

•

•

•

MGS 2011: FUN Lecture 1

Lazy Functional Programming

Henrik Nilsson

University of Nottingham, UK

•

•

•

•

•

•

•

•

•

•

MGS 2011: FUN Lecture 1 – p.1/40

•

•

•

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.

•

•

•

•

•

•

•

•

•

•

MGS 2011: FUN Lecture 1 – p.2/40

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)

MGS 2011: FUN Lecture 1 – p.3/40

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.)

MGS 2011: FUN Lecture 1 – p.4/40

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.

MGS 2011: FUN Lecture 1 – p.5/40

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.)

MGS 2011: FUN Lecture 1 – p.6/40

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!*

MGS 2011: FUN Lecture 1 – p.7/40

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*.

MGS 2011: FUN Lecture 1 – p.8/40

Evaluation Orders (2)

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

$$\begin{aligned}\underline{\text{main}} &\Rightarrow \underline{\text{sqr } (\text{dbl } (2 + 3))} \\ &\Rightarrow \underline{\text{dbl } (2 + 3)} * \text{dbl } (2 + 3) \\ &\Rightarrow ((\underline{2 + 3}) + (2 + 3)) * \text{dbl } (2 + 3) \\ &\Rightarrow (5 + (\underline{2 + 3})) * \text{dbl } (2 + 3) \\ &\Rightarrow (5 + 5) * \text{dbl } (2 + 3) \Rightarrow 10 * \underline{\text{dbl } (2 + 3)} \\ &\Rightarrow \dots \Rightarrow \underline{10 * 10} \Rightarrow 100\end{aligned}$$

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

MGS 2011: FUN Lecture 1 – p.9/40

Why Normal Order Reduction? (1)

NOR seems rather inefficient. Any use?

- Best possible termination properties. Two important theorems from the λ -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.

MGS 2011: FUN Lecture 1 – p.11/40

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:

$$\begin{aligned}(0/0) + 1 &= \perp + 1 = \perp \\ 1 + (0/0) &= 1 + \perp = \perp\end{aligned}$$

MGS 2011: FUN Lecture 1 – p.12/40

Strict vs. Non-strict Semantics (2)

Consider:

`foo x = 1`

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

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

Thus, NOR results in non-strict semantics.

Note: NOR gave well-defined result, AOR did not.

MGS 2011: FUN Lecture 1 – p1340

Lazy Evaluation (1)

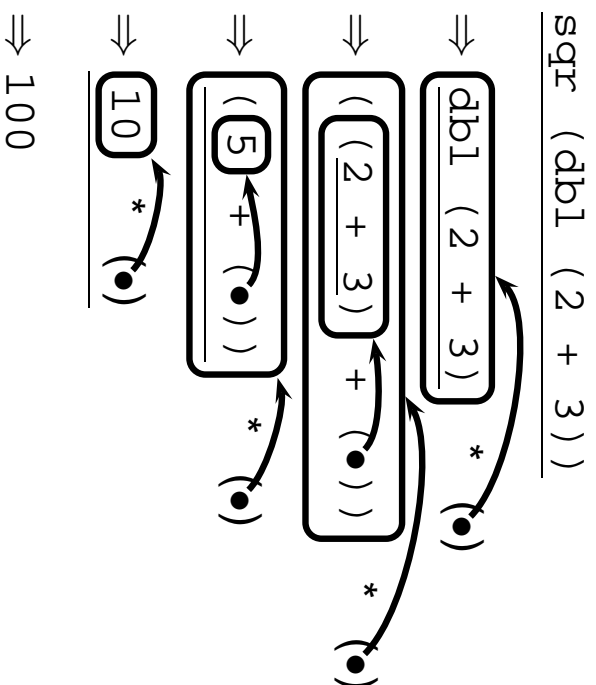
Lazy evaluation is an *technique for implementing NOR* more efficiently:

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

MGS 2011: FUN Lecture 1 – p1440

•

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)

Diagram illustrating the evaluation of the expression $\text{main} \Rightarrow^1 \text{take } 5 \ (\bullet)$ through a series of steps:

- $\text{main} \Rightarrow^1 \text{take } 5 \ (\bullet) \Rightarrow^4 0 : \text{take } 4 \ (\bullet)$
- $0 : \text{take } 4 \ (\bullet) \Rightarrow^6 0 : 1 : \text{take } 3 \ (\bullet) \Rightarrow^8 \dots$
- $0 : 1 : \text{take } 3 \ (\bullet) \Rightarrow^0 0 : 1 : 2 : 3 : 4 : \text{take } 0 \ (\bullet) \Rightarrow [0, 1, 2, 3, 4]$
- $[0, 1, 2, 3, 4] \Rightarrow^1 \boxed{\text{nats}}$
- $\boxed{\text{nats}} \Rightarrow^2 \text{from } 0 \Rightarrow^3 0 : \boxed{\text{from } 1}$
- $0 : \boxed{\text{from } 1} \Rightarrow^5 0 : 1 : \boxed{\text{from } 2} \Rightarrow^7 \dots \Rightarrow^0 0 : 1 : 2 : 3 : 4 : \boxed{\text{from } 5}$

MG5 2011: FUN Lecture 1 – p.17/40

Circular Data Structures (2)

```

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

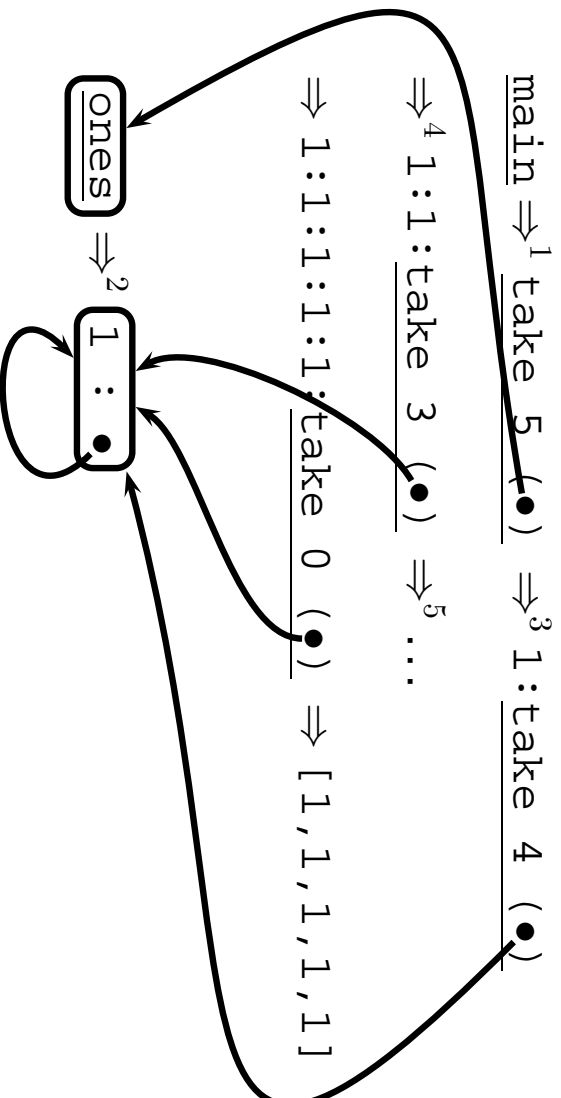
ones = 1 : ones

main = take 5 ones
```

```
ones = 1 : ones
```

```
main = take 5 ones
```

Circular Data Structures (2)



MGS 2011: FUN Lecture 1 – p19/40

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!

MGS 2011: FUN Lecture 1 – p20/40

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
```

MGS 2011: FUN Lecture 1 – p22/40

Circular Programming (3)

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

$$(t', m) = \text{fmr } m \ t$$

Thus:

$$\text{findMinReplace } t = t'$$

where

$$(t', m) = \text{fmr } m \ t$$

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

MGS 2011: FUN Lecture 1 – p22/40

A Simple Spreadsheet Evaluator

	a	b	c
1	c3 + c2		
2	a3 * b2	2	a2 + b2
3	7		a2 + a3

\Rightarrow

	a	b	c
1	37		
2	14	2	16
3	7		21

S

r

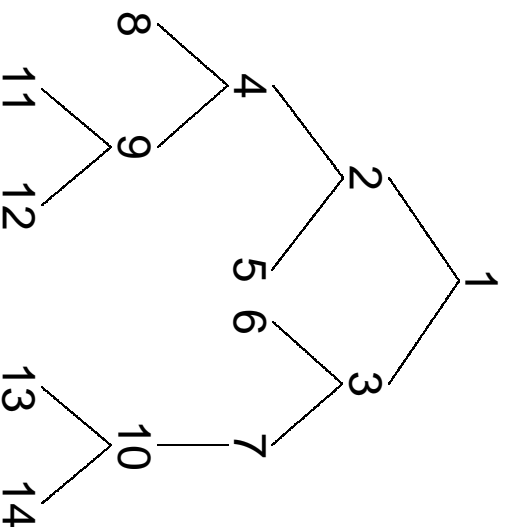
r = array (bounds s)

```
[ ((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:

width $t\ i$ The width of a tree t at level i (0 origin).

label $t\ i\ j$ The j th label at level i of a tree t (0 origin).

MGS 2011: FUN Lecture 1 – p25/40

Breadth-first Numbering (3)

The following system of equations defines breadth-first numbering:

$$\text{label } t\ 0\ 0 = 1 \quad (1)$$

$$\text{label } t\ (i + 1)\ 0 = \text{label } t\ i\ 0 + \text{width } t\ i \quad (2)$$

$$\text{label } t\ i\ (j + 1) = \text{label } t\ i\ j + 1 \quad (3)$$

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

MGS 2011: FUN Lecture 1 – p26/40

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.

MGS 2011: FUN Lecture 1 – p27/40

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.

MGS 2011: FUN Lecture 1 – p28/40

Breadth-first Numbering (6)

bfn :: Tree a -> Tree Integer *Eqns (1) & (2)*
 bfn t = t'

where

$(ns, t') = \text{bfnAux } (1 : ns) \ t$

bfnAux :: [Integer] -> Tree a

-> ([Integer], Tree Integer)

bfnAux ns

Empty

= (ns, Empty)

Eqn (3)

bfnAux (n : ns) (Node t1 _ tr) = ((n + 1) : ns''),
 Node t1' n tr')

where

(ns', t1') = bfnAux ns t1

(ns'', tr') = bfnAux ns' tr

MGS 2011: FUN Lecture 1 – p29/40

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.

MGS 2011: FUN Lecture 1 – p30/40

The Triangulation Problem (1)

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

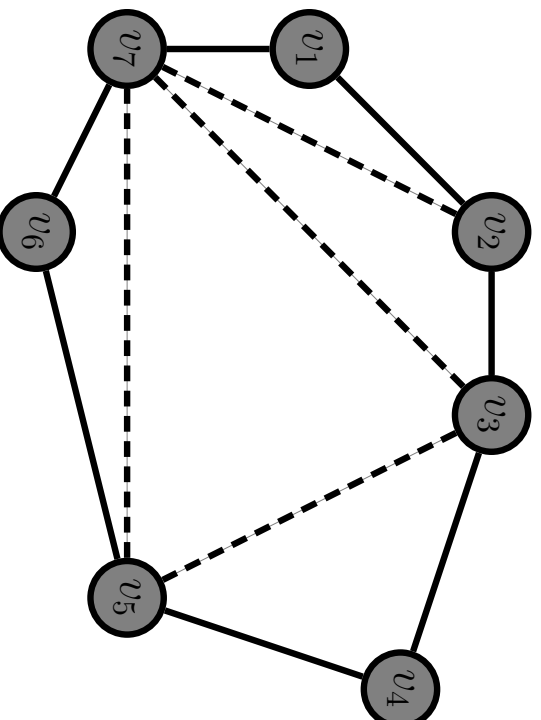
- no two chords cross each other
- the sum of their length is minimal.

We will only consider computing the minimal length.

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

MG5 2011: FUN Lecture 1 – p32/40

The Triangulation Problem (2)



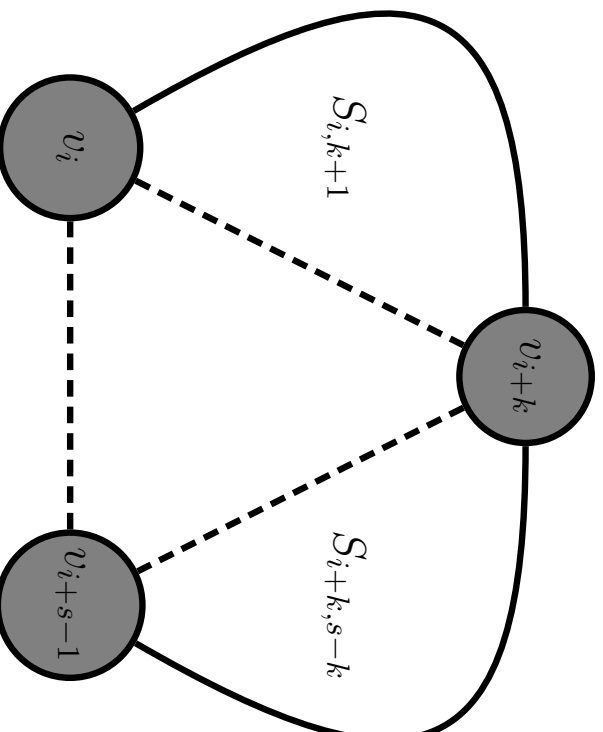
MG5 2011: FUN Lecture 1 – p32/40

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 $v_i, v_{i+1}, \dots, v_{i+s-1}$ (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!

MGS 2011: FUN Lecture 1 – p33/40

The Triangulation Problem (4)



MGS 2011: FUN Lecture 1 – p34/40

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 $s \geq 4$:
- $$C_{is} = \min_{k \in [1, s-2]} \left\{ C_{i, k+1} + C_{i+k, s-k} + D(v_i, v_{i+k}) + D(v_{i+k}, v_{i+s-1}) \right\}$$
- For $s < 4$, $S_{is} = 0$.

MGS 2011: FUN Lecture 1 – p35/40

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
```

MGS 2011: FUN Lecture 1 – p36/40

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.

MGS 2011: FUN Lecture 1 – p37/40

Attribute Grammars (2)

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

MGS 2011: FUN Lecture 1 – p38/40

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

MGS 2011: FUN Lecture 1 – p39/40

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 1 – p40/40

•

•

•

MGs 2011: FUN Lecture 2

Purely Functional Data Structures

Henrik Nilsson

University of Nottingham, UK

•

•

•

•

•

•

•

•

•

MGs 2011: FUN Lecture 2 – p. 1/40

•

•

•

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.

•

•

•

•

•

•

•

•

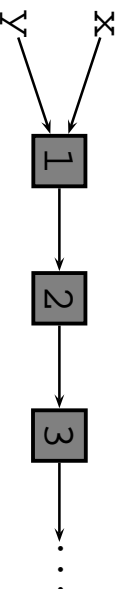
•

•

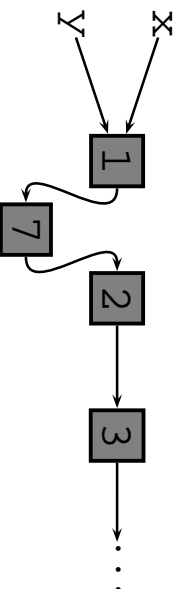
MGs 2011: FUN Lecture 2 – p.2/40

Purely Functional Data structures (2)

Linked list:

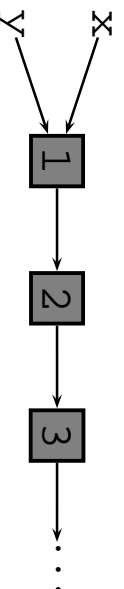


After insert, if ephemeral:

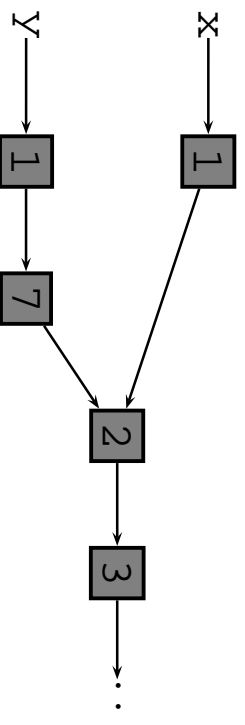


Purely Functional Data structures (3)

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.

MGS 2011: FUN Lecture 2 – p.6/40

Numerical Representations (1)

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

```
data List a =  
  Nil  
  | Cons a (List a)  
  
data Nat =  
  Zero  
  | Succ Nat  
  
tail (Cons _ xs) = xs  
pred (Succ n) = n  
  
append Nil      ys = ys  
append (Cons x xs) ys =  
  Cons x (append xs ys)  
  
plus Zero n      = n  
plus (Succ m) n =  
  Succ (plus m n)
```

MGS 2011: FUN Lecture 2 – p.6/40

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***.

MGS 2011: FUN Lecture 2 – p.7/40

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
```

MGS 2011: FUN Lecture 2 – p.8/40

Positional Number Systems (1)

- A number is written as a **sequence of digits** $b_0b_1 \dots b_{m-1}$, 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:

$$\text{value}(b_0b_1 \dots b_{m-1}) = \sum_0^{m-1} b_i w_i$$

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

MGS 2011: FUN Lecture 2 – p.9/40

Positional Number Systems (2)

- A number is written written in **base** B if $w_i = B^i$ and $D_i = \{0, \dots, B - 1\}$.
- 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.

MGS 2011: FUN Lecture 2 – p.10/40

•
•

Exercise 1: Positional Number Systems

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

• • • • •

MGS 2011: FUN Lecture 2 – p11/140

•
•

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
 $\text{value}(1211) = 1 \cdot 2^0 + 2 \cdot 2^1 + 1 \cdot 2^2 + 1 \cdot 2^3$

• • • • •

MGS 2011: FUN Lecture 2 – p12/140

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 $b_0b_1 \dots b_{m-1}$ 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.

MGS 2011: FUN Lecture 2 – p13/40

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

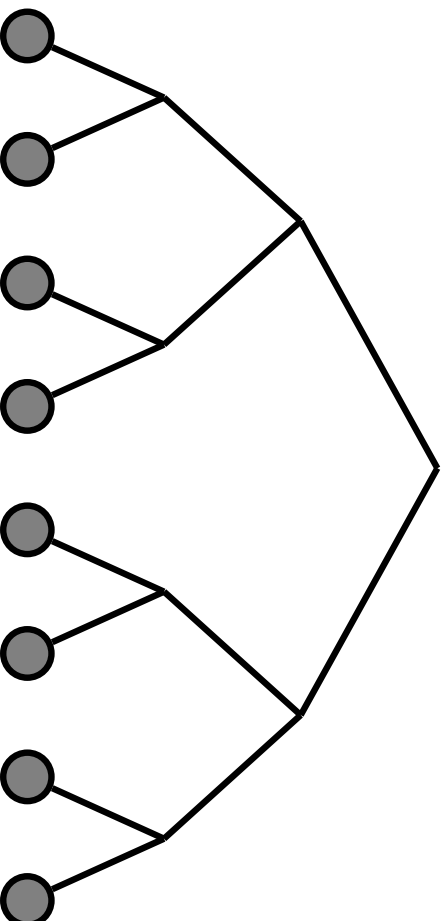
(Balance has to be ensured separately.)

MGS 2011: FUN Lecture 2 – p14/40

•

Example: Complete Binary Leaf Tree

Size $2^3 = 8$:



•

•

•

•

•

•

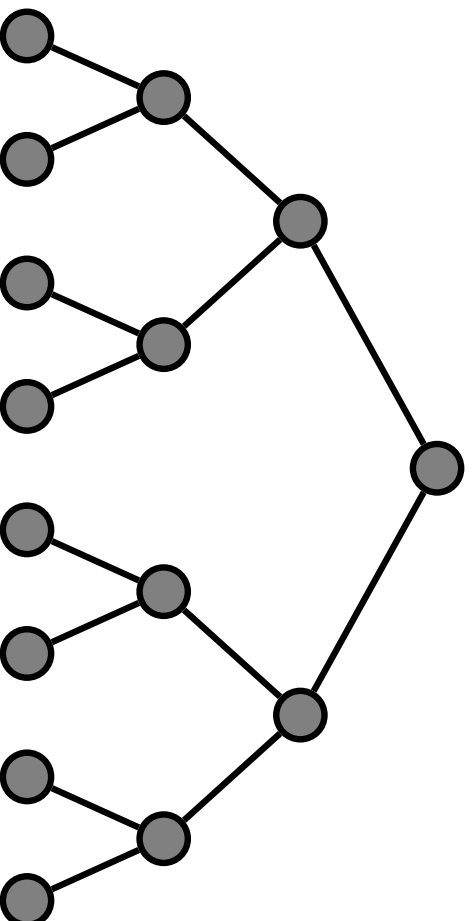
MG5 2011: FUN Lecture 2 – p15/40

•

•

Example: Complete Binary Tree

Size $2^4 - 1 = 15$:



•

•

•

•

•

•

MG5 2011: FUN Lecture 2 – p16/40

•

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:

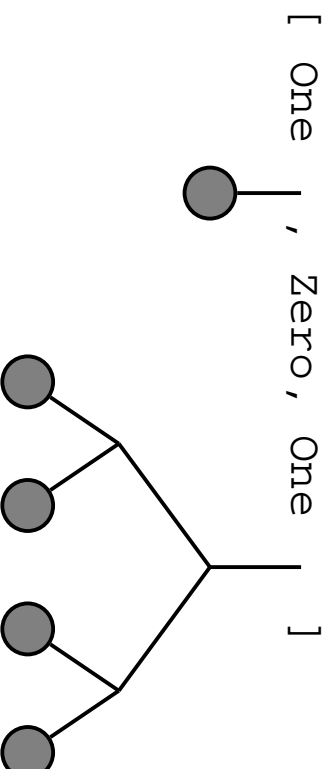
```
data Tree a = Leaf a
              | Node Int (Tree a) (Tree a)
data Digit a = Zero | One (Tree a)
type RList a = [Digit a]
```

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

MGS 2011: FUN Lecture 2 – p17/40

Binary Random Access Lists (2)

Example: Binary Random Access List of size 5:



MGS 2011: FUN Lecture 2 – p18/40

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
```

MGS 2011: FUN Lecture 2 – p19/40

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
```

MGS 2011: FUN Lecture 2 – p20/40

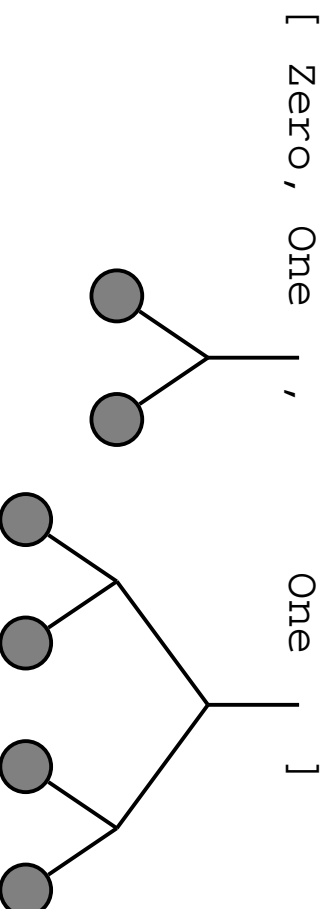
Binary Random Access Lists (5)

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:



Exercise 2: unconstTree

The decrement function on dense binary numbers:

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

Define unconstTree following the above pattern:

```
unconstTree :: Rlist a -> (Tree a, Rlist a)
```

And then head and tail:

```
head :: Rlist a -> a  
tail :: Rlist a -> Rlist a
```

MGS 2011: FUN Lecture 2 – p23/40

Exercise 2: Solution (1)

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

Note: partial operation.

MGS 2011: FUN Lecture 2 – p24/40

Exercise 2: Solution (2)

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

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

MGS 2011: FUN Lecture 2 – p25/40

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.

MGS 2011: FUN Lecture 2 – p26/40

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.

MGS 2011: FUN Lecture 2 – p27/40

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?

MGS 2011: FUN Lecture 2 – p28/40

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.

MGS 2011: FUN Lecture 2 – p29/40

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.

MGS 2011: FUN Lecture 2 – p30/40

Skew Binary Numbers (3)

- Inductive case. Assume n has a unique skew binary representation $b_0b_1 \dots b_{n-1}$
 - 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} .

MGS 2011: FUN Lecture 2 – p32/40

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.

MGS 2011: FUN Lecture 2 – p32/40

Exercise 3: Solution

- 00001, 0002, 0021, 0211

```
inc :: Nat -> Nat
inc (w1 : w2 : ws)
  | w1 == w2 = w1 * 2 + 1 : ws
inc ws      = 1 : ws
```

MGS 2011: FUN Lecture 2 – p33/40

Skew Binary Random Access Lists (1)

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

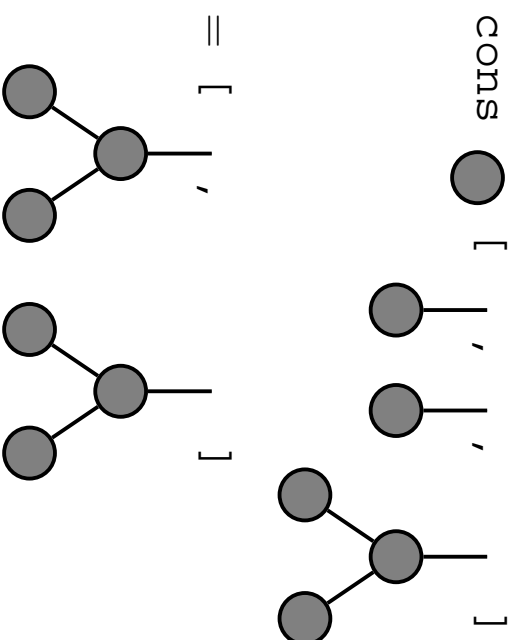
```
empty :: RList a
empty = []
```

```
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)
```

MGS 2011: FUN Lecture 2 – p34/40

Skew Binary Random Access Lists (2)

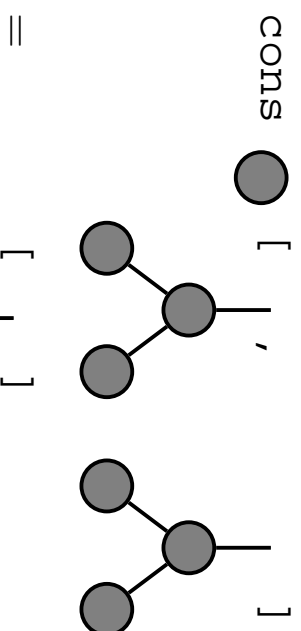
Example: Consing onto list of size 5:



MGS 2011: FUN Lecture 2 – p35f40

Skew Binary Random Access Lists (3)

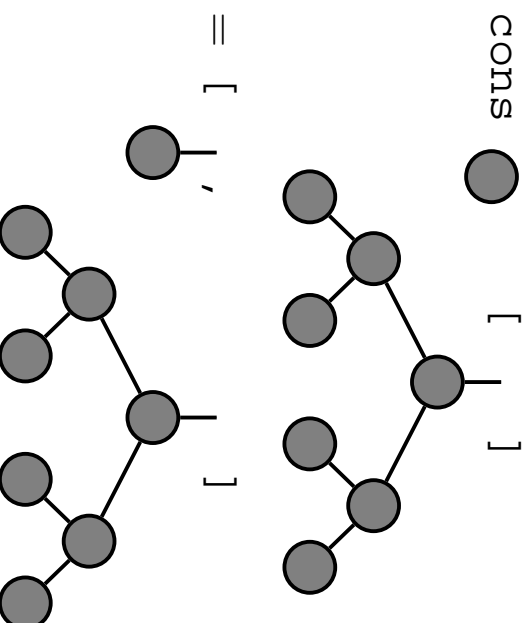
Example: Consing onto list of size 6:



MGS 2011: FUN Lecture 2 – p36f40

Skew Binary Random Access Lists (4)

Example: Consing onto list of size 7:



MGS 2011: FUN Lecture 2 – p37/40

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.

MGS 2011: FUN Lecture 2 – p38/40

•

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
```

MGS 2011: FUN Lecture 2 – p39/40

•

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 2 – p40/40

•

•

•

MGs 2011: FUN Lecture 3

Monads

Henrik Nilsson

University of Nottingham, UK

•

•

•

•

•

•

•

•

•

MGs 2011: FUN Lecture 3 – p.1/52

•

•

•

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

•

•

•

•

•

•

•

•

•

MGs 2011: FUN Lecture 3 – p.2/52

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.

MGS 2011: FUN Lecture 3 – p.3/52

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;  
    void set((int n)) { (x) = (n); }  
}
```

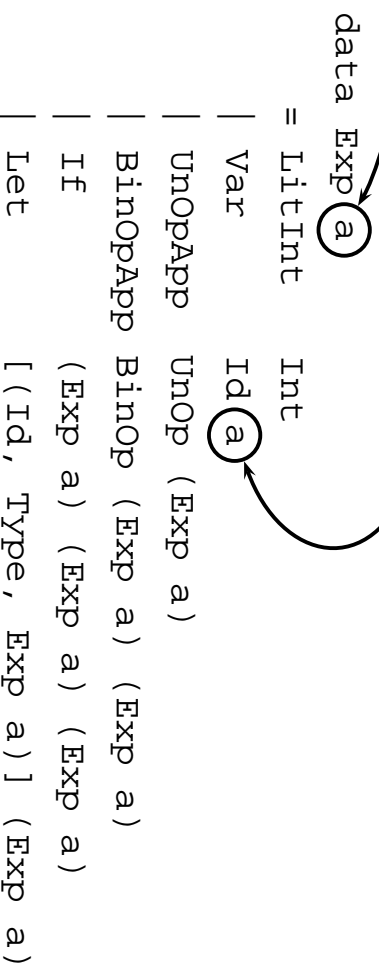
In the body of `set`, the one applied occurrence of

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

MGS 2011: FUN Lecture 3 – p.4/52

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*.



MGS 2011: FUN Lecture 3 – p.6/52

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))
```

MGS 2011: FUN Lecture 3 – p.6/52

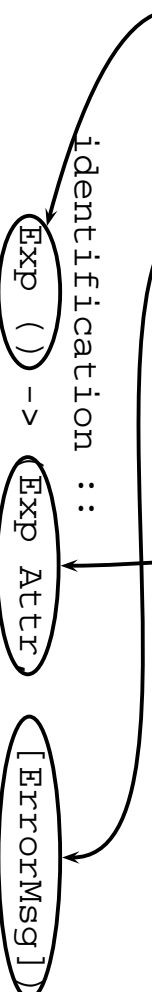
Example: A Compiler Fragment (4)

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.



MGS 2011: FUN Lecture 3 – p.7/52

Example: A Compiler Fragment (5)

Example: Before Identification

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

After identification:

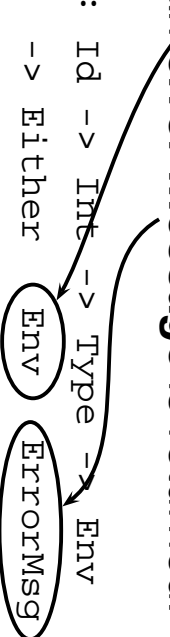
```
Let [ ("x", IntType, LitInt 7) ]  
  (BinOpApp Plus  
    (Var "x" (1, IntType))  
    (LitInt 35))
```

MGS 2011: FUN Lecture 3 – p.8/52

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 ErrorMessage
```



MGS 2011: FUN Lecture 3 – p.9/52

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])
```

MGS 2011: FUN Lecture 3 – p.10/52

•
•

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
```

MGS 2011: FUN Lecture 3 – p11/52

•
•

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**.

MGS 2011: FUN Lecture 3 – p12/52

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

MGS 2011: FUN Lecture 3 – p13/52

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,

MGS 2011: FUN Lecture 3 – p14/52

This Lecture

Pragmatic introduction to monads:

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

MGS 2011: FUN Lecture 3 – p15/52

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
```

MGS 2011: FUN Lecture 3 – p16/52

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)

MGS 2011: FUN Lecture 3 – p17/52

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)

MGS 2011: FUN Lecture 3 – p18/52

•
•

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)
```

• • • • •
MGS 2011: FUN Lecture 3 – p19/52

•
•

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)
```

• • • • •
MGS 2011: FUN Lecture 3 – p20/52

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.

MGS 2011: FUN Lecture 3 – p21/52

Sequencing Evaluations

```
evalSeq :: Maybe Integer  
      -> (Integer -> Maybe Integer)
```

```
      -> Maybe Integer
```

```
evalSeq ma f =
```

```
  case ma of
```

```
    Nothing -> Nothing
```

```
    Just a  -> f a
```

MGS 2011: FUN Lecture 3 – p22/52

•
•
•

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
```

MGS 2011: FUN Lecture 3 – p23/52

•
•
•

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)))
```

MGS 2011: FUN Lecture 3 – p24/52

•
•
•

Aside: Scope Rules of λ -abstractions

The scope rules of λ -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)`

...

• • • • •

MGS 2011: FUN Lecture 3 – p26/52

•
•
•

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)`

• • • • •

MGS 2011: FUN Lecture 3 – p26/52

•
•
•

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)
```

• • • • •

MGS 2011: FUN Lecture 3 – p27/52

•
•
•

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
```

• • • • •

MGS 2011: FUN Lecture 3 – p28/52

•

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
```

•

•

•

•

•

•

•

•

MGS 2011: FUN Lecture 3 – p29/52

•

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 exercise: verify the other cases.

•

•

•

•

•

•

•

•

•

MGS 2011: FUN Lecture 3 – p30/52

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.

MGS 2011: FUN Lecture 3 – p32/p52

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
```

MGS 2011: FUN Lecture 3 – p32/p52

Maybe Viewed as a Computation (3)

Failing computation:

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

MGS 2011: FUN Lecture 3 – p33/52

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))
```

MGS 2011: FUN Lecture 3 – p34/52

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'')
```

MGS 2011: FUN Lecture 3 – p35f52

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?

MGS 2011: FUN Lecture 3 – p36f52

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`.

MGS 2011: FUN Lecture 3 – p37/52

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.)

MGS 2011: FUN Lecture 3 – p38/52

Stateful Computations (3)

Computation of a value without changing the state (For ref.: $S\ a = Int \rightarrow (a, Int)$):

```
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'
```

MGS 2011: FUN Lecture 3 – p39/52

Stateful Computations (4)

Reading and incrementing the state
(For ref.: $S\ a = Int \rightarrow (a, Int)$):

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

MGS 2011: FUN Lecture 3 – p40/52

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')
```

MGS 2011: FUN Lecture 3 – p41/52

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!

MGS 2011: FUN Lecture 3 – p42/52

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**.

MGS 2011: FUN Lecture 3 – p43/52

Monads in Functional Programming

A monad is represented by:

- A type constructor

$M :: * \rightarrow *$

$M\ T$ 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.

MGS 2011: FUN Lecture 3 – p44/52

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
```

MGS 2011: FUN Lecture 3 – p45/52

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)
```

MGS 2011: FUN Lecture 3 – p46/52

Monad laws

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

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

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

MGS 2011: FUN Lecture 3 – p47/52

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.

MGS 2011: FUN Lecture 3 – p48/52

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.:

```
return x >>= f = id x >>= f
               = x >>= f
               = f x

```

MGS 2011: FUN Lecture 3 – p49/52

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

(Additionally, some laws must be satisfied.)

MGS 2011: FUN Lecture 3 – p50/52

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, η, μ) , where $T : \mathcal{C} \rightarrow \mathcal{C}$ is a functor, $\eta : \text{id}_{\mathcal{C}} \rightarrow T$ and $\mu : T^2 \rightarrow T$ are natural transformations.

(Additionally, some commuting diagrams must be satisfied.)

MGS 2011: FUN Lecture 3 – p51/52

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 3 – p52/52

•

•

•



MGs 2011: FUN Lecture 4

More about Monads

Henrik Nilsson

University of Nottingham, UK

•

•

•

•

•

•

•

•

•

•

MGs 2011: FUN Lecture 4 – p. 1/41



•

•

•

This Lecture

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

•

•

•

•

•

•

•

•

•

•

MGs 2011: FUN Lecture 4 – p. 2/41



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.

MGS 2011: FUN Lecture 4 – p.3/41

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
```

MGS 2011: FUN Lecture 4 – p.4/41

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.

MGS 2011: FUN Lecture 4 – p.5/41

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'
```

MGS 2011: FUN Lecture 4 – p.6/41

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
```

MGS 2011: FUN Lecture 4 – p.7/41

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)
```

MGS 2011: FUN Lecture 4 – p.8/41

•
•
•

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
```

• • • • • MGS 2011: FUN Lecture 4 – p.9/41 •

•
•
•

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
```

• • • • • MGS 2011: FUN Lecture 4 – p.10/41 •

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
```

MGS 2011: FUN Lecture 4 – p11/41

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')
```

MGS 2011: FUN Lecture 4 – p12/41

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 ErrorMessage
```

into this:

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

and then `identDefs` from this ...

MGS 2011: FUN Lecture 4 – p13/41

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
```

MGS 2011: FUN Lecture 4 – p14/41

The Compiler Fragment Revisited (3)

into this:

```
identDefsd 1 env [] = return ([], env)
identDefsd 1 env ((i,t,e) : ds) = do
  e' <- identAuxD 1 env e
  env' <- enterVard i 1 t env
  (ds', env'') <- identDefsd 1 env' ds
  return ((i,t,e') : ds', env'')
```

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

MGS 2011: FUN Lecture 4 – p15/41

The Compiler Fragment Revisited (4)

Compare with the “core” identified earlier!

```
identDefs 1 env [] = ([], env)
identDefs 1 env ((i,t,e) : ds) =
  ((i,t,e') : ds', env'')
```

where

```
e' = identAux 1 env e
env' = enterVar i 1 t env
(ds', env'') = identDefs 1 env' ds
```

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

MGS 2011: FUN Lecture 4 – p16/41

•

The List Monad

Computation with many possible results,
“nondeterminism”:

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

Example:

Result:

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

•

•

•

•

•

•

•

•

•

MGS 2011: FUN Lecture 4 – p17/41

•

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
```

•

•

•

•

•

•

•

•

•

MGS 2011: FUN Lecture 4 – p18/41

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
```

MGS 2011: FUN Lecture 4 – p19/41

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))
```

MGS 2011: FUN Lecture 4 – p20/41

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.

MGS 2011: FUN Lecture 4 – p22/41

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*.

MGS 2011: FUN Lecture 4 – p22/41

Monad Transformers in Haskell (1)

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

$$T :: (* \rightarrow *) \rightarrow (* \rightarrow *)$$

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

$$\text{lift} :: M\ a \rightarrow T\ M\ a$$

MGS 2011: FUN Lecture 4 – p23/41

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
```

MGS 2011: FUN Lecture 4 – p24/41

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
```

MGS 2011: FUN Lecture 4 – p25/41

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
```

MGS 2011: FUN Lecture 4 – p26/41

•

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)
```

MGS 2011: FUN Lecture 4 – p27/41

•

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))
```

MGS 2011: FUN Lecture 4 – p28/41

•
•
•

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
```

• • • • • MGS 2011: FUN Lecture 4 – p29/41

•
•
•

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
```

• • • • • MGS 2011: FUN Lecture 4 – p30/41

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`.

MGS 2011: FUN Lecture 4 – p31/41

Exercise 2: Solution

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

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

MGS 2011: FUN Lecture 4 – p32/41

•
•
•

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'
```

• • • • •

MGS 2011: FUN Lecture 4 – p33/41

•
•
•

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))
```

• • • • •

MGS 2011: FUN Lecture 4 – p34/41

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
```

MGS 2011: FUN Lecture 4 – p35/41

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)
```

MGS 2011: FUN Lecture 4 – p36/41

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)
```

MGS 2011: FUN Lecture 4 – p37/41

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))
```

MGS 2011: FUN Lecture 4 – p38/41

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.

MGS 2011: FUN Lecture 4 – p39/41

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

MGS 2011: FUN Lecture 4 – p40/41

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.

•

•

•



MG5 2011: FUN Lecture 5

Concurrency

Henrik Nilsson

University of Nottingham, UK

•

•

•

•

•

•

•

•

•

•

MG5 2011: FUN Lecture 5 – p. 1/36



•

•

•

This Lecture

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

•

•

•

•

•

•

•

•

•

•

MG5 2011: FUN Lecture 5 – p. 2/36



-
-
-

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.

MGS 2011: FUN Lecture 5 – p.3/36

-
-
-

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**.

MGS 2011: FUN Lecture 5 – p.4/36

•
•
•

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)
```

MGS 2011: FUN Lecture 5 – p.5/36

•
•
•

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)
```

MGS 2011: FUN Lecture 5 – p.6/36

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
```

MGS 2011: FUN Lecture 5 – p.7/36

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)
```

MGS 2011: FUN Lecture 5 – p.8/36

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)
```

MGS 2011: FUN Lecture 5 – p.9/36

Example: Concurrent Processes

```
p1 :: CM ()      p2 :: CM ()      p3 :: CM ()
p1 = do
  cPrint 'a'      cPrint '1'      cFork p1
  cPrint 'b'      cPrint '2'      cPrint 'A'
  ...            ...
  cPrint 'j'      cPrint '0'      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.)

MGS 2011: FUN Lecture 5 – p.10/36

•

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 ()      p2 :: CM ()      p3 :: CM ()
p1 = do          p2 = do          p3 = do
  cPrint 'a'     cPrint '1'       cFork p1
  cPrint 'b'     undefined       cPrint 'A'
  ...           ...              cFork p2
  cPrint 'j'     cPrint '0'       cPrint 'B'

main = print (runCM p3)
```

Result: aAbc1Bd*** Exception:
Prelude.undefined

Any Use?

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

MGS 2011: FUN Lecture 5 – p13/36

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
```

MGS 2011: FUN Lecture 5 – p14/36

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.

MGS 2011: FUN Lecture 5 – p15/36

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
```

MGS 2011: FUN Lecture 5 – p16/36

•
•
•

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!"
```

MGS 2011: FUN Lecture 5 – p17/36

•
•
•

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)
```

MGS 2011: FUN Lecture 5 – p18/36

•

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
```

• MGS 2011: FUN Lecture 5 – p19/36

•

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 [])
```

• MGS 2011: FUN Lecture 5 – p20/36

•

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
```

•

•

•

•

•

•

•

•

•

MGS 2011: FUN Lecture 5 – p23/36

•

Compositionality? (2)

What about this?

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

•

•

•

•

•

•

•

•

•

MGS 2011: FUN Lecture 5 – p24/36

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!

MGS 2011: FUN Lecture 5 – p25/36

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.

MGS 2011: FUN Lecture 5 – p26/36

Software Transactional Memory (2)

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

MGS 2011: FUN Lecture 5 – p27/36

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?)

MGS 2011: FUN Lecture 5 – p28/36

-
-

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
```

-
-
-
-
-
-
-
-
-

MGS 2011: FUN Lecture 5 – p29/36

-
-

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)
```

-
-
-
-
-
-
-
-
-

MGS 2011: FUN Lecture 5 – p30/36

•

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])
```

MGS 2011: FUN Lecture 5 – p31/36

• • • • • • • • • •

•

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)
  ...
```

MGS 2011: FUN Lecture 5 – p32/36

• • • • • • • • • •

•
•

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
```

• • • • • MGS 2011: FUN Lecture 5 – p33/36

•
•

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  
  ...
```

• • • • • MGS 2011: FUN Lecture 5 – p34/36

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!

MGS 2011: FUN Lecture 5 – p35/36

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.

MGS 2011: FUN Lecture 5 – p36/36