad is represented by:

- A type constructor

  `M :: * -> *`

  `M T` represents computations of a value of type `T`
- A polymorphic function

  `return :: a -> M a`

  for lifting a value to a computation.
- A polymorphic function

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

  for sequencing computations.

## Exercise 2: `join` and `fmap`

Equivalently, the notion of a monad can be captured through the following functions:

```
return :: a -> M a
join :: (M (M a)) -> M a
fmap :: (a -> b) -> (M a -> M b)
```

`join` "flattens" a computation, `fmap` "lifts" a function to map computations to computations.

Define `join` and `fmap` in terms of >>= (and `return`), and >>= in terms of `join` and `fmap`.

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



## Exercise 2: Solution

```
join :: M (M a) -> M a
join mm = mm >>= id

fmap :: (a -> b) -> M a -> M b
fmap f m = m >>= \a -> return (f a)
```

or:

```
fmap :: (a -> b) -> M a -> M b
fmap f m = m >>= return . f

(>>=) :: M a -> (a -> M b) -> M b
m >>= f = join (fmap f m)
```



## Monad laws

Additionally, the following ***laws*** must be satisfied:

$$\begin{aligned} \texttt{return}\ x\ \texttt{>>=}\ f &= f\ x \\ m\ \texttt{>>=}\ \texttt{return} &= m \\ (m\ \texttt{>>=}\ f)\ \texttt{>>=}\ g &= m\ \texttt{>>=}\ (\lambda x \rightarrow f\ x\ \texttt{>>=}\ g) \end{aligned}$$

I.e., `return` is the right and left identity for >>=, and >>= is associative.



## Exercise 3: The Identity Monad

The ***Identity Monad*** can be understood as representing ***effect-free*** computations:

```
type I a = a
```

1. Provide suitable definitions of `return` and >>=.
2. Verify that the monad laws hold for your definitions.

## Exercise 3: Solution

```
return :: a -> I a
return = id

(>>=) :: I a -> (a -> I b) -> I b
m >>= f = f m
-- or: (>>=) = flip ($)
```

Simple calculations verify the laws, e.g.:

$$\begin{aligned} \texttt{return}\ x\ \texttt{>>=}\ f &= \texttt{id}\ x\ \texttt{>>=}\ f \\ &= x\ \texttt{>>=}\ f \\ &= f\ x \end{aligned}$$



## Monads in Category Theory (1)

The notion of a monad originated in Category Theory. There are several equivalent definitions (Benton, Hughes, Moggi 2000):

- ***Kleisli triple/triple in extension form:*** Most closely related to the >>= version:

  A ***Klesili triple*** over a category $\mathcal{C}$ is a triple $(T, \eta, \_^*)$, where $T : |\mathcal{C}| \to |\mathcal{C}|$, $\eta_A : A \to TA$ for $A \in |\mathcal{C}|$, $f^* : TA \to TB$ for $f : A \to TB$.

  (Additionally, some laws must be satisfied.)



## Monads in Category Theory (2)

- ***Monad/triple in monoid form:*** More akin to the `join`/`fmap` version:

  A ***monad*** over a category $\mathcal{C}$ is a triple $(T, \eta, \mu)$, where $T : \mathcal{C} \to \mathcal{C}$ is a functor, $\eta : \mathrm{id}_{\mathcal{C}} \dot{\to} T$ and $\mu : T^2 \dot{\to} T$ are natural transformations.

  (Additionally, some commuting diagrams must be satisfied.)



## Reading


- Philip Wadler. The Essence of Functional Programming. *Proceedings of the 19th ACM Symposium on Principles of Programming Languages (POPL'92)*, 1992.
- Nick Benton, John Hughes, Eugenio Moggi. Monads and Effects. In *International Summer School on Applied Semantics 2000*, Caminha, Portugal, 2000.
- *All About Monads.* `http://www.haskell.org/all_about_monads`

# MGS 2011: FUN Lecture 4

*More about Monads*

Henrik Nilsson

University of Nottingham, UK



# This Lecture

- Monads in Haskell
- Some standard monads
- Combining effects: monad transformers



# Monads in Haskell

In Haskell, the notion of a monad is captured by a ***Type Class***:

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

Allows names of the common functions to be overloaded and sharing of derived definitions.



# The `Maybe` Monad in Haskell

```
instance Monad Maybe where
    -- return :: a -> Maybe a
    return = Just

    -- (>>=) :: Maybe a -> (a -> Maybe b)
    --          -> Maybe b
    Nothing  >>= _ = Nothing
    (Just x) >>= f = f x
```

## Exercise 1: A State Monad in Haskell

Haskell 98 does not permit type synonyms to be instances of classes. Hence we have to define a new type:

```
newtype S a = S (Int -> (a, Int))

unS :: S a -> (Int -> (a, Int))
unS (S f) = f
```

Provide a `Monad` instance for `S`.



## Exercise 1: Solution

```
instance Monad S where
    return a = S (\s -> (a, s))

    m >>= f = S $ \s ->
        let (a, s') = unS m s
        in unS (f a) s'
```



## Monad-specific Operations (1)

To be useful, monads need to be equipped with additional operations specific to the effects in question. For example:

```
fail :: String -> Maybe a
fail s = Nothing

catch :: Maybe a -> Maybe a -> Maybe a
m1 `catch` m2 =
    case m1 of
        Just _  -> m1
        Nothing -> m2
```



## Monad-specific Operations (2)

Typical operations on a state monad:

```
set :: Int -> S ()
set a = S (\_ -> ((), a))

get :: S Int
get = S (\s -> (s, s))
```

Moreover, need to "run" a computation. E.g.:

```
runS :: S a -> a
runS m = fst (unS m 0)
```

## The do-notation (1)

Haskell provides convenient syntax for programming with monads:

```
do
    a <- exp1
    b <- exp2
    return exp3
```

is syntactic sugar for

```
exp1 >>= \a ->
exp2 >>= \b ->
return exp3
```



## The do-notation (2)

Computations can be done solely for effect, ignoring the computed value:

```
do
    exp1
    exp2
    return exp3
```

is syntactic sugar for

```
exp1 >>= \_ ->
exp2 >>= \_ ->
return exp3
```



## The do-notation (3)

A `let`-construct is also provided:

```
do
    let a = exp1
        b = exp2
    return exp3
```

is equivalent to

```
do
    a <- return exp1
    b <- return exp2
    return exp3
```



## Numbering Trees in do-notation

```
numberTree :: Tree a -> Tree Int
numberTree t = runS (ntAux t)
  where
    ntAux :: Tree a -> S (Tree Int)
    ntAux (Leaf _) = do
        n <- get
        set (n + 1)
        return (Leaf n)
    ntAux (Node t1 t2) = do
        t1' <- ntAux t1
        t2' <- ntAux t2
        return (Node t1' t2')
```

## The Compiler Fragment Revisited (1)

Given a suitable “Diagnostics” monad `D` that collects error messages, `enterVar` can be turned from this:

```
enterVar :: Id -> Int -> Type -> Env
            -> Either Env ErrorMgs
```

into this:

```
enterVarD :: Id -> Int -> Type -> Env
             -> D Env
```

and then `identDefs` from this . . .



## The Compiler Fragment Revisited (2)

```
identDefs l env [] = ([], env, [])
identDefs l env ((i,t,e) : ds) =
  ((i,t,e') : ds', env'', ms1++ms2++ms3)
  where
    (e', ms1) = identAux l env e
    (env', ms2) =
       case enterVar i l t env of
          Left env' -> (env', [])
          Right m   -> (env,  [m])
    (ds', env'', ms3) =
      identDefs l env' ds
```



## The Compiler Fragment Revisited (3)

into this:

```
identDefsD l env [] = return ([], env)
identDefsD l env ((i,t,e) : ds) = do
    e'           <- identAuxD l env e
    env'         <- enterVarD i l t env
    (ds', env'') <- identDefsD l env' ds
    return ((i,t,e') : ds', env'')
```

(Suffix D just to remind us the types have changed.)



## The Compiler Fragment Revisited (4)

Compare with the “core” identified earlier!

```
identDefs l env [] = ([], env)
identDefs l env ((i,t,e) : ds) =
  ((i,t,e') : ds', env'')
  where
    e'           = identAux l env e
    env'         = enterVar i l t env
    (ds', env'') = identDefs l env' ds
```

The monadic version is very close to ideal, without sacrificing functionality, clarity, or pureness!

## The List Monad

Computation with many possible results, “nondeterminism”:

```
instance Monad [] where
    return a = [a]
    m >>= f  = concat (map f m)
    fail s   = []
```

Example:

```
x <- [1, 2]
y <- ['a', 'b']
return (x,y)
```

Result:

```
[(1,'a'),(1,'b'),
 (2,'a'),(2,'b')]
```



## The Reader Monad

Computation in an environment:

```
instance Monad ((->) e) where
    return a = const a
    m >>= f  = \e -> f (m e) e

getEnv :: ((->) e) e
getEnv = id
```



## The Haskell IO Monad

In Haskell, IO is handled through the IO monad. IO is ***abstract***! Conceptually:

```
newtype IO a = IO (World -> (a, World))
```

Some operations:

```
putChar     :: Char -> IO ()
putStr      :: String -> IO ()
putStrLn    :: String -> IO ()
getChar     :: IO Char
getLine     :: IO String
getContents :: String
```



## Monad Transformers (1)

What if we need to support more than one type of effect?

For example: State and Error/Partiality?

We could implement a suitable monad from scratch:

```
newtype SE s a = SE (s -> Maybe (a, s))
```

## Monad Transformers (2)

However:

- Not always obvious how: e.g., should the combination of state and error have been

```
newtype SE s a = SE (s -> (Maybe a, s))
```

- Duplication of effort: similar patterns related to specific effects are going to be repeated over and over in the various combinations.



## Monad Transformers (3)

***Monad Transformers*** can help:

- A ***monad transformer*** transforms a monad by adding support for an additional effect.
- A library of monad transformers can be developed, each adding a specific effect (state, error, . . . ), allowing the programmer to mix and match.
- A form of ***aspect-oriented programming***.



## Monad Transformers in Haskell (1)

- A ***monad transformer*** maps monads to monads. Represented by a type constructor `T` of the following kind:

```
T :: (* -> *) -> (* -> *)
```

- Additionally, a monad transformer ***adds*** computational effects. A mapping `lift` from computations in the underlying monad to computations in the transformed monad is needed:

```
lift :: M a -> T M a
```



## Monad Transformers in Haskell (2)

- These requirements are captured by the following (multi-parameter) type class:

```
class (Monad m, Monad (t m))
      => MonadTransformer t m where
    lift :: m a -> t m a
```

## Classes for Specific Effects

A monad transformer adds specific effects to ***any*** monad. Thus the effect-specific operations needs to be overloaded. For example:

```
class Monad m => E m where
    eFail :: m a
    eHandle :: m a -> m a -> m a

class Monad m => S m s | m -> s where
    sSet :: s -> m ()
    sGet :: m s
```



## The Identity Monad

We are going to construct monads by successive transformations of the identity monad:

```
newtype I a = I a
unI (I a) = a

instance Monad I where
    return a = I a
    m >>= f = f (unI m)

runI :: I a -> a
runI = unI
```



## The Error Monad Transformer (1)

```
newtype ET m a = ET (m (Maybe a))
unET (ET m) = m
```

Any monad transformed by ET is a monad:

```
instance Monad m => Monad (ET m) where
    return a = ET (return (Just a))

    m >>= f = ET $ do
        ma <- unET m
        case ma of
            Nothing -> return Nothing
            Just a  -> unET (f a)
```



## The Error Monad Transformer (2)

We need the ability to run transformed monads:

```
runET :: Monad m => ET m a -> m a
runET etm = do
    ma <- unET etm
    case ma of
        Just a  -> return a
        Nothing -> error "Should not happen"
```

ET is a monad transformer:

```
instance Monad m =>
         MonadTransformer ET m where
    lift m = ET (m >>= \a -> return (Just a))
```

## The Error Monad Transformer (3)

Any monad transformed by ET is an instance of E:

```
instance Monad m => E (ET m) where
    eFail = ET (return Nothing)
    m1 `eHandle` m2 = ET $ do
        ma <- unET m1
        case ma of
            Nothing -> unET m2
            Just _  -> return ma
```



## The Error Monad Transformer (4)

A state monad transformed by ET is a state monad:

```
instance S m s => S (ET m) s where
    sSet s = lift (sSet s)
    sGet = lift sGet
```



## Exercise 2: Running Transf. Monads

Let

```
ex2 = eFail `eHandle` return 1
```

1. Suggest a possible type for ex2. (Assume `1 :: Int`.)
2. Given your type, use the appropriate combination of "run functions" to run ex2.



## Exercise 2: Solution

```
ex2 :: ET I Int
ex2 = eFail `eHandle` return 1

ex2result :: Int
ex2result = runI (runET ex2)
```

## The State Monad Transformer (1)

```
newtype ST s m a = ST (s -> m (a, s))
unST (ST m) = m
```

Any monad transformed by ST is a monad:

```
instance Monad m => Monad (ST s m) where
    return a = ST (\s -> return (a, s))

    m >>= f = ST $ \s -> do
        (a, s') <- unST m s
        unST (f a) s'
```



## The State Monad Transformer (2)

We need the ability to run transformed monads:

```
runST :: Monad m => ST s m a -> s -> m a
runST stf s0 = do
    (a, _) <- unST stf s0
    return a
```

ST is a monad transformer:

```
instance Monad m =>
         MonadTransformer (ST s) m where
    lift m = ST (\s -> m >>= \a ->
                       return (a, s))
```



## The State Monad Transformer (3)

Any monad transformed by ST is an instance of S:

```
instance Monad m => S (ST s m) s where
    sSet s = ST (\_ -> return ((), s))
    sGet   = ST (\s -> return (s, s))
```

An error monad transformed by ST is an error monad:

```
instance E m => E (ST s m) where
    eFail = lift eFail
    m1 `eHandle` m2 = ST $ \s ->
        unST m1 s `eHandle` unST m2 s
```



## Exercise 3: Effect Ordering

Consider the code fragment

```
ex3a :: (ST Int (ET I)) Int
ex3a = (sSet 42 >> eFail) `eHandle` sGet
```

Note that the exact same code fragment also can be typed as follows:

```
ex3b :: (ET (ST Int I)) Int
ex3b = (sSet 42 >> eFail) `eHandle` sGet
```

What is

```
runI (runET (runST ex3a 0))
runI (runST (runET ex3b) 0)
```

## Exercise 3: Solution

```
runI (runET (runST ex3a 0)) = 0
runI (runST (runET ex3b) 0) = 42
```

Why? Because:

```
ST s (ET I) a ≅ s -> (ET I) (a, s)
              ≅ s -> I (Maybe (a, s))
              ≅ s -> Maybe (a, s)
ET (ST s I) a ≅ (ST s I) (Maybe a)
              ≅ s -> I (Maybe a, s)
              ≅ s -> (Maybe a, s)
```



## Exercise 4: Alternative ST?

To think about.

Could ST have been defined in some other way, e.g.

```
newtype ST s m a = ST (m (s -> (a, s)))
```

or perhaps

```
newtype ST s m a = ST (s -> (m a, s))
```



## Problems with Monad Transformers

- With one transformer for each possible effect, we get a lot of combinations: the number grows quadratically; each has to be instantiated explicitly.
- Jaskelioff (2008,2009) has proposed a possible, more extensible alternative.



## Reading (1)


- Nick Benton, John Hughes, Eugenio Moggi. Monads and Effects. In *International Summer School on Applied Semantics 2000*, Caminha, Portugal, 2000.
- Sheng Liang, Paul Hudak, Mark Jones. Monad Transformers and Modular Interpreters. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages (POPL'95)*, January 1995, San Francisco, California

# Reading (2)

- Mauro Jaskelioff. Monatron: An Extensible Monad Transformer Library. In *Implementation of Functional Languages (IFL'08)*, 2008.
- Mauro Jaskelioff. Modular Monad Transformers. In *European Symposium on Programming (ESOP,09)*, 2009.

# MGS 2011: FUN Lecture 5

*Concurrency*

Henrik Nilsson

University of Nottingham, UK



# This Lecture

- A concurrency monad (adapted from Claessen (1999))
- Basic concurrent programming in Haskell
- Software Transactional Memory (the STM monad)



# A Concurrency Monad (1)

A `Thread` represents a process: a stream of primitive ***atomic*** operations:

```
data Thread = Print Char Thread
            | Fork Thread Thread
            | End
```

Note that a `Thread` represents the ***entire rest*** of a computation.



# A Concurrency Monad (2)

Introduce a monad representing “interleavable computations”. At this stage, this amounts to little more than a convenient way to construct threads by sequential composition.

How can `Threads` be constructed sequentially? The only way is to parameterize thread prefixes on the rest of the `Thread`. This leads directly to ***continuations***.

## A Concurrency Monad (3)

```
newtype CM a = CM ((a -> Thread) -> Thread)

fromCM :: CM a -> ((a -> Thread) -> Thread)
fromCM (CM x) = x

thread :: CM a -> Thread
thread m = fromCM m (const End)

instance Monad CM where
    return x = CM (\k -> k x)
    m >>= f  = CM $ \k ->
         fromCM m (\x -> fromCM (f x) k)
```



## A Concurrency Monad (4)

Atomic operations:

```
cPrint :: Char -> CM ()
cPrint c = CM (\k -> Print c (k ()))

cFork :: CM a -> CM ()
cFork m = CM (\k -> Fork (thread m) (k ()))

cEnd :: CM a
cEnd = CM (\_ -> End)
```



## Running a Concurrent Computation (1

Running a computation:

```
type Output = [Char]
type ThreadQueue = [Thread]
type State = (Output, ThreadQueue)

runCM :: CM a -> Output
runCM m = runHlp ("", []) (thread m)
    where
        runHlp s t =
            case dispatch s t of
                Left (s', t) -> runHlp s' t
                Right o      -> o
```



## Running a Concurrent Computation (2

Dispatch on the operation of the currently running `Thread`. Then call the scheduler.

```
dispatch :: State -> Thread
            -> Either (State, Thread) Output
dispatch (o, rq) (Print c t) =
    schedule (o ++ [c], rq ++ [t])
dispatch (o, rq) (Fork t1 t2) =
    schedule (o, rq ++ [t1, t2])
dispatch (o, rq) End =
    schedule (o, rq)
```

## Running a Concurrent Computation (3)

Selects next `Thread` to run, if any.

```
schedule :: State -> Either (State, Thread)
                            Output
schedule (o, [])   = Right o
schedule (o, t:ts) = Left ((o, ts), t)
```



## Example: Concurrent Processes

```
p1 :: CM ()
p1 = do
    cPrint 'a'
    cPrint 'b'
    ...
    cPrint 'j'

p2 :: CM ()
p2 = do
    cPrint '1'
    cPrint '2'
    ...
    cPrint '0'

p3 :: CM ()
p3 = do
    cFork p1
    cPrint 'A'
    cFork p2
    cPrint 'B'

main = print (runCM p3)
```

Result: `aAbc1Bd2e3f4g5h6i7j890`
***Note:*** As it stands, the output is only made available after ***all*** threads have terminated.)



## Incremental Output

Incremental output:

```
runCM :: CM a -> Output
runCM m = dispatch [] (thread m)

dispatch :: ThreadQueue -> Thread -> Output
dispatch rq (Print c t)  = c : schedule (rq ++ [t])
dispatch rq (Fork t1 t2) = schedule (rq ++ [t1, t2])
dispatch rq End          = schedule rq

schedule :: ThreadQueue -> Output
schedule []     = []
schedule (t:ts) = dispatch ts t
```



## Example: Concurrent processes 2

```
p1 :: CM ()
p1 = do
    cPrint 'a'
    cPrint 'b'
    ...
    cPrint 'j'

p2 :: CM ()
p2 = do
    cPrint '1'
    undefined
    ...
    cPrint '0'

p3 :: CM ()
p3 = do
    cFork p1
    cPrint 'A'
    cFork p2
    cPrint 'B'

main = print (runCM p3)
```

Result: `aAbc1Bd*** Exception: Prelude.undefined`

## Any Use?

- A number of libraries and embedded langauges use similar ideas, e.g.
  - Fudgets
  - Yampa
  - FRP in general
- Studying semantics of concurrent programs.
- Aid for testing, debugging, and reasoning about concurrent programs.



## Concurrent Programming in Haskell

Primitives for concurrent programming provided as operations of the IO monad (or “sin bin” :-). They are in the module `Control.Concurrent`. Excerpts:

```
forkIO       :: IO () -> IO ThreadId
killThread   :: ThreadId -> IO ()
threadDelay  :: Int -> IO ()
newMVar      :: a -> IO (MVar a)
newEmptyMVar :: IO (MVar a)
putMVar      :: MVar a -> a -> IO ()
takeMVar     :: MVar a -> IO a
```



## MVars

- The fundamental synchronisation mechanism is the ***`MVar`*** (“em-var”).
- An `MVar` is a “one-item box” that may be ***empty*** or ***full***.
- Reading (`takeMVar`) and writing (`putMVar`) are ***atomic*** operations:
  - Writing to an empty `MVar` makes it full.
  - Writing to a full `MVar` blocks.
  - Reading from an empty `MVar` blocks.
  - Reading from a full `MVar` makes it empty.



## Example: Basic Synchronization (1)

```
module Main where

import Control.Concurrent

countFromTo :: Int -> Int -> IO ()
countFromTo m n
    | m > n     = return ()
    | otherwise = do
        putStrLn (show m)
        countFromTo (m+1) n
```

## Example: Basic Synchronization (2)

```
main = do
    start <- newEmptyMVar
    done  <- newEmptyMVar
    forkIO $ do
        takeMVar start
        countFromTo 1 10
        putMVar done ()
    putStrLn "Go!"
    putMVar start ()
    takeMVar done
    (countFromTo 11 20)
    putStrLn "Done!"
```



## Example: Unbounded Buffer (1)

```
module Main where

import Control.Monad (when)
import Control.Concurrent

newtype Buffer a =
    Buffer (MVar (Either [a] (Int, MVar a)))

newBuffer :: IO (Buffer a)
newBuffer = do
    b <- newMVar (Left [])
    return (Buffer b)
```



## Example: Unbounded Buffer (2)

```
readBuffer :: Buffer a -> IO a
readBuffer (Buffer b) = do
    bc <- takeMVar b
    case bc of
        Left (x : xs) -> do
            putMVar b (Left xs)
            return x
        Left []       -> do
            w <- newEmptyMVar
            putMVar b (Right (1,w))
            takeMVar w
        Right (n,w)   -> do
            putMVar b (Right (n + 1, w))
            takeMVar w
```



## Example: Unbounded Buffer (3)

```
writeBuffer :: Buffer a -> a -> IO ()
writeBuffer (Buffer b) x = do
    bc <- takeMVar b
    case bc of
        Left xs ->
            putMVar b (Left (xs ++ [x]))
        Right (n,w) -> do
            putMVar w x
            if n > 1 then
                putMVar b (Right (n - 1, w))
             else
                putMVar b (Left [])
```

## Example: Unbounded Buffer (4)

The buffer can now be used as a channel of communication between a set of “writers” and a set of “readers”. E.g.

```
main = do
    b <- newBuffer
    forkIO (writer b)
    forkIO (writer b)
    forkIO (reader b)
    forkIO (reader b)
    ...
```



## Example: Unbounded Buffer (5)

```
reader :: Buffer Int -> IO ()
reader n b = rLoop
    where
        rLoop = do
            x <- readBuffer b
            when (x > 0) $ do
                putStrLn (n ++ ": " ++ show x)
                rLoop
```



## Compositionality? (1)

Suppose we would like to read two ***consecutive*** elements from a buffer b?

That is, ***sequential composition***.

Would the following work?

```
x1 <- readBuffer b
x2 <- readBuffer b
```



## Compositionality? (2)

What about this?

```
mutex <- newMVar ()
...
takeMVar mutex
x1 <- readBuffer b
x2 <- readBuffer b
putMVar mutex ()
```

## Compositionality? (3)

Suppose we would like to read from ***one of two*** buffers.

That is, ***composing alternatives***.

Hmmm. How do we even begin?

- No way to attempt reading a buffer without risking blocking.
- We have to change or enrich the buffer implementation. E.g. add a `tryReadBuffer` operation, and then repeatedly poll the two buffers in a tight loop. Not so good!



## Software Transactional Memory (1)

- Operations on shared mutable variables grouped into ***transactions***.
- A transaction either succeeds or fails in its ***entirety***. I.e., ***atomic*** w.r.t. other transactions.
- Failed transactions are automatically ***retried*** until they succeed.
- ***Transaction logs***, which records reading and writing of shared variables, maintained to enable transactions to be validated, partial transactions to be rolled back, and to determine when worth trying a transaction again.



## Software Transactional Memory (2)

- ***No locks!*** (At the application level.)



## STM and Pure Declarative Languages

- STM perfect match for ***purely declarative languages***:
  - reading and writing of shared mutable variables explicit and relatively rare;
  - most computations are pure and need not be logged.
- Disciplined use of effects through monads a ***huge*** payoff: easy to ensure that ***only*** effects that can be undone can go inside a transaction.

  (Imagine the havoc arbitrary I/O actions could cause if part of transaction: How to undo? What if retried?)

## The STM monad

The software transactional memory abstraction provided by a monad `STM`. ***Distinct from IO!*** Defined in `Control.Concurrent.STM`.

Excerpts:

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



## Example: Buffer Revisited (1)

Let us rewrite the unbounded buffer using the STM monad:

```
module Main where

import Control.Monad (when)
import Control.Concurrent
import Control.Concurrent.STM

newtype Buffer a = Buffer (TVar [a])

newBuffer :: STM (Buffer a)
newBuffer = do
    b <- newTVar []
    return (Buffer b)
```



## Example: Buffer Revisited (2)

```
readBuffer :: Buffer a -> STM a
readBuffer (Buffer b) = do
    xs <- readTVar b
    case xs of
        []       -> retry
        (x : xs') -> do
            writeTVar b xs'
            return x

writeBuffer :: Buffer a -> a -> STM ()
writeBuffer (Buffer b) x = do
    xs <- readTVar b
    writeTVar b (xs ++ [x])
```



## Example: Buffer Revisited (3)

The main program and code for readers and writers can remain unchanged, except that STM operations must be carried out ***atomically***:

```
main = do
    b <- atomically newBuffer
    forkIO (writer b)
    forkIO (writer b)
    forkIO (reader b)
    forkIO (reader b)
    ...
```

## Example: Buffer Revisited (4)

```
reader :: Buffer Int -> IO ()
reader n b = rLoop
    where
        rLoop = do
            x <- atomically (readBuffer b)
            when (x > 0) $ do
                putStrLn (n ++ ": " ++ show x)
                rLoop
```



## Composition (1)

`STM` operations can be ***robustly composed***. That's the reason for making `readBuffer` and `writeBuffer` `STM` operations, and leaving it to client code to decide the scope of atomic blocks.

Example, sequential composition: reading two consecutive elements from a buffer `b`:

```
atomically $ do
    x1 <- readBuffer b
    x2 <- readBuffer b
    ...
```



## Composition (2)

Example, composing alternatives: reading from one of two buffers `b1` and `b2`:

```
x <- atomically $
        readBuffer b1
        `orElse` readBuffer b2
```

The buffer operations thus composes nicely. No need to change the implementation of any of the operations!



## Reading


- Koen Claessen. A Poor Man's Concurrency Monad. *Journal of Functional Programming*, 9(3), 1999.
- Wouter Swierstra and Thorsten Altenkirch. Beauty in the Beast: A Functional Semantics for the Awkward Squad. In *Proceedings of Haskell'07*, 2007.
- Tim Harris, Simon Marlow, Simon Peyton Jones, Maurice Herlihy. Composable Memory Transactions. In *Proceedings of PPoPP'05*, 2005
- Simon Peyton Jones. Beautiful Concurrency. Chapter from *Beautiful Code*, ed. Greg Wilson, O'Reilly 2007.