Today's Topic

Declarative programming...

- Functional style
- Logical style

If each of these two styles is appealing

- a combination of (features of) functional and logical programming

might be even more appealing.

Recent Article

- Sergio Antoy, Michael Hanus. *Functional Logic Programming*. Communications of the ACM, vol. 53, no. 4, pp. 74 - 85, 2010.

...highlights the benefits of combining the paradigm features of both logical and functional programming.

The next couple of slides shows quotes of this article...

Evolution of Programming Languages

...the stepwise introduction of abstractions hiding the underlying computer hardware and the details of program execution.

- *Assembly languages* ...introduce mnemonic instructions and symbolic labels for hiding machine codes and addresses
- *FORTRAN* ...introduces arrays and expressions in standard mathematical notation for hiding registers
- *ALGOL-like languages* ...introduce structured statements for hiding gotos and jump labels
- *Object-oriented languages* ...introduce visibility levels and encapsulation for hiding the representation of data and the management of memory

Evolution of Prog. Lang. (Cont'd)

- *Declarative languages*, most prominently *functional* and *logic languages* ...hide the order of evaluation by removing assignment and other control statements
  - A declarative program is a set of logical statements describing properties of the application domain
  - The execution of a declarative program is the computation of the value(s) of an expression wrt these properties

This way...

- The programming effort in a declarative language shifts from encoding the steps for computing a result to structuring the application data and the relationships between the application components
- Declarative languages are similar to formal specification languages *but* executable

Functional vs. Logic Languages

Functional languages

- are based on the notion of mathematical function
- programs are sets of functions that operate on data structures and are defined by equations using case distinction and recursion
- provide efficient, demand-driven evaluation strategies that support infinite structures

Logic languages

- are based on predicate logic
- programs are sets of predicates defined by restricted forms of logic foumulas, such as Horn clauses (implications)
- provide non-determinism and predicates with multiple input/output modes that offer code reuse

Functional Logic Languages: Examples

- Curry

  Michael Hanus (Ed.). *Curry: An Integrated Functional Logic Language* (*vers. 0.8.2, 2006*), `http://www.curry-language.org/`

- TOY

  F. López-Fraguas, J. Sánchez-Hernández. *TOY: A Multiparadigm Declarative System*. In Proceedings of RTA'99, Springer LNCS 1631, 1999, 244 - 247.

- Mercury

  Zoltan Somogyi, Fergus Henderson, Thomas Conway. *The Eexecution Algorithm of Mercury: An Efficient Purely Declarative Logic Programming Language*. Journal of Logic Programming 29 (1-3), 1996, 17 - 64.

  See also: The Mercury Project.
  `http://www.mercury.scce.unimelb.edu.au`

Functional Logic Languages: Examples

And there are many more...

- Escher
- Oz
- HAL
- ...

A Curry Appetizer

Regular Expressions

```
data RE a = Lit a
          | Alt (RE a) (RE a)
          | Conc (RE a) (RE a)
          | Star (RE a)
```

The Semantics of Regular Expressions

```
sem :: RE a -> [a]
sem (Lit c)    = [c]
sem (Alt r s)  = sem r ? sem s  // ? nondeterministic choice
sem (Conc r s) = sem r ++ sem s
sem (Star r)   = [] ? sem (Conc r (Star r))
```

A Curry Appetizer (Cont'd)

```
abstar = Conc (Lit 'a') (Star (Lit 'b'))

sem abstar => ["a","ab","abb"]

sem re =:= w  // =:= indicates that an equation is to be solved
              // rather than an operation to be defined
              // checks whether a given word w is in
              // the language of a given regular expression re

xs ++ sem re ++ ys =:= s  // checks whether a string s contains
    where xs, ys free     // a word generated by a regular
                          // expression re (similar to Unix's
                          // grep utility)
```

In this Lecture

...we will follow a different approach.

We will show how to...

- Integrate features of logical programming into functional programming
- *Central means*: Monads and monadic programming

Reference

This presentation is based on...

- Michael Spivey, Silvija Seres. *Combinators for Logic Programming*. In Jeremy Gibbons, Oege de Moor (Eds.), *The Fun of Programming*. Palgrave MacMillan, 2003.

Declarative Programming

- *Distinguishing* ...emphasizes the *"what"* rather than the *"how"*
  - *Essence* ...programs are declarative assertions about a problem, rather than imperative solution procedures
- *Variants* ...*functional* and *logical* programming
- Question ...can functional and logical programming be uniformly combined?

Combining Features of Functional and Logical Programming

Basic approaches...

- *Classical* ...designing new programming languages, which enjoy features of both programming styles (e.g. Curry)
- *Simpler* ...implementing an interpreter for one style using the other style
- *Still simpler* ...write "logical" programs in Haskell using a library of combinators

  $\leadsto$ this is the approach used in the following!

Further Reading

...on functional/logical programming languages:

- Michael Hanus, Herbert Kuchen, Juan Jose Moreno-Navarro. *Curry: A Truly Functional Logic Language*. In Proceedings of ILPS'95 Workshop on Visions for the Future of Logic Programming, 1995, 95-107.
- Zoltan Somogyi, Fergus Herderson, Thomas Conway. *Mercury: An Efficient Purely Declarative Logic Programming Language*. In Proceedings of the Australian Computer Science Conference, 1995, 499-512.

Remarks on the Combinator Approach used here

- Advantages and disadvantages in comparison to dedicated functional/logical programming languages
  - less costly
  - but less expressive

Central problems

- Modelling logical programs yielding
  - multiple answers
  - *logical* variables (no distinction between input and output variables)
- Modelling of the evaluation strategy inherent to logical programs

Running Example: Factoring of Natural Numbers

...decomposing a positive integer into the set of pairs of its factors

Example:

Integer	Factor-Pairs
24	(1,24), (2,12), (3,8), (4,6), ..., (24,1)

Obvious Solution:

```
factor   :: Int -> [(Int,Int)]
factor n =  [(r,s) | r <- [1..n], s <- [1..n], r*s == n]
```

In fact, we get:

```
?factor 24
[(1,24),(2,12),(3,8),(4,6),(6,4),(8,3),(12,2),(24,1)]
```

Observation

The previous solution exploits...

- Explicit domain knowledge
  - E.g. $r * s = n \;\Rightarrow\; r \leq n \wedge s \leq n$
  - This renders possible: Restriction to a finite search space [1..24] × [1..24]

Often such knowledge is not available. In general...

- The search space cannot be restricted a priori
- In the following thus: Considering the factoring problem as a search problem over an infinite search space [1..] × [1..]

Tackling the 1st Problem: Several Results

Solution ...lists of successes
⇝*lazy lists* (Phil Wadler)

Idea

- A function of type `a -> b` can be replaced by a function of type `a -> [b]`
- *Lazy evaluation* ensures that the elements of the result list (*list of successes*) are provided as they are found, rather than as a complete list after termination of the computation

Back to the Example

Realizing this idea in the factoring example (assuming that the search space cannot be bounded a priori):

```
factor   :: Int -> [(Int,Int)]
factor n =  [(r,s) | r <- [1..], s <- [1..], r*s == n]

?factor 24
[(1,24)

...followed by an infinite wait.
```

This is of questionable practical value.

Remedy: Fair Order via Diagonalization

Run through the search space of pairs in a fair order:

```
factor n = [(r,s) | (r,s) <- diagprod [1..][1..], r*s == n]
```

where

```
diagprod       :: [a] -> [b] -> [(a,b)]
diagprod xs ys = [(xs!!i, y!!(n-i)) | n <- [0..], i <- [0..n]]

...each pair (x,y) is reached after a finite number of steps
[(1,1),(1,2),(2,1),(1,3),(2,2),(3,1),(1,4),(2,3),(3,2),...]
```

Hence, in our example:

```
?factor 24
[(4,6),(6,4),(3,8),(8,3),(2,12),(12,2),(1,24),(24,1)

...and consequently all results; followed, however, by
   an infinite wait again.
```

Of course, this was expected, since the search space is infinite.

Systematic Remedy: Using Monads

Reminder:

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

Convention for the following development:

- `Stream a` ...for potentially infinite lists
- `[a]` ...for finite lists
- *Note*: The distinction between `Stream a` for infinite lists and `[a]` for finite lists is only conceptually. The following definition makes this explicit:

  ```
  type Stream a = [a]
  ```

List Monad

The monad of (potentially infinite) lists

```
-- return yields the singleton list
return :: a -> Stream a
return x = [x]

-- binding operator defined as follows
(>>=)    :: Stream a -> (a -> Stream b) -> Stream b
xs >>= f =  concat (map f xs)

-- other monad operations are irrelevant in our context
```

Benefit 1(2)

...`return` and `(>>=)` allow to model/to replace list comprehension:

For example, the meaning of the list comprehension

```
[(x,y) | x <- [1..], y <- [10..]]

   ...is equivalent to

concat (map (\x -> [(x,y) | y <- [10..]])[1..])

   ...is equivalent to

concat (map (\x -> concat (map (\y -> [(x,y)])[10..]))[1..])
```

Using `return` and `(>>=)` this can concisely be expressed by:

```
[1..] >>= (\x -> [10..] >>= (\y -> return (x,y)))
```

Benefit 2(2)

...and Haskell's `do`-notation allows an even more compact equivalent representation:

```
do x <- [1..]; y <- [10..]; return (x,y)
```

Recall:

General Rule:

```
do x1 <- e1; x2 <- e2; ... ; xn <- en; e
   ...is shorthand for
e1 >>= (\x1 -> e2 >>= (\x2 -> ... >>= (\xn -> e)...))
```

Fairness: Adapting the binding operator (>>=) 1(5)

Are we done? Not yet because...

Exploring the pairs of the search space is still not fair.

The expression

```
do x <- [1..]; y <- [10..]; return (x,y)
```

yields the stream

```
[(1,10),(1,11),(1,12),(1,13),(1,14),...
```

This problem is going to be tackled next...

Fairness: Adapting the binding operator (>>=) 2(5)

Idea ...embedding diagonalization in `(>>=)`

Implementation

Introducing a new type `Diag a`:

```
newtype Diag a = MkDiag (Stream a) deriving Show
```

...and an auxiliary function for stripping off the type constructor MkDiag

```
unDiag (MkDiag xs) = xs
```

Fairness: Adapting the binding operator (>>=) 3(5)

`Diag` is made an instance of the constructor class `Monad`:

```
instance Monad Diag where
  return x        = MkDiag [x]
  MkDiag xs >>= f = MkDiag (concat (diag (map (unDiag . f) xs)))
```

where

```
-- diag rearranges the values into a fair order
diag :: Stream (Stream a) -> Stream [a]
diag []       = []
diag (xs:xss) = lzw (++) [ [x] | x <- xs] ([] : diag xss)

-- lzw equals zipWith, however, the non-empty remainder
-- of the list is attached, if an argument list gets empty
lzw :: (a -> a -> a) -> Stream a -> Stream a -> Stream a
lzw f [] ys         = ys
lzw f xs []         = xs
lzw f (x:xs) (y:ys) = (f x y) : (lzw f xs ys)
```

Fairness 4(5)

Intuition:

- `return` yields the singleton list
- `undiag` strips off the constructor added by the function `f :: a -> Diag b`
- `diag` arranges the elements of the list into a fair order (and works equally well for finite and infinite lists)
- `lzw` reminds to "like zipWith"

Fairness 5(5)

The idea underlying `diag`:

...transforms an infinite list of infinite lists

```
[[x11,x12,x13,...],[x21,x22,...],[x31,x32,...],...]
```

...into an infinite list of finite diagonals

```
[[x11],[x12,x21],[x13,x22,x31],...]
```

Thereby:

```
?do x <- MkDiag [1..]; y <- MkDiag [10..]; return (x,y)
MkDiag[(1,10),(1,11),(2,10),(1,12),(2,11),(3,10),(1,13),...
```

Thus now achieved: The pairs are delivered in a fair order!

Back to the Factoring Problem 1(3)

Current state of our solution:

- Generating (pairs in a fair order): *done*
- Selecting (those pairs being part of the solution): *still open*

Approach for solving the selection problem, i.e., filtering out the pairs (r,s) *that satisfy* $r \times s = n$: ...filtering with conditions

For that purpose...

```
class Monad m => Bunch m where
  zero :: m a                 -- empty result, no answer
  alt  :: m a -> m a -> m a -- all answers either in xm or ym
  wrap :: m a -> m a          -- answers yielded by auxiliary
                              -- calculations; right now, wrap is
                              -- defined as the identity function
```

The value `zero` allows to express an empty answer set.

Back to the Factoring Problem 2(3)

In detail: The instance declaration for ordinary *lazy* lists

```
instance Bunch [] where
  zero      = []
  alt xs ys = xs ++ ys
  wrap xs   = xs
```

and for the monad `Diag`:

```
instance Bunch Diag where
  zero                        = MkDiag[]
  alt (MkDiag xs)(MkDiag ys) = MkDiag (shuffle xs ys)
  wrap xm                     = xm

shuffle [] ys     = ys
shuffle (x:xs) ys = x : shuffle ys xs
```

(Remark: `alt` and `wrap` will be used only later.)

Back to the Factoring Problem 3(3)

By means of `zero`, the function `test` yields the key for filtering...

```
test   :: Bunch m => Bool -> m()
test b =  if b then return() else zero
```

This doesn't look useful, but it provides the key to filtering:

```
?do x <- [1..]; () <- test (x 'mod' 3 == 0); return x
[3,6,9,12,15,18,21,24,27,30,33,...

?do x <- MkDiag [1..]; test (x 'mod' 3 == 0); return x
MkDiag[3,6,9,12,15,18,21,24,27,30,33,...
```

Are we done? 1(2)

Not yet!

Consider...

```
?do r <- MkDiag[1..]; s <- MkDiag[1..]; test(r*s==24); return (r,s)
MkDiag[(1,24)

...followed by an infinite wait.
```

What are the reasons for that?

```
do r <- MkDiag[1..]; s <- MkDiag[1..]; test(r*s==24); return (r,s)

   is equivalent to

do x <- MkDiag[1..]
   (do y <- MkDiag[1..]; test(x*y==24); return (x,y))
```

Are we done? 2(2)

I.e., the generator for y is merged with the subsequent test to the following (sub-) expression:

```
do y <- MkDiag[1..]; test(x*y==24); return (x,y)
```

Intuition:

- This expression yields for a given value of x all values of y with $x * y = 24$
- For $x = 1$ the answer $(1, 24)$ will be found, in order to search in vain for further values of y
- For $x = 5$ we thus do not observe any output

Solution Approach

The deeper reason for this undesired behaviour...
Missing associativity of (>>=) for Diag.

```
(xm >>= f) >>= g = xm >>= (\x -> f x >>= g)
  ...does not hold for (>>=) and Diag!
```

Remedy ...explicit grouping of generators to ensure fairness

```
?do (x,y) <- (do u <- MkDiag[1..]; v <- MkDiag[1..]; return (u,v))
    test (x*y==24); return (x,y)
MkDiag[(4,6),(6,4),(3,8),(8,3),(2,12),(12,2),(1,24),(24,1)

...all results, subsequently followed by an infinite wait
```

Remarks

- *All results, subsequently followed by an infinite wait*

  ...this is the best we can hope for if the search space is infinite.

- *Explicit grouping*

  ...required only because of missing associativity of >>=, otherwise both expressions would be equivalent.

- *In the following*

  ...avoid infinite waiting by indicating that a result has not (yet) been found.

Indicating that no solution is found...

To this purpose... a new type and breadth search

Intuition

- Type `Matrix` ...infinite list of finite lists
- Goal ...a program, which yields a matrix of answers, where row i contains all answers, which can be computed with costs $c(i)$.
- Solving the indication problem ...by returning the empty list in a row (means "nothing found")

Implementation... 1(3)

A new type

```
newtype Matrix a = MkMatrix (Stream [a]) deriving Show
```

with an auxiliary function for stripping off the constructor

```
unMatrix (MkMatrix xm) = xm
```

Implementation... 2(3)

Preliminary definitions to make `Matrix` an instance of class `Bunch`:

```
return x = MkMatrix[[x]]  -- Matrix with a single row
zero = MkMatrix[]         -- Matrix without rows
alt(MkMatrix xm) (MkMatrix ym) = MkMatrix(lzw (++) xm ym)
wrap(MkMatrix xm) = MkMatrix([]:xm) -- the clou is encoded in wrap!

(>>=) :: Matrix a -> (a -> Matrix b) -> Matrix b
(MkMatrix xm) >>= f = MkMatrix (bindm xm (unMatrix . f))

bindm :: Stream[a] -> (a -> Stream[b]) -> Stream[b]
bindm xm f = map concat (diag (map (concatAll . map f) xm))

concatAll :: [Stream [b]] -> Stream [b]
concatAll = foldr (lzw (++)) []
```

Implementation... 3(3)

In total we are now ready to make `Matrix` an instance of `Monad` and `Bunch`...

```
instance Monad Matrix where
  return x            = MkMatrix[[x]]
  (MkMatrix xm) >>= f = MkMatrix(bindm xm (unMatrix . f))

instance Bunch Matrix where
  zero                          = MkMatrix[]
  alt(MkMatrix xm)(MkMatrix ym) = MkMatrix(lzw (++) xm ym)
  wrap(MkMatrix xm)             = MkMatrix([]:xm)

intMat = MkMatrix[[n] | n <- [1..]]
```

Example

```
?do r <- intMat; s <- intMat; test(r*s==24); return (r,s)
MkMatrix[[],[],[],[],[],[],[],[],[(4,6),(6,4)],[(3,8),(8,3)],
  [],[],[(2,12),(12,2)],[],[],[],[],[],[],[],[],[],[],
  [(1,24),(24,1)],[],[],[],...
```

Independence of the Search Strategy 1(2)

Breadth search (`MkMatrix[[n]|n<-[1..]]`), depth search ([1..]), diagonalization...

Additional functions in order to be able to fix the strategy at the time of calling ("just in time")...

Control via a monad type...

```
choose        :: Bunch m => Stream a -> m a
choose (x:xs) =  wrap (return x 'alt' choose xs)

factor   :: Bunch m => Int -> m(Int, Int)
factor n =  do r <- choose[1..]; s <- choose[1..];
                          test(r*s==n); return (r,s)
```

Independence of the Search Strategy 2(2)

This allows...

- Usage of `factor` with different search strategies
- The specified type of `factor` determines the search monad (and hence the search strategy)

```
?factor 24 :: Stream(Int,Int)
[(1,24)

?factor 24 :: Matrix(Int, Int)
Matrix[[],[],[],[],[],[],[],[],[],[],[(4,6),(6,4)],
  [(3,8),(8,3)],[],[],[(2,12),(12,2)],[],[],[],[],[],[],
  [],[],[],[],[(1,24),(24,1)],[],[],[],...
```

Summary of Progress

Reminder...

Central problems

- Modelling logical programs with...
  - multiple results: **done** (essentially by means of lazy lists)
  - *logical* variables: **still open**
    - Common for logical programs: not a pure simplification of an initially completely given expression, but a simplification of an expression containing variables, for which appropriate values have to be determined. In the course of the computation, variables can be replaced by other subexpressions containing variables themselves, for which then appropriate values have to be found.
  - Modelling of the evaluation strategy inherent to logical programs: **done**
    - implicit search of logical programming languages has been made explicit
    - by means of type classes of Haskell even different search strategies were conveniently be realizable

Tackling the Final Problem: Terms, Substitutions & Predicates 1(5)

Towards the modelling in Haskell...

Terms will describe values of logical variables

```
data Term = Int Int | Nil | Cons Term Term | Var Variable
  deriving Eq
```

Named variables will be used for formulating queries, *generated variables* evolve in the course of the computation

```
data Variable = Named String | Generated Int
  deriving (Show, Eq)
```

Terms, Substitutions & Predicates 2(5)

Some auxiliary functions

- for transforming a string into a named variable

```
var   :: String -> Term
var s =  Var (Named s)
```

- for constructing a term representation of a list of integers

```
list    :: [Int] -> Term
list xs =  foldr Cons Nil (map Int xs)
```

Terms, Substitutions & Predicates 3(5)

Substitution and unification

```
-- Substitution: essentially a mapping from variables to terms
-- Details later
newtype Subst
```

Further support functions

```
apply :: Subst -> Term -> Term
idsubst :: Subst
unify :: (Term, Term) -> Subst -> Maybe Subst
```

Terms, Substitutions & Predicates 4(5)

Logical programs (in our Haskell environment) with `m` of type `bunch`:

```
-- Logical programs have type Pred m
type Pred m = Answer -> m Answer

-- Answers; the integer-component controls
-- the generation of new variables
newtype Answer = MkAnswer (Subst, Int)
```

Terms, Substitutions & Predicates 5(5)

```
-- "Initial answer"
initial :: Answer
initial =  MkAnswer (idsubst, 0)
run   :: Bunch m => Pred m -> m Answer
run p = p initial

-- "Program run of a predicate as query", where
-- p is applied to the initial answer
run p :: Stream Answer
```

Writing logical programs

Example...

`append(a,b,c)` where `a,b` lists and `c` concatenation of `a` and `b`

Implementation as a function of terms on predicates...

```
append :: Bunch m => (Term, Term, Term) -> Pred m

-- Implementation of append (later!) and of appropriate
-- Show-Functions is supposed
?run(append(list[1,2],list[3,4],var "z")) :: Stream Answer
[{z=[1,2,3,4]}]


-- note: more accurate and equivalent to the above list would be:
Cons 1 (Cons 2 (Cons 3 (Cons 4 Nil)))
```

Combinators for logical programs 1(4)

Simple predicates are formed by means of the operators `(=:=)` (equality of terms):

```
?run(var "x" =:= Int 3) :: Stream Answer
[{x=3}]
```

Implementation of `(=:=)` by means of `unify`:

```
(=:=) :: Bunch m => Term -> Term -> Pred m
(t=:=u)(MkAnswer(s,n)) =
   case unify(tu) s of
     Just s' -> return(MkAnswer(s',n))
     Nothing -> zero
```

Combinators for logical programs 2(4)

Conjunction of predicates by means of the operator `(&&&)` (conjunction):

```
?run(var "x" =:= Int 3 &&& var "y" =:= Int 4) :: Stream Answer
[{x=3,y=4}]

?run(var "x" =:= Int 3 &&& var "x" =:= Int 4) :: Stream Answer
[]
```

Implementation by means of the operator `(>>=)` of type `bunch`:

```
(&&&) :: Bunch m => Pred m -> Pred m -> Pred m
(p &&& q) s = p s >>= q

-- equivalent and emphasizing the sequentiality would be
do t <- p s; u <- q t; return u
```

Combinators for logical programs 3(4)

Disjunction of predicates by means of the operator `(|||)` (Disjunction):

```
?run(var "x" =:= Int 3 ||| var "x" =:= Int 4) :: Stream Answer
[{x=3,x=4}]
```

Implementation by means of the operator `alt` of type `bunch`:

```
(|||) :: Bunch m => Pred m -> Pred m -> Pred m
(p ||| q) s = alt (p s) (q s)
```

Combinators for logical programs 4(4)

Introducing new variables in predicates (exploiting the integer-component of answers)

...on the construction of local variables in recursive predicates

```
exists :: Bunch m => (Term -> Pred m) -> Pred m
exists p (MkAnswer (s,n)) =
  p (Var(Generated n)) (MkAnswer(s,n+1))
```

Also for handling recursive predicates

...ensures that in connection with `Matrix` the costs per recursion unfolding increase by 1

```
step     :: Bunch m => Pred m -> Pred m
step p s =  wrap (p s)
```

Example

Examples of applications of `wrap` and `step`

```
?run (var "x" =:= Int 0) :: Matrix Answer
MkMatrix[[{x=0}]]

?run(step(var "x" =:= Int 0)) :: Matrix Answer
MkMatrix[[],[{x=0}]]
```

Recursive Programs 1(2)

This allows us to provide the implementation of `append`:

```
append(p,q,r) =
  step(p =:= Nil &&& q =:= r
       ||| exists (\x -> exists (\a -> exists (\b ->
             p =:= Cons x a &&& r =:= Cons x b
             &&& append(a,q,b)))))
```

Recursive Programs 2(2)

Also the following application is possible (which is common for logical programs):

The concatenation of which lists equals the list `[1,2,3]`?

```
?run(append(var "x", var "y", list[1,2,3])) :: Stream Answer
[{x = Nil, y = [1,2,3]},
  {x = [1], y = [2,3]},
  {x = [1,2], y = [3]},
  {x = [1,2,3], y = Nil}]
```

A More Complex Example 1(2)

Constructing "good" sequences consisting of zeros and ones.

Convention

1. The sequence [0] is good
2. If the sequences s1 and s2 are good, then also the sequence [1] ++ s1 ++ s2
3. Except of the sequences according to 1. and 2., there are no other good sequences

A More Complex Example 2(2)

Implementation as predicate

```
good(s) =
  step (s =:= Cons(Int 0) Nil
        ||| exist (\t -> exists (\q -> exists (\r ->
               s =:= Cons (Int 1) t &&& append(q,r,t)
               &&& good(q) &&& good(r)))))
```

Examples 1(4)

Test of being "good":

```
?run (good (list[1,0,1,1,0,0,1,0,0])) :: Stream Answer
[{}]  -- empty answer set, if list is good

?run (good (list[1,0,1,1,0,0,1,0,1])) :: Stream Answer
[]    -- no answer, if list is not good
```

Note: The "empty answer" and "no answer" correspond to "yes" and "no" of a Prolog system.

Examples 2(4)

Constructing good lists

```
-- Unfair bunch-type: some answers are missing
?run(good(var "s")) :: Stream Answer
[{s=[0]},
  {s=[1,0,0]},
  {s=[1,0,1,0,0]},
  {s=[1,0,1,0,1,0,0]},
  {s=[1,0,1,0,1,0,1,0,0]},...
```

Examples 3(4)

```
-- For comparison: fair bunch-type
?run(good(var "s")) :: Diag Answer
Diag[{s=[0]},
  {s=[1,0,0]},
  {s=[1,0,1,0,0]},
  {s=[1,0,1,0,1,0,0]},
  {s=[1,1,0,0,0]},
  {s=[1,0,1,0,1,0,1,0,0]},
  {s=[1,1,0,0,1,0,0]},
  {s=[1,0,1,1,0,0,0]},
  {s=[1,1,0,0,1,0,1,0,0]},...
```

Examples 4(4)

```
-- For comparison: breadth-first search bunch-type
-- The output of results is more "predictable"
?run(good(var "s")) :: Matrix Answer
MkMatrix[[],
   [{s=[0]}],[],[],[],
   [{s=[1,0,0]}],[],[],[],
   [{s=[1,0,1,0,0]}],[],
   [{s=[1,1,0,0,0]}],[],
   [{s=[1,0,1,0,1,0,0]}],[],
   [{s=[1,0,1,1,0,0,0]},{s=[1,1,0,0,1,0,0]}],[],
   ...
```

Finally: Definitions still to be delivered 1(4)

New infix operators

```
infixr 4 =:=
infixr 3 &&&
infixr 2 |||
```

Substition

```
newtype Subst = MkSubst [(Var, Term)]
unSubst(MkSubst s) = s

idsubst = MkSubst[]
extend x t (MkSubst s) = MkSubst ((x,t):s)
```

Definitions to be delivered 2(4)

Application of substitution

```
apply :: Subst -> Term -> Term
apply s t =
       case deref s t of
         Cons x xs -> Cons (apply s x) (apply s xs)
         t’        -> t’

deref :: Subst -> Term -> Term
deref s (Var v) =
       case lookup v (unSubst s) of
         Just t    -> deref s t
         Nothing   -> Var v
deref s t = t
```

Definitions to be delivered 3(4)

Unification

```
unify :: (Term, Term) -> Subst -> Maybe Subst
unify (t,u) s =
  case (deref s t, deref s u) of
    (Nil, Nil) -> Just s
    (Cons x xs, Cons y ys)  -> unify (x,y) s >>= unify (xs, ys)
    (Int n, Int m) | (n==m) -> Just s
    (Var x, Var y) | (x==y) -> Just s
    (Var x, t)              -> if occurs x t s then Nothing
                                     else Just (extend x t s)
    (t, Var x)              -> if occurs x t s then Nothing
                                     else Just (extend x t s)
    (_,_)                   -> Nothing
```

Definitions to be delivered 4(4)

```
occurs :: Variable -> Term -> Subst -> Bool
occurs x t s =
  case deref s t of
    Var y     -> x == y
    Cons y ys -> occurs x y s || occurs x ys s
    _         -> False
```

Conclusion

Current functional logic languages aim at balancing

- generality (in terms of paradigm integration)
- efficient implementations

Functional logic programming offers

- support of specification, prototyping, and application programming within a single language
- terse, yet clear, support for rapid development by avoiding some tedious tasks, and allowance of incremental refinements to improve efficiency

Overall: Functional logic programming

- an emerging paradigm with appealing features

Next Course Meeting...

- Thu, April 29, 2010, lecture time: 4.15 p.m. to 5.45 p.m., lecture room on the ground floor of the building Argentinierstr. 8