Modelling Cyclic Structures by Nested Datatypes

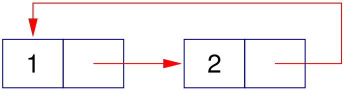
Varmo Vene

Teooriapäevad, Koke, 3-5 February 2006

Motivation

- Algebraic datatypes provide a nice way to represent tree-like structures.
- Lazy languages, eg. Haskell, allow to build also cyclic structures.

```
cycle = 1 : 2 : cycle
```



- Allows to represent complete infinite structures in finite memory

Motivation

- However, there is no support for manipulating cyclic structures
- Eg. mapping over cyclic list gives an infinite list
  `map (+1) cycle  ==>  [2,3,2,3,2,3,2,3,....`
- In fact, there is no way to distinguish cyclic structures from infinite ones
- Our aim is to represent cyclic structures inductively, hence to separate them from infinite (coinductive) structures.
- This gives the ability to explicitly manipulate cyclic structures either directly or using generic operations like `fold`, etc.

Cyclic Lists – 1st attempt

- Sheard, Fegaras 1996 (??)
- Definition:

```
data CList = Nil
           | Cons Int CList
           | Rec (CList -> CList)
```

  with an extra axiom

```
Rec f = f (Rec f)
```

- Examples:

```
clist1 = Rec (λ xs -> Cons 1 xs)
clist2 = Rec (λ xs -> Cons 1 (Cons 2 xs))
clist3 = Cons 1 (Rec (λ xs -> Cons 2 xs))
```

- Doesn't have unique representation
- Requires higher-order recursive datatypes

Cyclic Lists – 2nd attempt

- Definition:

```
data CList = Nil
           | Cons Int CList
           | Var Int
```

- Examples:

```
clist1 = Cons 1 (Var 1)
clist2 = Cons 1 (Cons 2 (Var 1))
clist3 = Cons 1 (Cons 2 (Var 2))
```

- Can have pointers outside of the list.
- This can be avoided by using dependent types:

```
data CList (n :: Int) = Nil
                      | Cons Int (CList (n+1))
                      | Var 1 ... Var n
```

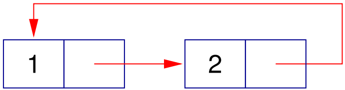
Cyclic Lists as Nested Datatype

- Definition:

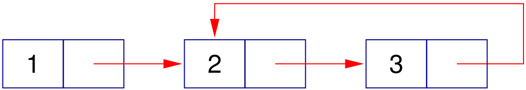
```
data CList a = Nil
             | Cons Int (CList (Maybe a))
             | Var a
```

- `Var a` represents a backward pointer to an element in a list.
- `Nothing` is the pointer to the first element of a cyclic list.
- `Just Nothing` is for the second element, and
- `Just (Just Nothing)` is for the third element, etc.
- The complete cyclic list has type `CList Empty`, where `Empty` is a type without constructors.

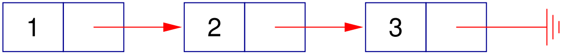
Examples



`Cons 1 (Cons 2 (Var Nothing))`



`Cons 1 (Cons 2 (Cons 3 (Var (Just Nothing))))`



`Cons 1 (Cons 2 (Cons 3 Nil))`

"Folding" a cycle

- "Standard" fold

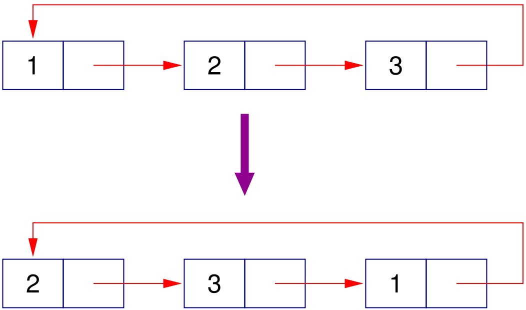
```
fold :: (forall a. a -> f a)
     -> (forall a . f a)
     -> (forall a. Int -> f (Maybe a) -> f a)
     -> CList a -> f a
fold v n c (Var x)      = v x
fold v n c Nil          = n
fold v n c (Cons x l) = c x (fold v n c l)
```

- Example:

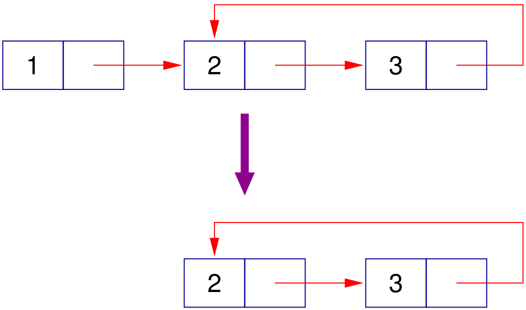
```
newtype K a = K Int
summ = fold (λ x -> K 0) (K 0) (λ i (K j) -> K (i+j))
```

- In genereal, not very easy to use.
- Generalized folds for nested data types (Bird, Patterson)

Tail of a Cyclic List



Tail of a Cyclic List



Coalgebraic structure on Cyclic Lists

```
chead :: CList Empty -> Int
chead (Cons x l) = x

ctail :: CList Empty -> CList Empty
ctail (Cons x l) = csnoc x l

csnoc :: Int -> CList (Maybe a) -> CList a
csnoc h Nil                = Nil
csnoc h (Var Nothing)  = Cons h (Var Nothing)
csnoc h (Var (Just x)) = Var x
csnoc h (Cons x l)       = Cons x (csnoc h l)

unwind :: CList Empty -> [Int]
unwind Nil = []
unwind l   = chead l : unwind (ctail l)
```

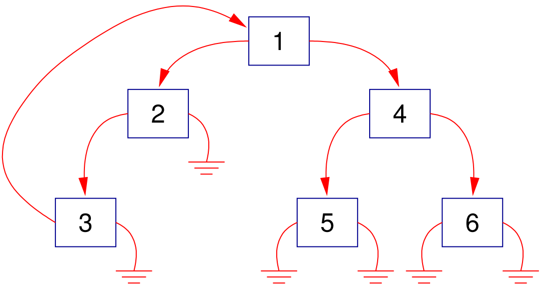
Cyclic Binary Trees

- Definition:

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

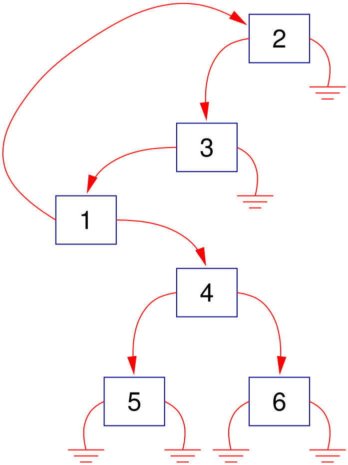
- Nodes are "numbered" top-down.
- All nodes on the same level have the same "number".
- Has only backpointers to form cycles.
- Pointers to other directions forbidden, hence no sharing.

Example




```
Node 1 (Node 2 (Node 3 (VarT Nothing) Leaf)
                Leaf)
       (Node 4 (Node 5 Leaf Leaf)
               (Node 6 Leaf Leaf))
```

Cyclic children



Coalgebra structure on Cyclic Binary Trees

```
sonL :: CTree a -> CTree a
sonL (Node x t1 t2) = tsnocL x t2 t1

tsnocL :: Int -> CTree (Maybe a)
              -> CTree (Maybe a) -> CTree a
tsnocL x t (VarT Nothing)  = Node x (VarT Nothing) t
tsnocL x t (VarT (Just n)) = VarT n
tsnocL x t Leaf            = Leaf
tsnocL x t (Node y t1 t2)  = Node y (tsnocL x t' t1)
                                    (tsnocL x t' t2)
    where t' = relabel t

relabel :: CTree a -> CTree (Maybe a)
relabel (VarT x) = VarT (Just x)
relabel Leaf = Leaf
relabel (Node x t1 t2) = Node x (relabel t1) (relabel t2)
```

Conclusions

- Generic framework to model cyclic structures.
- Backward pointers — no sharing, just cycles.
- Type system guarantees the safety of pointers.
- To do: Work out all the details ...
- New examples: zip, reverse, etc.