Functional Programming

Zoltán Horváth

University Eötvös Loránd, Budapest, Hungary
Department of General Computer Science
E-mail: hz@inf.elte.hu

Literature

- Plasmeijer - Eekelen: Functional Programming and Parallel Graph Rewriting, Addison Wesley, 1993.

- Plasmeijer et al.: Programming in Clean, http://www.cs.kun.nl/~clean

- Peter Achten: Object IO tutorial, http://www.cs.kun.nl/~clean

- Simon Thompson: Haskell - The Craft of Functional Programming, Addison-Wesley, 1999.

1. Introduction

Functional programming languages

- subset of declarative programming languages: computation is defined by set of declarations
- type, class, function definitions, initial expression
- computation means evaluation of the initial expression (rewriting rules)
- mathematical model of computation: λ-calculus (Church, 1932-33, computationally equivalent to Turing machine)

Functional programming style

- specification of problem, refinement of problem are the main concerns

- program components solving subproblems do not cause side-effects

- specification by pre- and postconditions, function compositions given in postcondition correspond to the structure of solution

Evaluation

- evaluation = sequence of rewriting (reduction) steps
- example for a function definition:
  `sqr x = x * x`
  function identifier, formal argument, function body (expression)
- aspects: computability, effectiveness
- Reduction step: substitution (rewriting) of a function application by its definition in the body, until we reach normal form.
- Evaluation strategy: selection order of redexes (reducible expressions), well-known strategies: lazy (function application first), eager or strict (arguments first), parallel.
- Normal form is unique (in confluent rewriting systems), Lazy evaluation order always finds the normal form, if it exists.

Examples

```
inc  x   = x+1
square x = x*x
squareinc x = square (inc x)
fact n = product [1..n]

fact 10
squareinc 7

 strict:
squareinc 7 -> square (inc 7) -> square (7+1)
                    -> square 8     -> 8*8 -> 64
 lazy:
squareinc 7 -> square (inc 7) -> (inc 7) * (inc 7)
                    -> 8 * (inc 7)  -> 8*8 -> 64
```

Characterization of modern purely functional languages

- no destructive assignments
- referential transparency - equational reasoning (same expression means always the same value)
- strongly typed (every subexpression has a static type), type deduction, polymorphism, abstract algebraic data types
- higher order functions (argument or value is a function)
  `twice f x = f (f x))`
- Currying - functions with 1 argument
  `(+) x y` vs. `((+) x) y).`
- recursion
- lazy evaluation and strictness analysis `f x = 0; f (5+1); 2 * (5+1)`

- Zermelo-Fraenkel set-expressions

```
 { x*x \\ x <- [1..] | odd(x) }
```

- pattern matching of arguments

```
fac 0 = 1
fac n | n > 0 = n * fac (n-1)
```

- off-side rule determining scope of identifiers

```
add4 = twice succ
       where
  succ x = x+2
add = ... succ ...
```

- I/O models: i/o stream, monads, unique world

Small Clean programs

```
module test
import StdEnv
Start =
//    5 + 2*3                          //    sum [1..10]
//    reverse (sort [1,6,2,7])         //    1 < 2 && 3 < 4
//    2 < 1 || 3 < 4                   //    [1,2] ++ [3,4,5]
//    and [True, 2<1, 6>5 ]            //    take 3 [1,2,3,4,5]
      map my_abs2 [7,-4,3]
my_abs x
   | x < 0  = ~x
   | x >= 0 = x
my_abs2 x
   | x < 0 = ~x
   | otherwise = x
```

Quadratic equation

```
module quadratic

import StdEnv

qeq :: Real Real Real -> (String,[Real])
qeq a b c
 | a == 0.0       = ("not quadratic",[])
 | delta < 0.0    = ("complex roots",[])
 | delta == 0.0 = ("one root",[~b/2.0*a])
 | delta > 0.0    = ("two roots", [(~b+radix)/(2.0*a),
                               (~b-radix)/(2.0*a)])
 where
     delta = b*b-4.0*a*c
     radix = sqrt delta

Start = qeq 1.0 (-4.0) 1.0
```

Queens

```
module queens
import StdEnv

queens 0 = [[]]
queens n = [ [q:b] \\ b <- queens (n-1),
          q <- [0..7]  | safe q b ]

safe q b = and [not (checks q b i)
                    \\ i <- [0 .. (length b)-1] ]

checks q b i = (q == b!!i) || (abs(q-b!!i)==(i+1))

Start = (length(queens 8),queens 8)
```

Simple IO on console

```
module helloconsole
import StdEnv

Start :: *World -> *World

Start w
    # (console,w) = stdio w
      console     = fwrites "enter your name:\n" console
      (name,console) = freadline console
      console = fwrites ("Hello " +++ name) console
      (_,console) = freadline console
      (ok, nw) = fclose console w
    | not ok = abort "error"
    | otherwise = nw
```

Test environment

```
module functiontest
import funtest, StdClass, StdEnv
Start :: *World -> *World
Start w =  functionTest funs w
dubl :: Int -> Int
dubl x =  x * 2
plus :: Int Int -> Int
plus x y = x+y
fl :: [[Int]] -> [Int]
fl a = flatten a
funs :: [(([String] -> String),[String],String)]
funs = [ (one_arg dubl, ["2"] , "dubl"),
         (two_arg plus, ["2","10"] , "plus"),
         (no_arg "Hello world", [], "Program"),
         (one_arg fl, ["[[1,2,3,4],[],[4]]"],"flatten")]
```

2. Simple Elements of Clean

Pattern matching

```
hd [x:y] = x  // partial
tl [x:y] = y  // partial

fac 0 = 1
fac n
    | n > 0 = n * fac (n-1)  // partial

sum []  = 0
sum [x:xs] = x + sum xs

length [] = 0
length [_:rest] = 1 + length rest
```

Type checking

```
1 + True  // Type error: "argument 2 of +" cannot unify
          // demanded type Int with Bool
length 3  // "argument 1 of length" cannot unify
          // demanded type (a b) | length a with Int
```

Type definitions, annotations

```
// Elementary types: Int, Real, Bool, Char
// Types identifiers starts by uppercase letters

Start :: Int     x:: [Int]    y::[Bool]
Start = 3+4      x=[1,2,3]    y=[True,True,False]

z::[[Int]]                      sum::[Int]->Int
z= [[1,2,3],[1,2]]              sqrt:: Real->Real
```

Annotations

Annotations in type definitions (`!`, `*`, etc.) may specify strict evaluation of arguments or a unique reference to the argument.

Polymorphic type

Type containing type variables,
functions with polymorphic types are called polymorphic functions.

```
length :: [a] -> Int    // a is a type variable,
hd     :: [a] -> a      //   id started by lower letter
```

The functionality of the polymorphic function is not depending on the actual type.

Overloading, "ad hoc polymorphism", classes

There are many instances of +, the functionality of + is depending on the type. The signature is the same.

```
(+) :: a a -> a  // illustration
```

Classes declare overloaded identifiers having the same signature.

```
class (+) infixl 6 a :: !a !a -> a // abstract (+) function
                                   // with strict evaluation
double :: a a -> a | + a           // if (+) has an instance
double n :== n + n                 // then double too
```

Instance definition by systematic substitution:

```
instance + Bool
 where
 (+) :: Bool Bool -> Bool   // instance
 (+) True b = True
 (+) a    b = b
```

Synonyms

- Global constants, evaluated once (run-time), reusable. Optimization: execution time decreased vs. memory usage increased.

```
smallodds =: [1,3 .. 10000]
```

- Type synonyms (replaced in compile time)

```
:: Color :== Int       // Color is the same type as integer
```

- Macros, synonyms of expressions (replaced in compile time)

```
Black :== 1            White :== 0
```

Higher order functions on lists

- filter - selecting elements satisfying a property

```
filter :: (a -> Bool) [a] -> [a]
filter p []     = []
filter p [x:xs]
    | p x       = [  x : filter p xs]
    | otherwise = filter p xs
even x =  x mod 2 == 0
odd = not o even   // odd x = not (even x)
evens  =  filter even [0 .. ]
```

- map - function applied elementwise (length is preserved)

```
map :: (a -> b) [a] -> [b]
map f []     = []
map f [x:xs] = [ f x : map f xs]
odds = map inc evens
```

- foldr - elementwise consumer

```
foldr    :: (.a -> .(.b -> .b)) .b ![.a] -> .b
// foldr :: (  a       a ->  a)  a   [a]  -> a
foldr op e []        = e
foldr op e [x:xs]    = op x (foldr op e xs)
sum = foldr (+) 0         //  sum xs = foldr (+) 0 xs
and = foldr (&&) True
```

- takeWhile - takes while p, dropWhile - drops while p

```
takeWhile p []       = []
takeWhile p [x:xs]
         | p x        = [ x : takeWhile p xs ]
         | otherwise = []
```

Iteration

Iteration of `f` while not `p`:

```
until :: (a -> Bool) (a -> a) a -> a
until p f x
      | p x        = x
      | otherwise  = until p f (f x)
powerOfTwo = until ((<) 1000) ((*) 2) 1  // 1024
```

Example - square root by Newton iteration

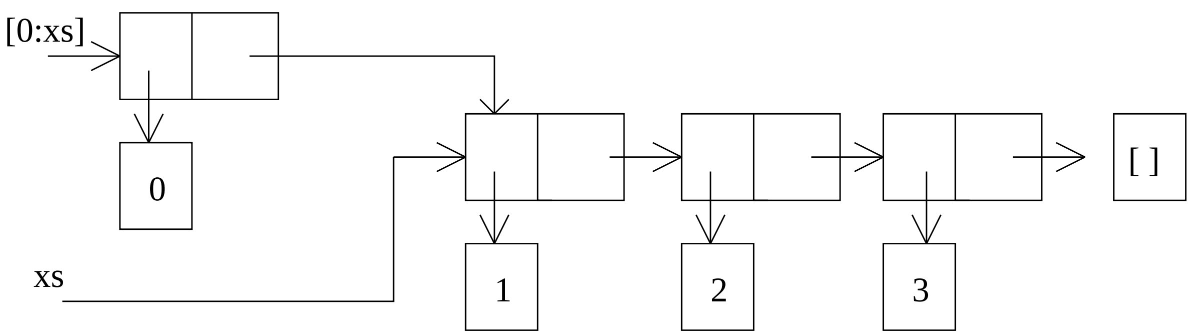
```
sqrtn :: Real Real -> Real
sqrtn x = until goodEnough improve 1.0
      where
         improve y = 0.5 * (y+x/y)
         goodEnough y = (y * y) ~=~ x
         (~=~) a b = abs(a-b) < 0.00001
```

3. Lists

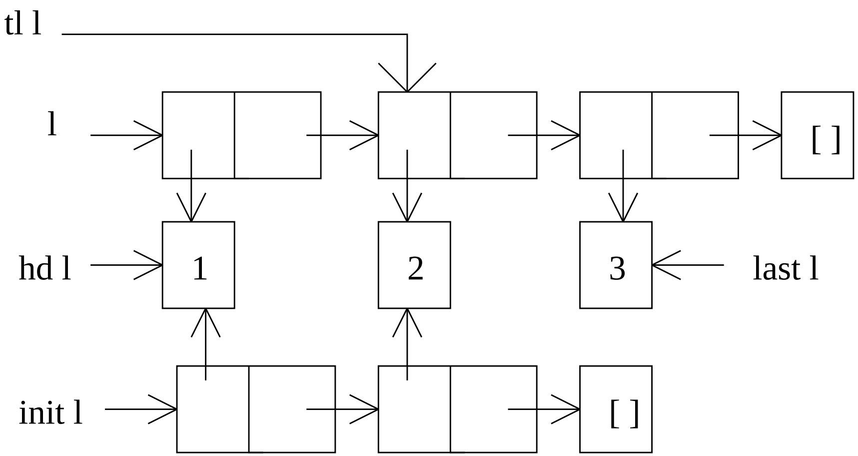
```
[1,2,3*x,length[1,2]] :: [Int]  // enumeration of the elements
[sin, cos, tan]       :: [Real->Real]
[]                    :: a
[3<4,a==5,p&&q]       :: [Bool]
[1,3..12]    [100,80,..]  // arithmetical sequences
```

Representation of list: `xs=[1,2,3]`.

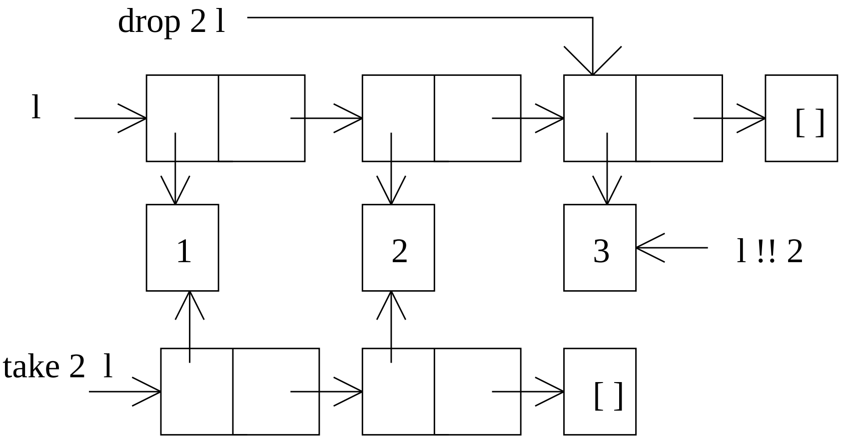
Spine and elements. Extending the list by an element.



Standard functions on lists



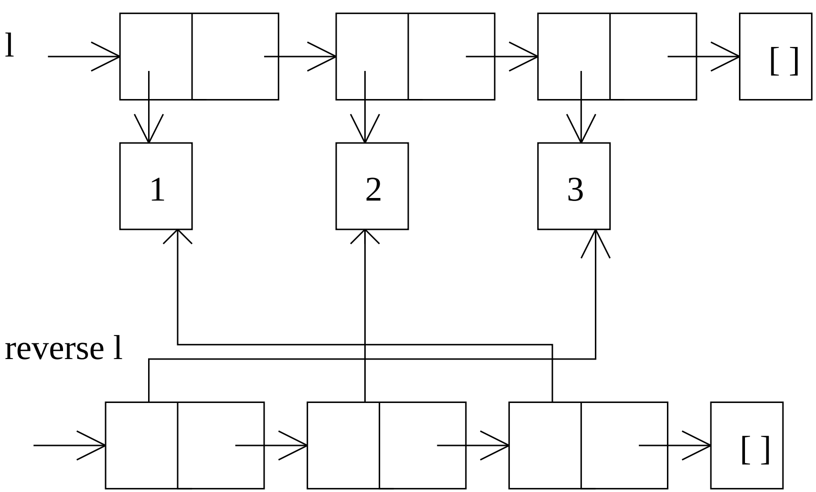

```
hd [a:x]  = a                      tl [a:x] = x
hd [] = abort "hd of []"           tl []  = abort "tl of []"
last [a]  = a                      init [] = []
last [a:tl] = last tl              init [x]  = []
last [] = abort "last of []"       init [x:xs] = [x: init xs]
```




```
(!!) infixl 9 :: [a] Int -> a
(!!) [] _    = subscript_error
(!!) list i = index list i
   where  index [hd:tl] 0 = hd
          index [hd:tl] n = index tl (n - 1)
          index [] _      = subscript_error

take 0 _     = []
take n [a:x] = [a:take (dec n) x]
take n []    = []
```

```
drop n cons=:[a:x]
   | n>0 = drop (n-1) x
         = cons
drop n [] = []
(%) list (frm,to) = take (to - frm + 1) (drop frm list)
reverse list = reverse_ list []
where reverse_ [hd:tl] list   = reverse_ tl [hd:list]
      reverse_ [] list        = list
```



```
(++) infixr 5 :: [a] [a] -> [a]
(++) [hd:tl]  list     = [hd:tl ++ list]
(++) nil      list     = list

length xs = acclen 0 xs
        where
            acclen n [x:xs] = acclen (inc n) xs
            acclen n []     = n

isMember x [hd:tl] = hd==x || isMember x tl
isMember x []    = False

flatten [h:t]    = h ++ flatten t
flatten []               = []
```

```
instance == [a] | Eq a
        where
        (==) [] []     = True
        (==) [] _      = False
        (==) [_:_] [] = False
        (==) [a:as] [b:bs]
                | a == b  = as == bs
                          = False
instance < [a] | Ord a
        where
        (<) []    []      = False
        (<) [] _          = True
        (<) [_:_] []      = False
        (<) [a:as] [b:bs]
                | a < b = True
                | a > b = False
                        = as < bs
```

```
repeat x = cons
      where          cons = [x:cons]
   // repeat 3  is  [3,3..]

iterate f x     = [x:iterate f (f x)]
   // iterate inc 3  is    [3,4,..]

removeAt 0 [y : ys]     = ys
removeAt n [y : ys]     = [y : removeAt (n-1) ys]
removeAt n []           = []
```

Sorting by insertion

```
Insert :: a [a] -> [a] | Ord a
Insert e [] = [e]
Insert e [x:xs]
   | e<=x       = [e,x:xs]
   | otherwise = [x: Insert e xs]

isort :: [a] -> [a] | Ord a
isort [] = []
isort [a:x] = Insert a (isort x)
```

Sorting by merging

```
Merge [] ys = ys
Merge xs [] = xs
Merge [x:xs] [y:ys]
  | x <= y      = [x: merge xs [y:ys]]
  | otherwise   = [y: merge [x:xs] ys]
msort :: [a] -> [a] | Ord a
msort xs
  | len <=1    = xs
  | otherwise = Merge (msort ys) (msort zs)
 where
    ys = take half xs
    zs = drop half xs
    half = len/2
    len = length xs
```

Quicksort / List comprehensions

```
qsort :: [a] -> [a] | Ord a
qsort [] = []
qsort [a:xs] = qsort [x \\ x <- xs | x < a] ++ [a]
               ++ qsort [x \\ x <-xs | x > a]
sieve [p:xs] = [p: sieve [ i \\ i <- xs | i mod p <> 0]]
// take 100 (sieve [2..])
```

Orthogonal generators:

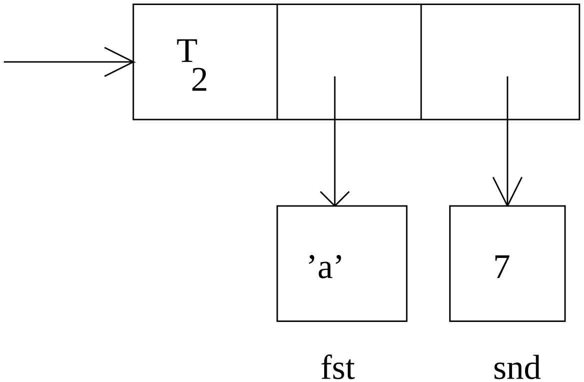
```
[(x,y) \\ x <- [1..4], y <- [1..x] | isEven x ] is
[(2,1),(2,2),(4,1),(4,2),(4,3),(4,4)]
```

Last varies fastest, variable of inner generator is not allowed to use in generators preceding it.

Parallel generators:

```
[ (x,y)  \\ x <- [1..2] & y <- [4..6]] is  [(1,4),(2,5)]
```

4. Tuples and records




```
fst (x,_) = x              snd (_,y) = y


splitAt 0     xs          =        ([],xs)
splitAt _     []          =        ([],[])
splitAt n [x:xs]          =        ([x:xs‘],xs‘‘)
        where  (xs‘,xs‘‘) = splitAt (n-1) xs
zip2 [a:as] [b:bs]        = [(a,b):zip2 as bs]
zip2 as bs                               = []
```

```
average list = s / toReal l
 where
   (s,l) = sumlength list 0.0 0
   sumlength [x:xs] sum l = sumlength xs (sum+x) (l+1)
   sumlength [] sum     l = (sum,l)

search [] s = abort "none"
search [(x,y):ts] s
       | x==s  = y
       | otherwise = search ts s
book = [(1,'a'),(2,'b'),(3,'c')]
// search book 1
```

Records

```
:: Point = {  x        ::  Real
           , y        ::  Real
           , visible ::  Bool
           }
:: Vector = { dx        ::  Real
            , dy        ::  Real
            }
Origo :: Point
Origo = { x = 0.0
        , y = 0.0
        , visible = True
        }
```

Pattern matching on selectors

```
IsVisible :: Point -> Bool
IsVisible {visible = True} = True
IsVisible _                = False

xcoordinate :: Point -> Real
xcoordinate p = p.x

hide :: Point -> Point
hide p = { p & visible = False }

Move :: Point Vector -> Point
Move p v = { p & x = p.x + v.dx, y = p.y + v.dy }
```

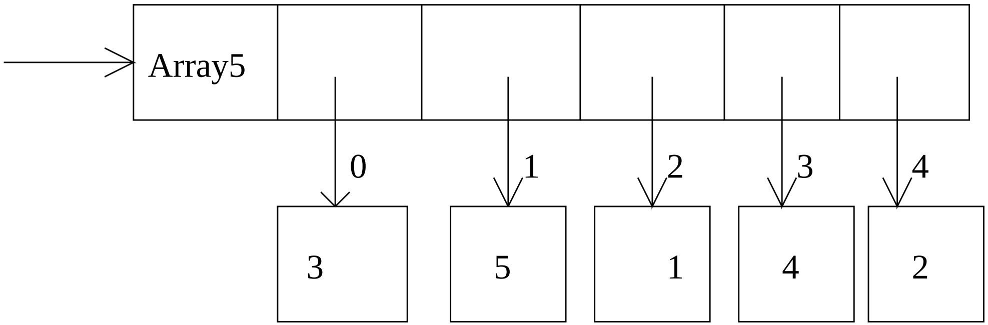
Rational numbers

```
:: Q = { nom :: Int
       , den :: Int
       }
QZero = { nom = 0, den = 1 }    QOne = { nom = 1, den = 1 }
simplify {nom=n,den=d}
  | d == 0 = abort " denominator is 0"
  | d < 0  = { nom = ~n/g, den = ~d/g}
  | otherwise =  { nom = n/g, den = d/g}
 where g = gcd n d
gcd x y = gcdnat (abs x) (abs y)
  where gcdnat x 0 = x
        gcdnat x y = gcdnat y (x mod y)
mkQ n d = simplify { nom = n, den = d }
```

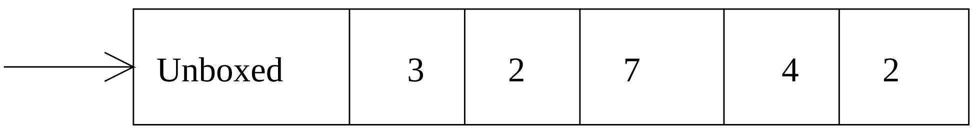
```
instance * Q
 where (*) a b = mkQ (a.nom*b.nom) (a.den*b.den)
instance / Q
 where (/) a b = mkQ (a.nom*b.den) (a.den*b.nom)
instance + Q
 where (+) a b = mkQ (a.nom*b.den+b.nom*a.den) (a.den*b.den)
instance - Q
 where (-) a b = mkQ (a.nom*b.den-b.nom*a.den) (a.den*b.den)
instance toString Q
 where
   toString q = toString sq.nom +++ "/" +++ toString sq.den
              where sq = simplify q
```

Arrays

```
Array5 :: *{Int}
Array5 = {3,5,1,4,2}
```




```
Unboxed :: {#Int}
Unboxed = {3,2,7,4,2}
```



Operations on arrays

Selection:

```
Array5.[1]+Unboxed.[0]
```

Array comprehensions:

```
narray = { e \\ e <- [1,2,3] }
nlist = [ e \\ e <-: Array5 ]
```

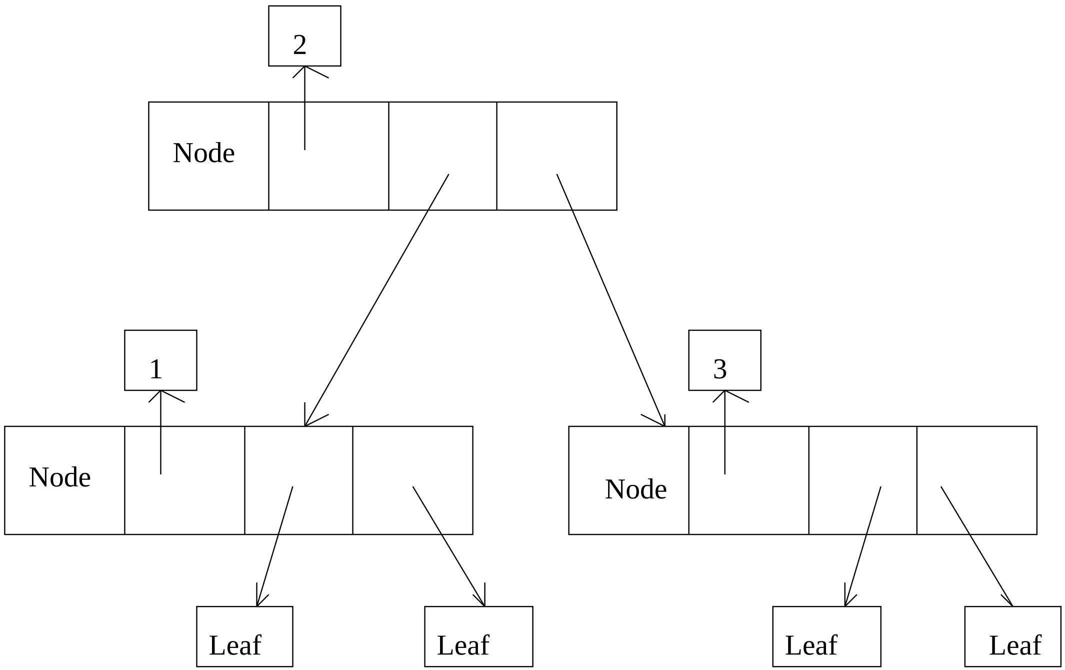
Unique arrays:

```
mArray5 = { Array5 & [3]=3, [4]=4 }
mArray  = { Array5 & [i]=k \\ i <- [0..4] & k <-[80,70..]}
```

5. Algebraic types

Trees (unary type constructor):

```
:: Tree a = Node a (Tree a) (Tree a)
          | Leaf
atree = Node 2 (Node 1 Leaf Leaf) (Node 3 Leaf Leaf)
```



Pattern matching - data constructors

```
depth :: (Tree a) -> Int
depth (Node _ l r) = (max (depth l) (depth r)) +1
depth Leaf = 0
```

MayBe - extension of type value set

```
MayBe a = Just a | Nothing
```

Enumeration type
(nullary type constructor, nullary data constructors):

```
:: Day = Mon | Tue | Wed | Thu | Fri | Sat | Sun
```

Abstract data types

```
definition module stack
:: Stack a
Push :: a (Stack a) -> Stack a
Pop  :: (Stack a) -> Stack a
top  :: (Stack a) ->  a
Empty :: Stack a
```

```
implementation module stack
:: Stack a :== [a]
Push :: a (Stack a) -> Stack a          Push e s = [e:s]
Pop  :: (Stack a) -> Stack a            Pop [e:s] = s
top  :: (Stack a) -> a                  top [e:s] = e
Empty :: Stack a                        Empty = []
------
Start = top (Push 1 Empty)
```

Classes

Classes define signatures of a set of abstract (higher order) functions.

```
class PlusMin a   // class variable
where (+)  infixl 6  :: !a !a -> a
      (-)  infixl 6  :: !a !a -> a
      zero           :: a
```

The type of the instance functions is produced by systematic substitution:

```
instance PlusMin Char
 where (+) :: !Char !Char -> Char
       (+) x y =  toChar (toInt(x) + toInt(y))
       (-) x y =  toChar (toInt(x) - toInt(y)) // negative?
       zero = toChar 0
```

Instantiation is not allowed for type synonyms.

There are derived functions:

```
double :: a -> a | + a
double n :== n+n
```

New classes for set of overloaded functions (singleton classes):

```
import StdClass
class Discriminant a | *, -, fromInt a
discriminant :: a  a a -> a | Discriminant a   // not a class
discriminant a b c = b*b - 4*a*c
```

Insolvable internal overloading:

```
Start = toString (zero + zero)

Start = toString sum
      where sum :: Q                // additional information needed
            sum = zero + zero
```

Default instance:

```
instance zero Q default
  where zero = mkQ 0 1
instance one Q default
  where one = mkQ 1 1
```

Generators are derived from instances of

```
class Enum a | <,+,-,zero,one a:

instance < Q
  where (<) x y = sx.nom * sy.den < sx.den * sy.nom
                where (sx,sy) = (simplify x, simplify y)
Start = [ toString q \\ q <- [zero, mkQ 1 3 .. mkQ 3 2]]
```

Derived class members: instances never defined, but inherited.

Ordering and equality - derived class members:

```
class Eq a | == a
where
  (<>) infix  4 :: !a   !a  ->  Bool | Eq a
  (<>) x y :== not (x == y)
class Ord a     | < a
where
  (>) infix  4 :: !a !a -> Bool | Ord a
  (>) x y  :== y < x
  (<=) infix 4 :: !a !a -> Bool | Ord a
  (<=) x y :== not (y<x)
  (>=) infix 4 :: !a !a -> Bool | Ord a
  (>=) x y :== not (x<y)
  min :: !a !a -> a | Ord a
  min x y  :== if (x<y) x y
  max :: !a !a -> a | Ord a
  max x y  :== if (x<y) y x
```

Higher order types, type constructor classes

Set of higher order classes as instances of the same abstract signature. Examples:

```
map :: (a->b) [a] -> [b]
map f xs [a:as] = [f a:map f as]
:: Tree a = Node a [Tree a]          map f [] = []
mapTree :: (a->b) (Tree a) -> Tree b
mapTree f (Node el ls) = Node (f el) (map (MapTree f) ls)
```

Common abstract signature, classvariable `t` is defined over higher order types (type constructors):

```
class map t :: (a->b) (t a) -> (t b)
instance map []
  where map f l = [ f e \\ e <-l]
instance map Tree
  where map f (Node el ls) = Node (f el) (map (map f) ls)
```

Union and existential types

Union of two types:

```
:: OneOf a b = A a |  B b
:: List_of_Int_or_Char :== [OneOf Int Char]
[(B 'a'),(A '5'),(B 'c')]
```

List of union of all existing types:

```
:: List = E.a : Cons a List | Nil
eList =  Cons 'a' (Cons '5'  (Cons 'c' Nil))
Hd (Cons hd tl) = hd   // type error
Tl (Cons hd tl) = tl
elist = [ Cons 'a', Cons '5', Cons 'c']  //  [] : hd tl
Start = (hd elist, Tl eList)
  // ((Cons 'a'),(Cons '5' (Cons 'c' Nil)))
```

`Hd` : Existentional type variable is not allowed to appear in result type.

Composition of sequence of functions

```
seq :: [t->t] t -> t
seq []     s = s
seq [f:fs] s = seq fs (f s)

:: Pipe a b = Direct ( a->b )
            | E.via : Indirect ( a->via ) (Pipe via b)
ApplyPipe :: (Pipe a b) a -> b
ApplyPipe (Direct f)        x  =  f  x
ApplyPipe (Indirect f pipe) x  = ApplyPipe pipe (f x)
ApplyPipe (Indirect toReal (Indirect exp (Direct toInt))) 7
```

Using existential types

```
:: Point :== (!Int,!Int)
:: Line  :== (!Point,!Point)
:: Rectangle :== (!Point,!Point)
:: Oval :== Rectangle
:: Curve :== (!Oval,!Int,!Int)
:: Drawable = E.a : { state :: a
                    , move  :: Point -> a -> a
                    , draw  :: a Picture -> Picture
                    }
MakeLine :: Line -> Drawable
MakeLine line
 = { state = line
   , move  = \dist line -> line + (dist,dist)
   , draw  = DrawLine }
```

```
MakeCurve :: Curve -> Drawable
MakeCurve curve
 = { state = curve
   , move  = \dist (rect,a1,a2) -> (rect + (dist,dist),a1,a2)
   , draw  = DrawCurve
   }
MakeRectangle :: Rectangle -> Drawable
MakeRectangle ((x1,y1),(x2,y2))
 = { state
     = [ MakeLine ((x1,y1),(x1,y2)), MakeLine((x1,y2),(x2,y2))
       , MakeLine ((x2,y2),(x2,y1)), MakeLine((x2,y1),(x1,y1))]
   , draw  = \s p ->
           foldl (\pict {state,draw} -> draw state pict) p s
   , move  = \d -> map (MoveDrawable d)
        }
MoveDrawable :: Point  Drawable -> Drawable
MoveDrawable p d =: {state,move} = {d & state = move p state}
```

```
:: AlgDrawable
   = Line Line
   | Curve Curve
   | Rect [Line]
   | Wedge [AlgDrawable]
move :: Point AlgDrawable -> AlgDrawable
move p object
 = case object of
    Line line             -> Line (line + (p,p))
    Curve (rect,a1,a2) -> Curve (rect + (p,p),a1,a2)
    Rect lines            -> Rect [line + (p,p) \\ line <- lines]
    Wedge parts           -> Wedge (map (move p) parts)
```

Uniqueness types

Destructive updates preserving referential transparency. Update of an argument (of files, windows, etc.) is allowed if there is just a single reference to the argument at evaluation time.

- Uniqueness is a function property (just a single reference to the argument and/or to the value).

- The type system derives the uniqueness properties of all functions.

- It is possible to reuse unique argument components instead of rebuilding them.

Uniqueness propety may be lost, if the result is multiply referenced.

```
duplicate :: *a -> (*a,*a) // wrong
duplicate x =(x,x)         // result is not unique
```

Uniqueness propagation (outwards):

```
head :: [*a] -> *a  //  spine unique
```

Uniqueness polymorphism: $u \leq * \Leftrightarrow u = *$.

```
id :: u:a -> u:a
class (++) infixr 5  a :: v:[u:a] w:[u:a] -> x:[u:a],
                          [v w x<=u , w<=x]
class (++) infixr 5  a :: v:[.a] w:[.a] -> x:[.a], [w<=x]
                        // equvivalent
```

I/O in Clean

- I/O in Clean uses the *world as value* paradigm,
- environments (external resources, file system, event stream) are passed explicitly as a value to functions,
- I/O programs are functions of type unique `:: *World -> *World`.

Environment passing:

```
fwritec :: Char *File -> *File
AppendAB :: *File -> *File
AppendAB file = fileAB
      where fileA = fwritec 'a' file
            fileAB = fwritec 'b' fileA
```

```
Start  w = CopyFileInWorld w
CopyFileInWorld :: *World -> *World
CopyFileInWorld w = appFiles (CopyFile infn opfn) w
  where
     infn = "source.txt"
     opfn = "copy.txt"
CopyFile :: String String *Files ->*Files
CopyFile infn opfn filesys
   | readok && writeok && closeok =  finalfilesystem
   | not readok       = abort "read error"
   | not writeok      = abort "write error"
   | not closeok      = abort "close error"
 where
  (readok,inf,touchfilesys) = sfopen infn FReadText filesys
  (writeok,outf,nwfilesys) = fopen opfn FWriteText touchfilesys
  copiedfile = LineFileCopy inf outf
  (closeok,finalfilesystem) = fclose copiedfile nwfilesys
```

```
LineFileCopy :: File *File -> *File
LineFileCopy inf outf = LineListWrite (LineListRead inf) outf
LineListRead :: File -> *[String]
LineListRead f
    | sfend  f  = []
    | otherwise = [line:LineListRead filerest]
   where
    (line,filerest) = sfreadline f
LineListWrite :: [String] *File -> *File
LineListWrite [] f      = f
LineListWrite [l:lines] f = LineListWrite lines (fwrites l f)

Start w = CopyFileInWorld w
```

6. Interactive Clean processes

- Object I/O: unique state space, initialization and state-transition functions,
- interactive processes created and closed dynamically
- interactive objects: windows, dialogues, menus, timers, and receivers created and closed dynamically,
- interactive process implement state transition systems,
- state: `PSt l p = {ls :: l, ps :: p, io :: *IOSt l p}`, logical state, public state, I/O state, additional locale state
- logical state and local state is defined by the programmer
- `io` state: the external environment, the current state of all interactive objects of the interactive process.

Interactive Objects

- objects are defined by algebraic data type values containing the *state transitions* of the interactive process
- state transitions are higher order function arguments of the algebraic data types having the type:
  `(PSt .l .p) -> (PSt .l .p)`,
- event handlers of I/O processes on the same processor are interleaved, atomic actions correspond to handling of one event,
- Object I/O system keeps evaluating all interactive processes until each of them has terminated,
- `closeProcess` : closes all current interactive objects from the IO state component and turn it into the final empty IO state.

```
module hellooio
import StdEnv, StdIO
Start :: *World -> *World
Start world
 = startIO NDI Void (snd o openDialog undef hello) [] world
 where
   hello = Dialog "" (TextControl "Hello world!" [])
                  [WindowClose (noLS closeProcess)]
```

```
module hellowin
import StdIO, StdEnv
:: NoState = NoState
Start :: *World -> *World
Start w
 = startIO MDI  0 (openwindow o openmenu) []  w
 where
  openwindow = snd o (openWindow NoState window)
  window = Window "Hello window" NilLS
    [ WindowKeyboard filterKey Able quitFunction,
      WindowMouse  filterMouse Able quitFunction,
      WindowClose quit,
      WindowViewDomain {corner1=zero,corner2={x=160,y=100}},
      WindowLook True look
    ]
```

```
openmenu   = snd o (openMenu NoState file)
file = Menu "File"
     ( MenuItem "Quit" [MenuShortKey 'Q'
                        ,MenuFunction quit]) []
quitFunction _ ps = quit ps
quit (ls,ps) = (ls,closeProcess ps)
look _ _     = drawAt {x=30,y=30} "Hello World"
filterKey key
   = getKeyboardStateKeyState key <> KeyUp
filterMouse mouse
   = getMouseStateButtonState mouse==ButtonDown
```

```
definition module funtest
from StdString import String
import StdEnv, StdIO, conversion
functionTest :: [(([String]->String),[String],String)]
                   *World -> *World
no_arg :: y [String] -> String | toS y
one_arg :: (x->y) [String] -> String
             | fromS x & toS y
two_arg :: (x y -> z) [String] -> String
             | fromS x & fromS y & toS z
three_arg :: (x y z -> w) [String] -> String
              | fromS x & fromS y & fromS z & toS w
```

```
implementation module funtest
import StdEnv,StdIO,conversion

functionTest :: [(([String] -> String),[String],String)]
                     *World -> *World
functionTest [] w = w
functionTest funs w
 # (ids,w) = (openIds (length funs)) w
 = startIO MDI 0 (initialIO funs ids) [] w
initialIO funs dialogIds = openfunmenu o openfilemenu
 where
  openfilemenu = snd o openMenu undef filemenu
   where filemenu = Menu "File"
          ( MenuItem "Quit"  [MenuShortKey 'Q',
                              MenuFunction quit]) []
```

```
openfunmenu = snd o openMenu undef funmenu
 where
  funmenu = Menu "Functions"
   (ListLS
    [ MenuItem fname [(MenuFunction (noLS opentest))
       : (if (c<='9') [MenuShortKey c] []) ]
     \\  (_,_,fname) <- funs
          & opentest <- opentests
          & c <- ['1'..]
    ])[]
  opentests =
    [ functiondialog id fun
     \\ fun <- funs & id <- dialogIds ]
```

```
functiondialog ::
 Id (([String]->String),[String],String) (PSt .l)
   -> (PSt .l )
functiondialog dlgId (fun,initvals,name) ps
 # (argIds, ps) = accPIO (openIds arity) ps
   (resultId, ps) = accPIO openId ps
   (evalId, ps) = accPIO openId ps
 = snd (openDialog 0
       (dialog argIds resultId evalId) ps)
    where
      nrlines = 2
      width = PixelWidth 100
      arity = length initvals
```

```
eval id argIds resultId fun (ls,ps)
# (Just wstate,ps) = accPIO (getWindow id) ps
  input = [fromJust arg \\ (_,arg)
                <- getControlTexts argIds wstate]
= (ls, appPIO (setControlTexts
                [(resultId, fun input)]) ps)
dialog  argIds resultId evalId
 = Dialog name
   ( ListLS
     [ TextControl ("arg "+++toString n)
                [ControlPos (Left,zero)]
       :+: EditControl val width nrlines
                [ ControlId (argIds!!n)]
     \\ val <- initvals & n <- [0..]
     ]
     :+: TextControl "result" [ControlPos (Left,zero)]
```

```
    :+: EditControl "" width nrlines
                  [ ControlId resultId]
    :+: ButtonControl "Close"
                  [ ControlFunction (close dlgId) ]
    :+: ButtonControl "Quit"
                  [ ControlFunction quit ]
    :+: ButtonControl "Eval"
                  [ ControlId evalId,
                    ControlFunction
                   (eval dlgId argIds resultId fun) ]
)
[ WindowId dlgId, WindowOk evalId ]
```

```
close :: Id (.ls,PSt .l ) -> (.ls,PSt .l )
close id (ls,ps) = (ls, closeWindow id ps)
quit :: (.ls,PSt .l ) -> (.ls, PSt .l )
quit (ls,ps) = (ls, closeProcess ps)
no_arg :: y [String] -> String | toS y
no_arg f [] = toS f
no_arg f l = "This function should have no arguments instead of "
             +++toString (length l)
one_arg :: (x -> y) [String] -> String | fromS x & toS y
one_arg f [x] = toS (f (fromS x))
one_arg f l = "This function should have one arguments instead of
              +++toString (length l)
two_arg :: (x y -> z) [String] -> String
          | fromS x & fromS y & toS z
two_arg f [x,y] = toS (f (fromS x) (fromS y))
```

7. Increase effeciency

- strict arguments, if possible

- accumulating (instead of naive recursive calls):

```
lfib n = accfib n 1 1
       where accfib :: !Int !Int !Int -> !Int
             accfib 0 x y = x
             accfib n x y = accfib (dec n) y (x+y)
```

- left recursion instead of right recursion:

```
Length [a:x] = 1 + Length x
length l = acclen 0 1
acclen n [a:x] = acclen (inc n) x
```

- inversion, omitting intermediate data structures
- abstract data types encapsulate implementatation details, use arrays instead of lists
- avoid Curried functions at critical points
- use macros (less rewriting in run time)
- use special versions of funtions (inc)

8. Installation of Clean for Windows

1. unzip Clean133 and ObjectIO_1.2.1 by Winzip to D:Clean133, subdirectories Clean 1.3.3 and Object IO 121 are created (CleanIDE included).
2. move all subdirectories (StdLib 1.0, ObjectIO 1.2.1 , ObjectIO Examples 1.2.1) of Object IO 121 to Clean 1.3.3.
3. run change_registry.exe
4. start CleanIDE and set environment
   append StdLib 1.0, ObjectIO 1.2.1, ObjectIO 1.2.1 OS Windows
   under Environment - Edit Current - Path
5. open objectIO Examples hello
   new project , save , set main module, bring up to date and run

9. Installation of Clean for Linux

1. tar xvf Clean131.tar
2. cd clean
3. make
4. (in subdirectories stdenv and iolib/CleanSystemFiles touch *.abc, wait 60 seconds, touch *.o)
5. .bash_profile: export PATH=$PATH:..../clean/bin
6. clm hello
7. ./a.out