UppLysning

24 September 1997

Haskell and Lazy Functional Programming

Henrik Nilsson

Outline

- What is a functional language?
- Evaluation order and laziness.
- A brief introduction to Haskell.
- Lazy programming techniques.
  - Infinite data structures.
  - Dynamic programming.
  - Circular programming.
- Pure functional I/O.
- Problems.
- Monads.
  - What is a monad?
  - Monadic I/O.
  - Imperative state.

What is a functional language?

The *imperative languages:*

- Implicit state.
- Computation through execution of a sequence of side-effecting statements.

The *declarative languages*:

- *No* implicit state.
- A program may be regarded as a theory.
- Computation can then be seen as deduction from this theory.
- Examples:
  - Logic programming.
  - Functional programming.

What is a functional language? (cont.)

Less formally:

- Algorithm = Logic + Control
- Declarative programming emphasizes the logic (‘what’) rather than the control component (‘how’).
- Examples:
  - Resolution.
  - Lazy evaluation.

Distinguishing features of functional programming languages

In practice, what matters is perhaps the *style* of programming.

Characteristic features of the functional style:

- Functions are first class objects.
  - Higher-order functions.
  - Function-valued expressions. E.g. (f o g) (3+4)
  - Lambda-expressions, i.e. expressions with free variables that evaluate to anonymous function values.
  - Currying & partial application.

Distinguishing features ... (cont.)

- All computation is essentially carried out by function application.
  - Recursion.
  - Proper handling of tail-recursive functions.
- Referential transparency.
  - Makes equational reasoning possible.
  - No side-effects; functions are really functions.
  - Static scope; dynamic scope breaks RT.
  - Automatic memory management; a free operation breaks RT.

Distinguishing features ... (cont.)

For pragmatical reasons, one might want to allow some side-effects. Thus:

- *Pure* functional languages: no side-effects whatsoever.
- *Mostly* functional languages: some side-effects, e.g. for I/O.

It is quite common to refer to both flavours as functional languages.

Functional or not?

- Haskell: Pure functional language (lazy).
- ML: Mostly functional language (strict).
  - Side-effects for I/O.
  - Side-effects to *some* data structures.
  - No imperative variables.
- Scheme, Erlang 4.4: Borderline cases.
  - Scheme: Fairly unrestricted side-effects.
  - Erlang 4.4: Side-effects for I/O and process communication.
- Common Lisp, Old Erlang: Function based?
  - Second class functions.
- Old Lisp: Not functional.
  - Dynamic scope.

Evaluation order

Is it natural to evaluate the arguments before calling a function (call-by-value)?

Why, then, are some functions special?

- C: `&&`, `||`, `?:`
- Lisp: `setq`, `cond`.

Could one do without these special functions? Consider using `myIf` instead of `?:` and conditional statements in C:

```
int myIf(int p, int x, int y)
{if (p) return x;else return y;}

int fac(int n)
{ return (myIf (n == 0,
                1,
                n*fac(n-1)));
}
```

Applicative order reduction

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

Many possible reduction orders.

Leftmost, innermost *redex* first is called *applicative order*.

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

This is just *call-by-value*.

Normal order reduction

Outermost, leftmost redex first.

```
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 * ((2 + 3) + (2 + 3)) ⇒ ... ⇒
10 * 10 ⇒ 100
```

Demand driven evaluation, call by need.

Strict vs non-strict semantics

$\perp$, 'bottom', the undefined value.

A function `f` is *strict* in an argument iff:

```
f ⊥ = ⊥
```

E.g. + is strict in both its arguments:

```
(0/0) + 1 = ⊥ + 1 = ⊥
1 + (0/0) = 1 + ⊥ = ⊥
```

What about `foo x = 1`? Applicative order reduction:

```
foo (0/0) ⇒ foo ⊥ ⇒ ⊥
```

`foo` becomes *strict*.

Normal order reduction:

```
foo (0/0) ⇒ 1
```

`foo` becomes non-strict.

Why normal order reduction?

Why bother?

- Better termination properties.
  Two important λ-calculus theorems:
  - Church-Rosser Theorem I:
    'No term has more than one normal form.'
  - Church-Rosser Theorem II:
    'If an expression has a normal form, then normal order reduction will find it.'
- More expressive power, e.g:
  - 'Infinite' data structures.
  - Circular programming.
  - Control issues may often be disregarded.

Thus: NOR $\Rightarrow$ No need for special functions!

Lazy functional languages

Lazy evaluation:

- Implementation technique for NOR.
- Arguments evaluated by demand.
- Arguments evaluated at most once.

```
sqr (dbl (2 + 3)) ⇒

dbl (2 + 3) * (●) ⇒

((2 + 3) + (●)) * (●) ⇒

(5 + (●)) * (●) ⇒

10 * (●) ⇒ 100
```

Introduction to Haskell

Haskell is the major lazy functional language.

Named in honour of the mathematician Haskell B. Curry

Some highlights:

- Pure, lazy, functional language:
  - no (unsafe) destructive updating,
  - functional I/O.
- Syntactical features:
  - equational style,
  - pattern matching with guards,
  - currying & partial application,
  - free declaration order,
  - layout sensitive (if desired).

Introduction to Haskell (cont.)

- Strongly, statically typed language:
  - polymorphic Hindley/Milner system,
  - type inference,
  - extended with type & constructor classes.
- Algebraic (concrete) data types.

Introductory example

```
module Main where

fac 0         = 1
fac n | n > 0 = n * fac (n - 1)
      | n < 0 = error "Bad arg."

-- Note: otherwise = True

main = print (fac 7)

{- Note:

print x = putStrLn (show x)

-}
```

Layout

To avoid explicit bracketing, Haskell permits *layout* to be used.

The use of the layout rules is voluntary.

Declaration lists are enclosed in ‘`{`’ and ‘`}`’, declarations separated by ‘`;`’. E.g.:

```
let { decl₁; decl₂; ... } in exp
```

The *layout* (or *off-side*) *rule* takes effect when ‘`{`’ omitted after `where`, `let` or `of`:

- Indentation of next lexeme remembered, ‘`{`’ inserted.
- Subsequent lines:
  - indented more – nothing inserted
  - same indent – ‘`;`’ inserted
  - indented less – ‘`}`’ inserted.

Layout (cont.)

Example:

```
f x = g x + h x
      where
      -- Local definitions
        g x = 2 * x + head [1,2,
              3,4]
        h x = 10 * x
main = print (f 7)
```

has the same meaning as:

```
f x = g x + h x
      where {
      -- Local definitions
        g x = 2 * x +
              head [1,2,3,4];
        h x = 10 * x
      }
main = print (f 7)
```

Naming

Six kinds of names:

- variables
- constructors
- type variables
- type constructors
- type classes
- module names.

Constraints:

1. Variables and type variables start with a small letter; all the other with a capital.
2. Constructor operators start with ‘:’, the rest are variable operators.
3. Type constructors and class names must be distinct within the same scope.

Algebraic data types and pattern matching

```
data Colour = Red | Green | Blue
data Stack a = Stack [a]
data Tree a = Leaf a
            | Tree a :^: Tree a

isBlue Blue = True
isBlue _    = False

height :: Tree a -> Int
height (Leaf _)     = 0
height (t1 :^: t2) =
    1 + max (height t1) (height t2)

Stack s1 +++ Stack s2 =
    Stack (s1 ++ s2)
```

Note how the lexical distinction between variables and constructors helps when reading the code above.

Lambda abstractions

Lambda abstractions denote functional values. \ ... -> ... may be regarded as a kind of constructor for the function type.

Lambda abstractions are sometimes referred to as anonymous functions.

```
\x -> x + 1
\x -> \y -> x + y
\x y -> x + y
\(x,y) -> x + y
```

The equations

```
inc x = x + 1
add x y = x + y
```

are really shorthand for

```
inc = \x -> x + 1
add = \x y -> x + y
```

Currying and partial application

Currying is when functions of more than one argument is regarded as nested lambda-abstractions, and application is performed one argument at a time.

```
f x y = exp
≡ f = \x y -> exp
≡ f = (\x -> (\y -> exp))
```

Permits defining new functions by partial application.

```
add 1 ≡ (\x -> (\y -> x + y)) 1
≡ (\y -> (x + y)[x⟶1])
≡ (\y -> 1 + y)
```

Thus:

```
inc = add 1
```

Currying is a natural, convenient and important syntactical device.

Currying and partial application (cont.)

Further examples:

```
foldr (*) z []      = z
foldr (*) z (x:xs) =
    x * foldr (*) z xs

sum = foldr (+) 0

xs ++ ys = foldr (:) ys xs

concat = foldr (++) []

map f = foldr ((:) . f) []
```

Note: Placing an operator in round brackets means that it can be used as a normal function/ variable. E.g. `a + b` ≡ `(+) a b`

Currying and partial application (cont.)

```
foldr (*) z []      = z
foldr (*) z (x:xs) =
    x * foldr (*) z xs
sum = foldr (+) 0
```

```
sum [1,2] ≡ sum (1:2:[])
⇒ foldr (+) 0 (1:2:[])
⇒ 1 + foldr (+) 0 (2:[])
⇒ 1 + (2 + foldr (+) 0 []))
⇒ 1 + (2 + 0)
⇒ 3
```

Parametric polymorphism

Polymorphism means ‘having many shapes’.

Two main kinds:

- *Parametric* polymorphism.
  - *One* function can have many types.
  - Expressed through parameterized types.
  - Example: `length` (of a list).
- *Ad hoc* polymorphism, or overloading.
  - *Different* functions sharing *one name*.
  - Example: +

Some examples of parametric polymorphism:

```
length :: [a] -> Int
(++) :: [a] -> [a] -> [a]
map :: (a -> b) -> [a] -> [b]
```

Parametric polymorphism (cont.)

Parametric polymorphism allows frequent code reuse while preserving strong, static typing.

No mean feat. Consider:

- Pascal.
  - Strong, static typing (almost).
  - But one version of `map` must be created for each type at which it is used: `mapIntInt`, `mapIntDouble`, ...
- Lisp.
  - Only one `map` needed.
  - But no static typing.
- C.
  - One version of `map`, provided the type-system is bypassed through `void*`.

Type Classes

Dealing with *ad hoc polymorphism/overloading* in a systematic and user extensible way.

Instances of overloading:

- Literals, e.g. `1`, `2`, `3`...
- Numeric operators, e.g. `+`, `-`, `*`...
- Equality.
- The `show` function.

Basic idea:

A *type class* is a set of types on which some operations associated with the class are defined. Cf. SML's eq-types, Java interfaces.

Permits overloading on the return type. `fromInteger`: takes an `Integer` to *any* numeric type.

Defining Type Classes

What is the type of the following function?

```
x `elem` []       = False
x `elem` (y:ys) = x == y ||
                    (x `elem` ys)
```

Is it `a->[a]->Bool`?

No, we should not expect it to work on lists of functions for instance.

However, we would expect it to work for all types on which == is defined, i.e. for *instances* of the class `Eq`:

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

== has the type `(Eq a)=>a->a->bool`.

`Eq a` is a *constraint* on a, called a *context*.

Defining Type Classes (cont.)

We may now give the proper type to `elem`:

```
elem :: (Eq a) => a->[a]->Bool
```

The next step is to declare instances of the class:

```
instance Eq Int where
  x == y = intEq x y

instance Eq Float where
  x == y = floatEq x y
```

The definition of == is called *method*.

Defining Type Classes (cont.)

What about types built from other types?

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

instance (Eq a) => Eq (Tree a) where
    Leaf a          == Leaf b = a == b
    (Branch l1 r1) == (Branch l2 r2) = (l1 == l2) &&
                                       (r1 == r2)
    _               == _              = False
```

Default Methods and Class Inclusion

Real definition of `Eq` class:

```
class Eq a where
  (==), (/=) :: a->a->Bool
  x /= y = not (x == y)
```

Instances of `Eq` thus only have to define ==.

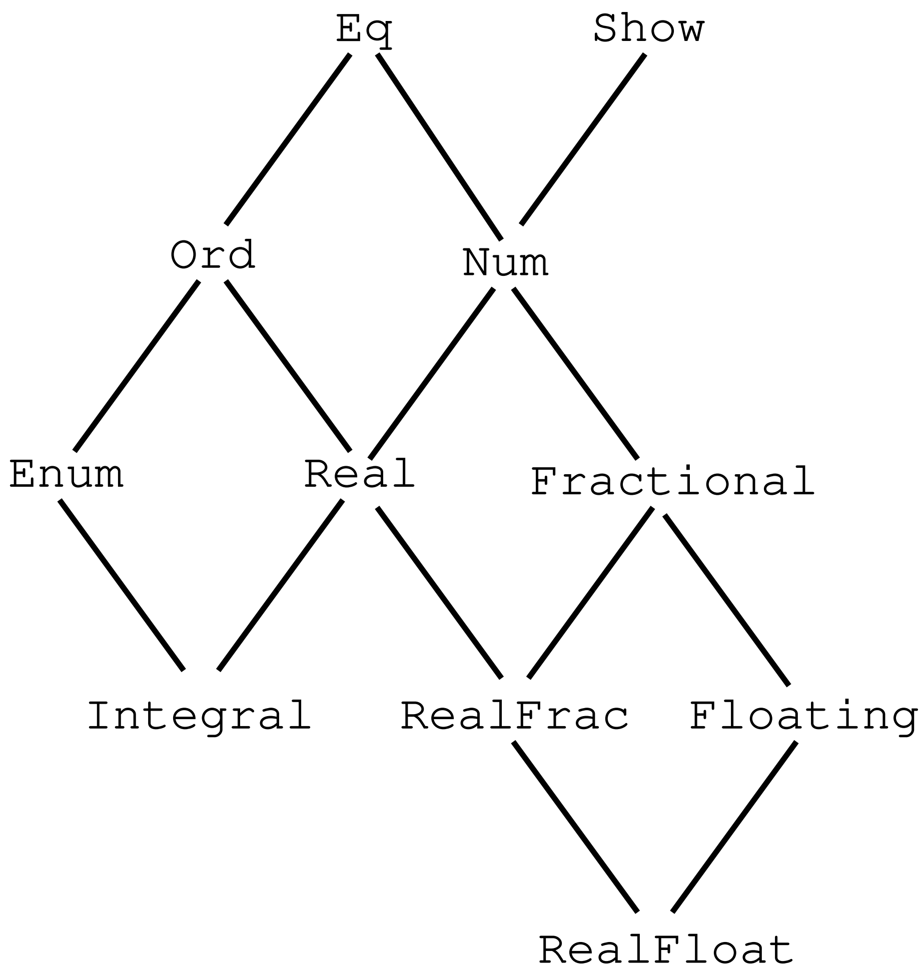
Suppose we define a class `Ord` with the operations <, <=, >=, >. Anything in `Ord` must obviously also be in `Eq`:

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

`Eq` is superclass of `Ord`. `Ord` inherits all operations of `Eq`.

Multiple inheritance is no problem.

Part of the Haskell Class Hierarchy



Implementation of type classes

Suppose we did not have overloading. What to do then?

Take out overloaded function as extra arg.!

```
elem::(a->a->Bool)->a->[a]->Bool
elem eq x []     = False
elem eq x (y:ys) = eq x y ||
                   (x `elem` ys)

main = elem intEq 1 [1,2,3]
```

This is basically what is done in Haskell.

Thus, overloading incurs a substantial overhead!

A program can sometimes be speeded up by inserting explicit type constraints.

Programming with infinte data structures

An example involving an infinite list.

```
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
```

Programming with infinite data structures (cont.)

main $\Rightarrow^{1}$ take 5 (•) $\Rightarrow^{4}$ 0:take 4 (•) $\Rightarrow^{6}$

0:1:take 3 (•) $\Rightarrow^{8}$ ... $\Rightarrow$ 0:1:2:3:4:take 0 (•) $\Rightarrow$

[0,1,2,3,4]

nats $\Rightarrow^{2}$ from 0 $\Rightarrow^{3}$ 0:from 1 $\Rightarrow^{5}$ 0:1:from 2 $\Rightarrow^{7}$

... $\Rightarrow$ 0:1:2:3:4:from 5

The sieve of Erathostene

```
module Main (main) where

sieve (x : xs) =
   x : sieve (filter ((/= 0) . (`mod` x)) xs)

primes = sieve [2..]

main = take 100 primes
```

Arrays in Haskell

In imperative languages arrays are typically created and manipulated incrementally.

This would be too costly in a pure imperative language.

Haskell's arrays are therefore *monolithic* – created in one go.

There is a an incremental array updating function as well, however.

The array creation functions are strict in the index set, but elements are evaluated lazily.

This allows arrays to be defined by *recurrence* relations, i.e. in terms of themselves.

This is very useful for *dynamic programming*.

Dynamic Programming

Dynamic programming:

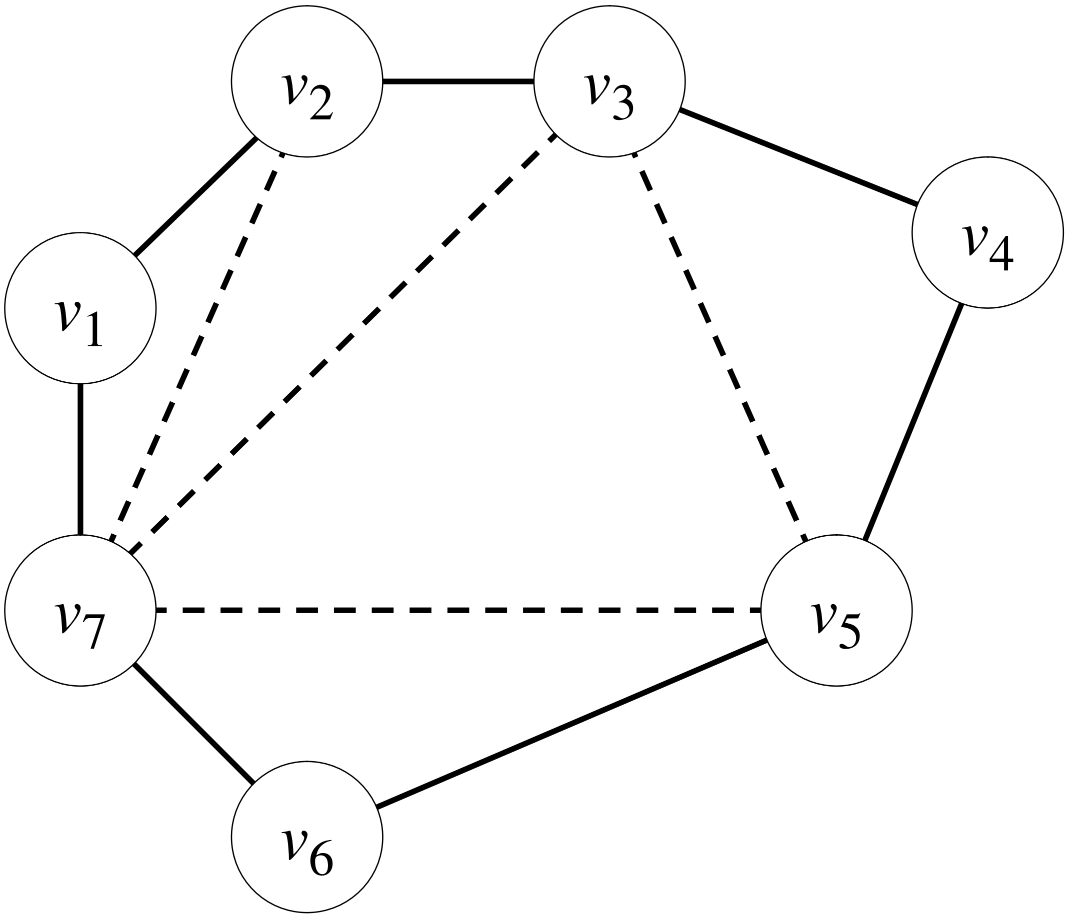
Create a table of all subproblems that ever will have to be solved. Fill in table without regard to whether the solution to that particular subproblem will be needed. Combine solutions to form overall solution.

Laziness and monolithic arrays is a perfect match!

- Might save work.
- We don't have to worry about the *order* in which the table should be filled in.

The Triangulation Problem

Select a set of *chords* that divides a convex polygon into triangles so that no two chords cross each other. The sum of their lengths should be minimal.



We will only compute the minimal length.

The Triangulation Problem (cont.)

The problem may be solved in the following way.

Define the subproblem of size s beginning at vertex v_i, denoted S_{is}, to be the minimum triangulation of the polygon v_i, $v_{i+1} \ldots v_{i+s-1}$ (counting modulo number of vertices).

Subproblems of size < 4 are trivial.

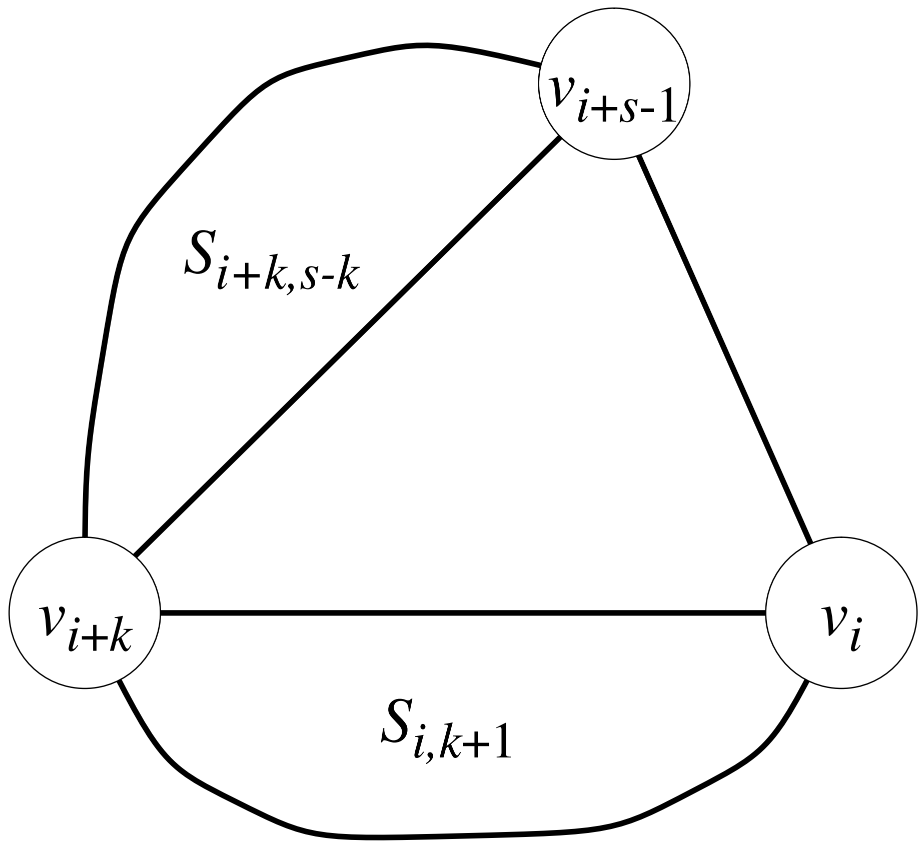
Solving S_{is} is done by picking a k, $1 \leq k \leq s-2$ and solving $S_{i,k+1}$ and $S_{i+k,s-k}$.

We obviously have to try all values for k. Writing the obvious recursive algorithm then gives 3^{s-4} recursive call.

Exponential complexity.

But for n vertices there are only $n(n-4)$ non-trivial subproblems!

Division into subproblems



Formulation as a recurrence

Denote the minimal triangulation cost of S_{is} by C_{is} and tabulate.

Denote the length of a chord between v_p and v_q by $D(v_p, v_q)$. This is 0 for non-chords, i.e. adjacent v_p and v_q.

Then we have:

$$C_{is} = \min\,[C_{i,k+1} + C_{i+k,s-k} + D(v_i,v_{i+k}) + D(v_{i+k},v_{i+s-1})]$$

where k ranges from 1 to s-2.

In a lazy language such as Haskell, this formulation may be entered almost verbatim.

Circular programming
or
Eating the cake before it is baked.

Suppose we have a tree type:

```
data Tree = Leaf Int
          | Tree :^: Tree
```

Then suppose that we would like to replace the leaf integers by the smallest integer in the tree.

How many passes over the tree are required?

ONE!

Circular programming (cont.)

```
fmr :: Int -> Tree -> (Tree, Int)
fmr m (Leaf i)   = (Leaf m, i)
fmr m (l :^: r) = (l' :^: r', min ml mr)
  where
    (l', ml) = fmr m l
    (r', mr) = fmr m r

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

Note: Some care is needed. The cake must be swallowed whole. Putting the teeth in it prematurely results in a *black hole*!

Another example: A Simple Spreadsheet

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

s

$\Longrightarrow$

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

r

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

Any problem with this solution?

Functional I/O

No side effects, so I/O must be performed via top level function.

Pre Haskell 1.3 used the *stream model*. The function `main` had the type:

```
Dialogue = [Response]->[Request]
```

What about *interactive* I/O?
No problem in a lazy language!

Consider a simpler case first:

```
main' :: [Char]->[Char]
main' s = "Enter a number: a = " ++
          sync a ++
          "2 * a = " ++ show (2 * a) ++
          "\n"
          where
              (w:ws) = words s
              a = read w

sync a = seq a []

main = interact main'
```

The Dialogue type

A function of type `Dialogue` consumes a stream of `Responses` and produces a stream of `Requests`.

One `Request` produces exactly one `Response`.

Example requests:

`ReadChan` – return channel as lazy char. list.
`ReadFile` – return file as lazy character list.
`WriteFile` – writes character list to file.
`AppendFile` – appends character list to file.
`GetArgs` – get program arguments.
`GetEnv` – get value of environment variable.
`SetEnv` – set value of environment variable.

The Dialogue type (cont)

Example responses:

`Str s` – successful reading.

`Success` – successful operation.

`Failure err` – operation failed.

Example of stream-based I/O

```
main ~(Success : ~((Str userInput) :
      ~(Success : ~(r4 : _)))) =
   [ AppendChan stdout "please type a filename\n",
     ReadChan stdin,
     AppendChan stdout name,
     ReadFile name,
     AppendChan stdout (case r4 of
                         Str contents -> contents
                         Failure ioerror -> "Error!")
   ] where (name : _) = lines userInput
```

Stream-based I/O

- Stream based I/O is completely general.
- There are, however, pragmatic problems:
  - Input & output done separately.
  - Difficult to understand the connection between them.
  - Easy to mess up the synchronization!
- A different I/O model is thus used in Haskell 1.4.
- Nevertheless, a generalized stream-model is used in the Fudget GUI toolkit.

Problems with functional languages

- Performance.
  - Implementation overhead.
  - Space leaks.
  - Inherent time-complexity penalty when there is no in-place updateable state. E.g. many graph algorithms.
- Debugging.
- Pragmatic aspects.
  - Stream-based I/O.
  - Everything is explicit.

Problems ... (cont.)

The explicitness is actually both a strength and weakness.

Explicitness is good because:

- The code becomes easier to understand and maintain.
- Increased flexibility.

Explicitness is bad because:

- Leads to *plumbing*.
- Plumbing can be error prone.

Problems ... (cont.)

- Too much plumbing tend to obscure what is important. Typical examples:

  - Error reporting.
  - Generation of unique names.
  - Exceptions.
  - Propagation of command line arguments.

Monads

Surprisingly, there is one very general structuring concept that provides a solution to many of these problems:

The Monad

What exactly is a monad?

Can be understood in several ways:

- A mathematical abstraction. Originated in category theory.
- A monadic value denotes a computation.
- Monads is a way of structuring e.g. functional programs that hides ‘plumbing’.

The Monad as a Mathematical Abstraction

A monad is a triple (*M*, *unit*, *bind*) where

- *M* is a type constructor
- *unit* :: *a* -> *M a*
- *bind* :: *M a* -> (*a* -> *M b*) -> *M b*

and the following laws hold:

- Left unit:
  unit a ‘*bind*‘ *b* -> *n* = *n*[*a*/*b*]
- Right unit:
  m ‘*bind*‘ *a* -> *unit a* = *m*
- Associative:
  m ‘*bind*‘ (*a* -> *n* ‘*bind*‘ *b* -> *o*) =
  (m ‘*bind*‘ *a* -> *n)* ‘*bind*‘ *b* -> *o*
  (*a* not free in *o*).

The Monad as a Computation

If M is a monad, then $M\ a$ can be understood as a computation returning a value of type a.

unit is then the trivial computation.

bind represents sequencing of computations.

In this light, we see that the monadic laws simply capture natural properties of sequencing.

Where do monads help?

- Monadic I/O.
- Gives a clean way of integrating *updateable state* into a functional language. (No loss of referential transparency!)
- Flexible structuring mechanism that can be used to:
  - handle exceptions,
  - hide state plumbing,
  - propagate information implicitly,
  - write programs almost in logic-programming style,
  - and much more!

A state monad

Consider numbering the leaves in a tree:

```
data Tree = Leaf Int | Tree :^: Tree

numTree :: Int -> Tree -> (Tree, Int)
numTree s (Leaf _) = (Leaf s, s+1)
numTree s (l :^: r) = (l' :^: r, s'')
  where
    (l', s') = numTree s l
    (r', s'') = numTree s' r
```

Typical example of plumbing. Also typical example where it is easy to make a mistake. E.g. forgetting a "'".

A state monad (cont.)

```
type M a = State -> (a, State)
type State = Int

unit :: a -> M a
unit a = \s -> (a, s)

bind :: M a -> (a -> M b) -> M b
m `bind` k = \s -> let (a, s') = m s in
                   let (b, s'') = k a s' in
                   (b, s'')

newLabel :: M Int
newLabel = \s -> (s, s+1)
```

A state monad (cont.)

Now `numTree` can be rewriting using the state monad to hide the plumbing.

```
numTree :: Tree -> M Tree
numTree (Leaf _) =
  newLabel `bind` \n ->
  unit (Leaf n)
numTree (l :^: r) =
  numTree l `bind` \l' ->
  numTree r `bind` \r' ->
  unit (l' :^: r')
```

Exercise: Prove that the two `numTree` are equivalent!

Further information

- `http://www.haskell.org`
  - *Gentle Introduction to Haskell*
  - *Haskell 1.4 Report*
- ‘Conception, Evolution, and Application of Functional Programming Languages.’ Paul Hudak. *ACM Computing Surveys*, Vol. 21, No. 3, September 1989.
- `http://www.lpac.ac.uk:80/SEL-HPC/Articles/`

Trying yourself

- At Lysator's machines:
  - GHC (Glasgow) and HBC (Chalmers) are available.
  - E.g. `module load hbc`, and off you go!
  - GHC information in Lysator's info tree.
- Or fetch a copy of a system.
  - Find a suitable system via `www.haskell.org`.
  - HUGS is a popular choice. Lightweight interpreter.