Representing cyclic structures as nested datatypes

Tarmo Uustalu

joint work with Neil Ghani, Makoto Hamana, Varmo Vene

TSEM, 16 March 2006

Cyclic structures?

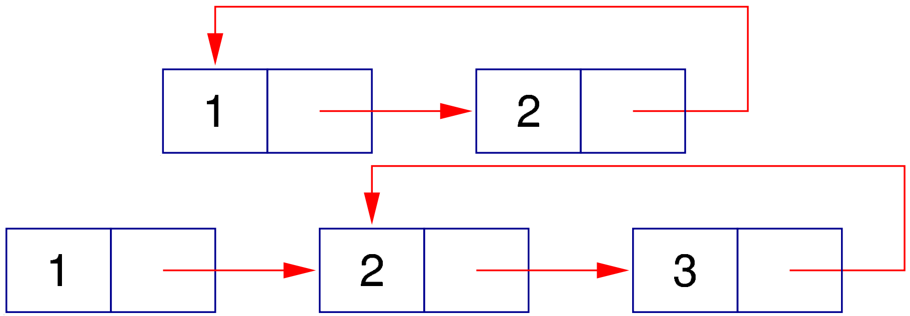
- Every now and then you'd like to represent cyclic structures in Haskell in such a way that the cycles can be manipulated explicitly (no implicit unwinding).
- This is tricky. Cf., e.g., Fegaras, Sheard or Turbak, Wells.
- An exercise about pointers in FP, but really a bit more (you do not want to think in terms of pointers too much, instead you want a representation that thinks for you).
- Here: We proceed from the solution of Fegaras and Sheard (explicit fixpoint operators) and improve on it, switching to a more accurate and better manipulable representation.

Cyclic lists

- By a cycle we mean a substructure in an infinite structure that repeats itself on a path down from the root.
- Examples:

```
clist1 = fix (\ xs -> 1 : 2 : xs)
clist2 = 1 : fix (\ xs -> 2 : 3 : xs)

fix :: (a -> a) -> a
fix f = f (fix f)
```



Cyclic lists as a mixed-variant datatype

- Fegaras and Sheard proposed making fixpoint operations explicit (cf explicit substitutions). An explicit fixpoint operator is a constructor, not a function, so it does not do anything by itself.
- Cyclic lists a la Fegaras and Sheard:

```
data CList = Nil
           | Cons Int CList
           | Rec (CList -> CList)    -- notice the two recursive occurrences
                                     -- of CList, one is negative!
```

- Examples:

```
clist1 = Rec (\ xs -> Cons 1 (Cons 2 xs))
clist2 = Cons 1 (Rec (\ xs -> Cons 2 (Cons 3 xs)))
```

- Functions manipulating these representations must unfold `Rec`-structures (there is not much else they could do).

  Tail function:

```
ctail :: CList -> CList
ctail (Cons x xs) = xs
ctail (Rec f)     = ctail (f (Rec f))
```

  Map function:

```
cmap :: (Int -> Int) -> CList -> CList
cmap g Nil         = Nil
cmap g (Cons x xs) = Cons (g x) (cmap g xs)
cmap g (Rec f)     = cmap g (f (Rec f))
```

- Further shortcomings of this representation:
- The semantic category has to be algebraically compact for mixed-variant types to make semantic sense.
- The argument type `CList` → `CList` of `Rec` is too big: we only want fixpoints of append-functions, not of just any list-functions. The following is not cyclic:

```
acyclic = Rec (\ xs -> Cons 1 (cmap (+1) xs))
```

- One can represent the unproductive empty cycle, which cannot be unwinded:

```
empty = Rec (\ xs -> xs)
```

- The representation is not unique: We can mark a position with zero, one or multiple bound variables:

```
clist1 = Rec (\ xs -> Rec (\ ys ->
         Cons 1 (Cons 2 (Rec \ zs -> xs))))
```

- A fix to two last problems: require that `Rec` always comes in combination with `Cons` and that `Cons` can never come alone:

```
data CList = Nil
           | RCons Int (CList -> CList)
```

- But overall, the approach is comparable the “higher-order abstract syntax” (HOAS) representation of lambda calculus syntax and the problems remain.
- A better alternative: make the Haskell-level lambda-abstractions object-level.

Something you did not know: Nested datatypes

- The parameterized datatype of lists is homogeneous or non-nested:

```
data List a = Nil | Cons a (List a)
```

- But one can also define parameterized datatypes that are heterogeneous or nested (terminology of Bird and Meertens): in the following definitions, the parameter varies in the recursion:

```
data Nest a = NilN | ConsN (a, Nest (a, a))
data Bush a = NilB | ConsB (a, Bush (Bush a))
```

- While homogeneous datatypes are just families of recursive types, heterogeneous datatypes are recursive families of types. You cannot define `Nest Int` in isolation from `Nest (Int, Int)`.

Cyclic lists as a nested datatype

- Idea: use de Bruijn levels, number the positions on the path from the head to the position immediately preceding the given one, refer to these numbers.

  (de Bruijn indices: number the positions in the opposite order starting from the position immediately before the given one.)

- Datatype:

```
data Void                    -- empty type
void :: Void -> a            -- empty function

data CList a = Var a         -- pointer
             | Nil
             | RCons Int (CList (Maybe a))
                             -- the tail can point to one position more
```

- Positions:

```
Nothing :: Maybe Void
Nothing, Just Nothing :: Maybe (Maybe Void)
Nothing, Just Nothing, Just (Just Nothing) :: Maybe (Maybe (Maybe Void)
...
```

- Examples:

```
clist1 = RCons 1 (RCons 2 (Var Nothing))
clist2 = RCons 1 (RCons 2 (RCons 3 (Var (Just Nothing))))
```

- Importantly, we can only define fixpoints of append-functions. And as always with de Bruijn notations, we need not worry about α-conversion.

- List algebra structure:

```
cnil :: CList Void
cnil = Nil

ccons :: Int -> CList Void -> CList Void
ccons x xs = RCons x (shift xs)

shift :: CList a -> CList (Maybe a)
shift (Var z)      = Var (Just z)
shift Nil          = Nil
shift (RCons x xs) = RCons x (shift xs)
```

(`shift` renumbers the positions)

- List coalgebra structure: the head function (undefined on the empty list):

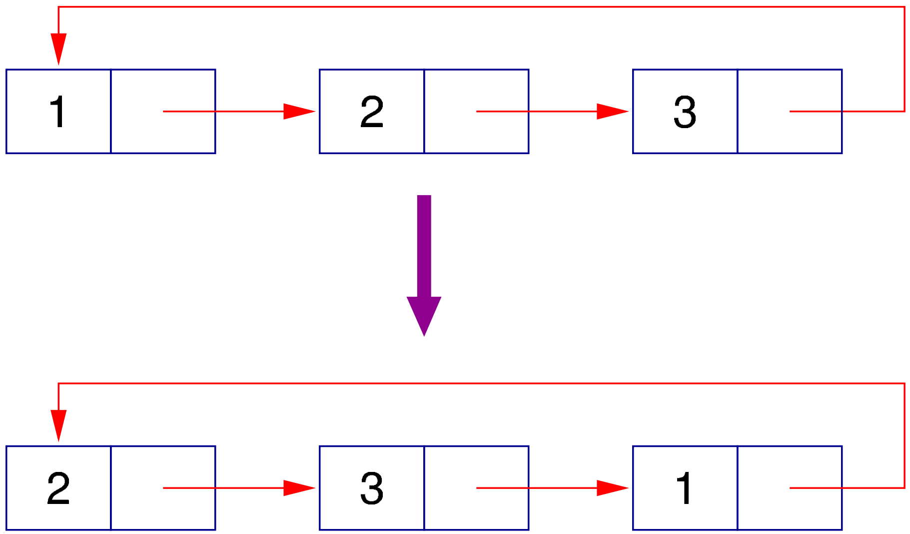
```
chead :: CList Void -> Int
chead (Var z)     = void z
chead (RCons x _) = x
```

- List coalgebra structure: the tail function (undefined on the empty list):

```
ctail :: CList Void -> CList Void
ctail (Var z)      = void z
ctail (RCons x xs) = csnoc x xs

csnoc :: Int -> CList (Maybe a) -> CList a
csnoc y (Var Nothing)  = RCons y (Var Nothing)
csnoc y (Var (Just z)) = Var z
csnoc y Nil            = Nil
csnoc y (RCons x xs)   = RCons x (csnoc y xs)
```

(`csnoc` renumbers the positions a list but also appends a value to it)


1
2
3
2
3
1

- Example of using the coalgebra structure: We can unwind a cyclic list into a possible infinite list:

```
unwind :: CList Void -> [Int]
unwind Nil = []
unwind xs  = chead xs : unwind (ctail xs)
```

- This is actually an *unfold* for possibly infinite lists:

```
unwind = unfoldr cheadtail

unfoldr :: (c -> Maybe (a, c)) -> c -> [a]
unfoldr f c = case f c of
                Nothing      -> []
                Just (a, c') -> a : unfoldr f c'

cheadtail :: CList Void -> Maybe (Int, CList Void)
cheadtail Nil = Nothing
cheadtail xs  = Just (chead xs, ctail xs)
```

- Unfolding list algebras into possibly infinite cyclic lists (detecting cycles) (assumes terminating equality on the state space):

  Idea: keep a list of the states already visited (together with an aligned list of the positions where this happened):

```
cunfoldL  :: Eq c => (c -> Maybe (Int, c)) -> c -> CList Void
cunfoldL = cunfoldL' [] []

cunfoldL' :: Eq c => [c] -> [a]
                   -> (c -> Maybe (Int, c)) -> c -> CList a
cunfoldL' cs as ht c = case lookup c (zip cs as) of
   Nothing -> case ht c of
      Nothing      -> Nil
      Just (x, c') -> let cs' = cs ++ [c]
                          as' = Nothing : map Just as
                      in RCons x (cunfoldL' cs' as' ht c')
   Just a  -> Var a
```

- Example application: zipWith for cyclic lists:

```
czipWith :: (Int -> Int -> Int)
                   -> CList Void -> CList Void -> CList Void
czipWith f xs ys = cunfoldL ht (xs, ys) where
        ht (xs, ys) = case cheadtail xs of
               Nothing       -> Nothing
               Just (x, xs') -> case cheadtail ys of
                  Nothing       -> Nothing
                  Just (y, ys') -> Just (f x y, (xs', ys'))
```

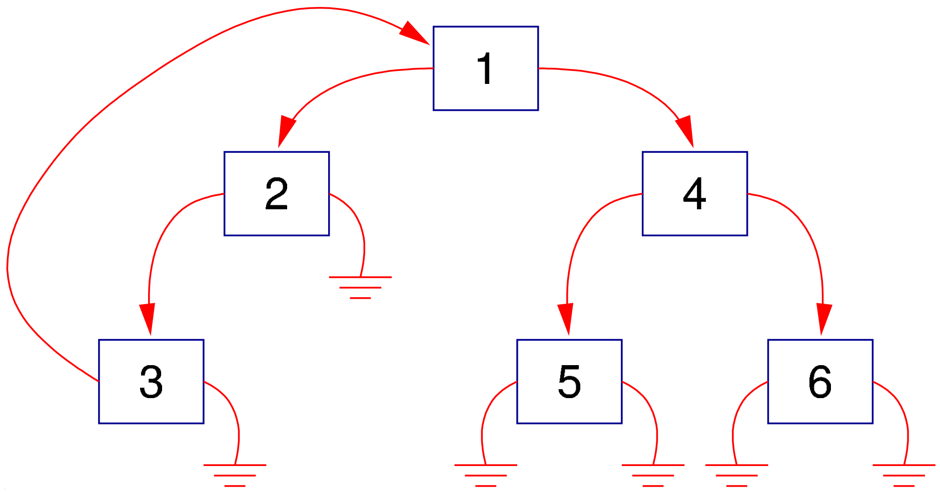
Cyclic binary trees

- What we just showed for lists, scales up to other datatypes.
- Consider binary trees. Because of non-linearity (multiple paths down from the root), they are more general.
- Datatype of cyclic binary trees:

```
data CTree a = VarT a
             | Leaf
             | RBin Int (CTree (Maybe a)) (CTree (Maybe a))
```

- Example:

```
ctree = RBin 1 (RBin 2 (RBin 3 (VarT Nothing) Leaf)
                       Leaf)
               (RBin 4 (RBin 5 Leaf Leaf)
                       (RBin 6 Leaf Leaf))
```


1
2
4
3
5
6

- Tree algebra structure:

```
cleaf :: CTree Void
cleaf = Leaf

cbin :: Int -> CTree Void -> CTree Void -> CTree Void
cbin x xsL xsR = RBin x (shiftT xsL) (shiftT xsR)

shiftT :: CTree a -> CTree (Maybe a)
shiftT (VarT x)         = VarT (Just x)
shiftT Leaf             = Leaf
shiftT (RBin x xsL xsR) = RBin x (shiftT xsL) (shiftT xsR)
```

(`shiftT` renumbers the positions)

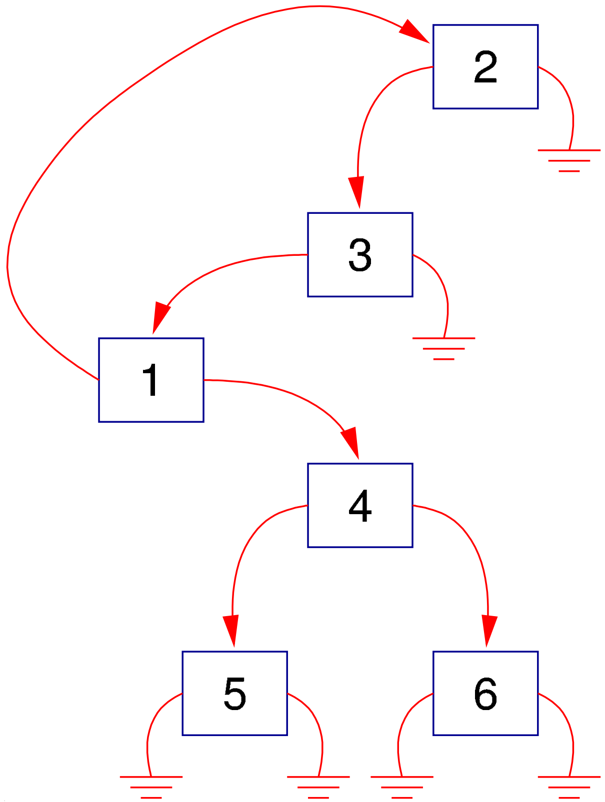
- Tree coalgebra structure:

  Here, the situation is more subtle than with lists: trees are nonlinear, the left subtree of a cyclic tree with back-pointed root node contains not only a relocated copy of this root node but also the right subtree.

```
csubL :: CTree Void -> CTree Void
csubL (VarT z)         = void z
csubL (RBin x xsL xsR) = csnocL x xsR xsL

csnocL :: Int -> CTree (Maybe a) -> CTree (Maybe a) -> CTree a
csnocL y ys (VarT Nothing)   = RBin y (VarT Nothing) ys
csnocL y ys (VarT (Just z))  = VarT z
csnocL y ys Leaf             = Leaf
csnocL y ys (RBin x xsL xsR) = RBin y (csnocL y ys' xsL)
                                      (csnocL y ys' xsR)
                             where ys' = shiftT ys

csubR :: CTree Void -> CTree Void
...
csnocR :: Int -> CTree (Maybe a) -> CTree (Maybe a) -> CTree a
...
```


2
3
1
4
5
6

Conclusions

- Fegaras and Sheard's basic idea to represent cycles as explicit fixpoints was correct, but it is considerably better to use de Bruijn notation instead of HOAS.
- The technique extends to all polynomial datatypes.
- Extend this to sharing: in addition to back-edges, allow edges to positions to the left from the spine.
- Develop a categorical account of rational and cyclic coinductive types.