Representing Cyclic Structures as Nested Datatypes

Neil Ghani[1], Makoto Hamana[2], Tarmo Uustalu[3], and Varmo Vene[4]

[1] School of Computer Science and IT, Univ. of Nottingham,
Jubilee Campus, Wollaton Road, Nottingham NG8 1BB, UK; nxg@cs.nott.ac.uk
[2] Dept. of Computer Science, Univ. of Gunma,
Tenjincho 1-5-1, Kiryu, Gunma 376-8515, Japan; hamana@cs.gunma-u.ac.jp
[3] Inst. of Cybernetics at Tallinn Univ. of Technology,
Akadeemia tee 21, EE-12618 Tallinn, Estonia; tarmo@cs.ioc.ee
[4] Dept. of Computer Science, Univ. of Tartu,
J. Liivi 2, EE-50409 Tartu, Estonia; varmo@cs.ut.ee

Abstract

We show that cyclic structures, i.e., finite or possibly infinite structures with back-pointers, unwindable into possibly infinite structures, can be elegantly represented as nested datatypes. This representation is free of the various deficiencies characterizing the more naive representation as mixed-variant datatypes. It is inspired by the representation of lambda-terms as a nested datatype via the de Bruijn notation.

1 INTRODUCTION

Inductive and coinductive types (or simply datatypes in languages where they coincide) and the associated programming constructs, in particular, structured recursion and corecursion schemes such as fold, unfold, primitive recursion etc., are central to functional programming languages. Luckily, they are also well-understood. This derives from their elegant semantics as initial functor-algebras and final functor-coalgebras.

But what about datastructures with cycles or sharing? Such structures arise naturally, e.g., in functional language implementations (term-graph rewriting), but how should we represent and manipulate them on the level of a programming language?

In the present paper, we develop a novel approach to representing cyclic datastructures. In this approach, they become nested datatypes (inductive or coinductive, depending on whether we want to represent finite cyclic structures or possibly infinite cyclic structures). The idea is to see pointers as variables bound at the positions they point to and to represent the resulting syntax with variable binding in the style of de Bruijn as a nested datatype (inductive or coinductive). As a consequence of this careful design, the resulting account is imbued with the same elegance that characterizes our understanding of inductive and coinductive types. We demonstrate the fundamental programming constructs for cyclic structures that the representation delivers and describe how these arise from the combinators we have available for nested datatypes.

Related work To our knowledge, there are not many works in the functional programming literature that study explicit representation and manipulation of cyclic datastructures in a systematic fashion. The central for us idea of pointers as variables bound at the pointed position was pioneered by Fegaras and Sheard [8]. But they chose to represent variable binding via higher-type dataconstructors as in higher-order abstract syntax, relying on mixed-variant datatypes. This representation suffers from multiple drawbacks, the most serious being that almost all useful functions manipulating cyclic lists must unwind them anyway, despite the cycles being explicit. Turbak and Wells [14] have defined a rather low-level representation, where every node in a structure gets a unique integer identifier, and developed some combinators for working with that representation. That de Bruijn notations of lambda-terms can be represented as a nested datatype was observed by Bird and Paterson [7] and Altenkirch and Reus [3]; the mathematical account in terms of presheaves is due to Fiore, Plotkin and Turi [9]. Structured recursion/corecursion for nested datatypes has been investigated by Bird, Meertens and Paterson [5, 6], Martin, Gibbons and Bayley [4], and Abel, Matthes and Uustalu [2].

Organization In Sec. 2, we discuss cyclic lists. We begin by outlining Fegaras and Sheard's representation and highlighting its shortcomings. Then we proceed to our own proposal, deriving it by small modifications from theirs. We describe the list algebra and coalgebra structures on cyclic lists as well as unfolding of list coalgebras into cyclic lists. In Sec. 3, we demonstrate that our approach scales to non-linear datatypes, treating the case of cyclic binary trees. In Sec. 4, we treat general cyclic structure datatypes specified by a base functor.

2 CYCLIC LISTS

Cyclic structures are a means to make explicit the cycles that may exist in possibly infinite structures of a coinductive type. By a cycle, we mean that a substructure repeats itself on a path down from the root.

We shall first limit our attention to cyclic lists. Lists are linear datastructures (only one path down from the root), thus an infinite list can contain at most one cycle, in which case it can be compacted into a finite cyclic list. For example, the list [1,2,3,4,5..] is cycle-free while the lists [1,2,1,2,1,2..] and [1,2,3,2,3,2,3..] both contain a cycle. They are described as cyclic lists in Fig. 1. We can see that, intuitively, a cyclic list is a list that can either finish normally or with a pointer back to a position somewhere before the final position.

In Haskell, the two cyclic lists are trivially defined using the fixpoint combinator `fix`:

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

fix :: (x -> x) -> x
```

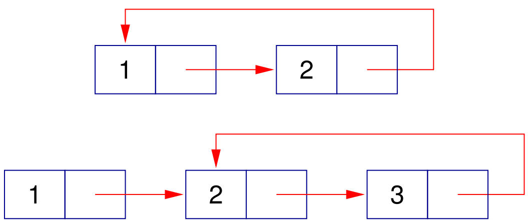

Figure 1. Two cyclic lists

```
fix f = x where x = f x
```

But obviously this representation gives no means for explicit manipulation of the cycles in the two lists.

2.1 Cyclic lists as a mixed-variant datatype

In [8], Fegaras and Sheard proposed a representation of cyclic structures as mixed-variant datatypes[1]. The idea is to represent cycles by explicit fixpoint expressions, i.e. in addition to the normal constructors, a cyclic datatype `C` has an extra constructor `Rec` : $(\mathtt{C} \rightarrow \mathtt{C}) \rightarrow \mathtt{C}$. For cyclic lists, this gives the following definition, which ought to be read inductively or coinductively depending on whether one wants to represent finite or possibly infinite cyclic lists (but see the remark on algebraic compactness below).

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

For instance, we can represent the cyclic lists from Fig. 1 as follows:

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

But, unfortunately, although cycles are explicit in this representation, almost all functions manipulating cyclic structures will need to unwind the cycles (based on the intuition that `Rec` means `fix`). The reason is that, although, we can recognize cycles as structures of the form `Rec f`, the argument `f` is a function and we cannot analyze functions well.

For instance, the tail function can be defined as

[1] In their terminology, mixed-variant datatypes are higher-order datatypes, as mixed-variance is achieved through *higher-type* dataconstructors. We prefer the term 'mixed-variant datatypes' to avoid confusion with nested datatypes that are fixpoints of *higher-kind* typeconstructors.

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

And similarly, the map function can be defined as

```
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))
```

(These function definitions can be justified in terms of structured recursion for mixed-variant datatypes, see e.g. [13, 8], but we will not so here.)

In addition to this drawback, the approach has further shortcomings. First, the use of mixed-variant datatypes means that the semantic category has to be algebraically compact [10], i.e., a category where inductive and coinductive types coincide. Categories like **Set** are ruled out immediately and one needs something like **CPO**. This makes the reasoning more complex. Second, and more important, the argument type `CList` $\rightarrow$ `CList` of `Rec` is far too big: we only want fixpoints of append-functions, not of just any list-functions. For example, the representation

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

does not correspond to a cyclic list. Third, it is hard to define cyclic list manipulating functions so that they do not unroll their input completely. This illustrated already by the examples of `ctail` and `cmap` above.

Yet another deficiency is that one can easily represent the empty cycle, which cannot be unrolled, because it is unproductive:

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

Further, the representation is not unique. For instance, the cyclic list `clist1` could be equivalently represented as

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

Essentially, the constructor `Rec` labels a position in a list with a variable which can be used in the rest of the list as a back-pointer to form a cycle. A list can contain at most one cycle, so there can be at most one position that is pointed to (this is because lists are linear, in branching structures there can be more cycles). But nothing prevents more than one label for this position and labels for other positions.

The problem of the possible empty cycle can be solved easily. To disable the unproductive empty cycle, it suffices to require that `Rec` always comes in combination with `Cons`. Hence, we introduce a new constructor `RCons` which combines both:

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

Apart from the empty cycle, this modification disables multiple labels for one position. But we can still choose for each non-pointed position whether to label it (to use `RCons`) or refrain from doing so (to use `Cons` instead). The easy way to remove this source of ambiguity is to require labelling universally. We arrive at a simpler definition:

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

This representation does indeed forbid the empty cycle and ensures that all cyclic lists have a unique representation.

2.2 Cyclic lists as a nested datatype

To derive a representation for cyclic lists that avoids a mixed-variant datatype and makes cycles really manipulable, we notice that the `RCons` constructor is intended to take as its second argument a lambda-expression whose bound variable acts a name for the position. Instead of using Haskell-level lambda-expressions, we could make this explicit by introducing variables and changing the type of `RCons` so that, instead of a list function, it expects a variable and a list possibly containing that variable, to be understood collectively as a formal lambda-abstraction. To avoid the problems of name-carrying syntax related to alpha-conversion, we opt for the de Bruijn notation. As shown in [7, 3], lambda-terms in de Bruijn notation admit a precisely typed representation as a nested datatype. ('Nested datatypes' [5] is a functional programming name for least or greatest fixpoints of rank-2 typeconstructors). The same technology can be applied here.

```
data Void

void :: Void -> a
-- illogically, Haskell rejects the empty case distinction
void _ = error "I will never be here"

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

Of the two versions, de Bruijn indices and de Bruijn levels, we use levels. The constructor `Var` signifies a backward pointer to form a cycle. The pointer to the first element of a cyclic list is represented by `Var Nothing`, to the second element by `Var (Just Nothing)`, etc. The extra parameter of the type constructor encodes the depth of the position. A complete cyclic list is of type `CList Void`, where

`Void` is the empty type and `void` is the empty function. This guarantees that we cannot have any "dangling pointers", i.e., pointers which point outside the list.

For instance, we can represent the cyclic lists from Fig. 1 as follows:

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

An important feature of this solution is that it only allows fixpoints of append-functions (represented by a sequence of `RCons`es ended by a variable pointing to the beginning of the sequence). Moreover, it turns out that one can define many useful cyclic-list-inspecting functions that do not fully unwind cycles.

2.3 Cyclic lists as a list algebra

Obviously, cyclic lists should carry a list algebra structure, and they do, but this is not entirely trivial, as consing an element with a cyclic list involves relabeling the positions.

```
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)
```

2.4 Cyclic lists as a list coalgebra

In order to interpret cyclic lists as infinite lists (i.e., to unwind them), we have to define a list coalgebra structure on it, i.e. functions `chead` and `ctail`, which compute the head and the tail of a non-empty cyclic list. The first is easy:

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

Note that the missing equation for the `Nil` constructor says that the `chead` function is partial. The only possible definition for `Var` is the empty function, as one cannot point back from the first position.

The `ctail` function is a little more complicated. As `chead`, it is undefined on `Nil` and, on the top level, it can only receive a `RCons`-list. However, it is not enough to return just the second argument of `RCons`. In the case of a full cycle, its tail is itself rotated to the left by one. This is illustrated in Fig. 2. If the list has a smaller cycle, then we can return the second argument, but we need to decrement the pointer, as the position it is pointing to now is one position to the left.

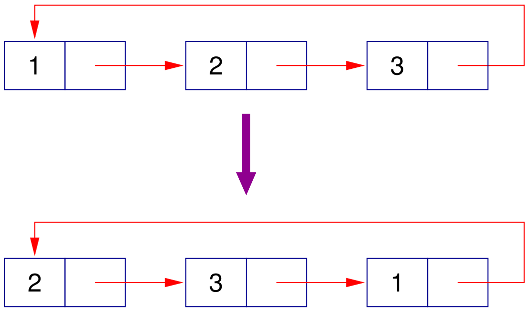

Figure 2. Tail of a cycle

```
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)
```

Now we can transform cyclic lists to possibly infinite lists by

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

This definition is legitimate, if the output type of lists is understood coinductively. (The input type of cyclic lists may be inductive or coinductive.) Indeed, `unwind` is really an unfold into the coinductive list type:

```
unwind = unfoldr cheadtail

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

2.5 Unfolding list coalgebras into cyclic lists

A salient feature of cyclic lists is that one can define unfolding of a list coalgebra into a cyclic list. Remember that a list coalgebra is essentially a state machine and the unfold of the list coalgebra computes the possibly infinite output trace of the state machine.

The idea with unfolding into a cyclic list is to keep a list of visited states and detect, if some state is revisited. This is possible, if the state space has a decidable (i.e., terminating) equality relation. The definition is the following:

```
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
```

In this definition, the output type of cyclic lists must be read as coinductive (possibly infinite cyclic lists).

A nice application is zipWith for finite (inductive) cyclic input lists. We use pairs of finite cyclic lists as the state space. This is fine, since equality of finite cyclic lists is decidable.

```
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'))
```

The same definition loses its sense, if we want understand the input types as coinductive instead, because the equality relation of possibly infinite cyclic lists is undecidable.

For the same reason, we cannot use `cunfoldr` for cyclification of possibly infinite lists. We would have to use possibly infinite lists as states, but cannot decide their equality.

Note that, in algebraically compact settings, where the distinction between inductive and coinductive types vanishes, we have just one datatype of (lazy) lists and one datatype of (lazy) cyclic lists, but cyclification is still impossible, since the datatype of lazy lists does not have a terminating equality.

2.6 Structured recursion/corecursion for cyclic lists

In the previous section, we did not explain why the definitions of `shift`, `csnoc` and `cunfoldL′` are legitimate—our definitions relied on general recursion, which is not an acceptable principle in total settings. This is best done resorting to structured recursion for finite cyclic lists resp. structured corecursion for possibly infinite cyclic lists.

The simplest structured recursion scheme, fold or iteration, is defined in Haskell as follows:

```
cfold :: (forall a . a -> g a)
        -> (forall a . g a)
        -> (forall a . Int -> g (Maybe a) -> g a)
        -> CList a -> g a
cfold v n r (Var z)      = v z
cfold v n r Nil          = n
cfold v n r (RCons x xs) = r x (cfold v n r xs)
```

The function `shift` is an exemplary example of a fold:

```
newtype CListMaybe a = CLM {unCLM :: CList (Maybe a) }

shift :: CList a -> CList (Maybe a)
shift = unCLM . cfold v n r where
             v z    = CLM (Var (Just z))
             n      = CLM Nil
             r x ys = CLM (RCons x (unCLM ys))
```

To justify `csnoc`, it is convenient to rely on a generalized fold combinator à la [6, 4, 2] (which is itself expressible via the fold combinator and right Kan extensions). This recursor takes a distributive law as one argument.

```
cefold :: (forall a. Maybe (h a) -> h (Maybe a))
     -> (forall a . h a -> g a)
     -> (forall a . g a)
     -> (forall a . Int -> g (Maybe a) -> g a)
     -> CList (h a) -> g a
cefold d v n r (Var z)      = v z
cefold d v n r Nil          = n
cefold d v n r (RCons x xs) = r x (cefold d v n r (fmap d xs))

csnoc :: Int -> CList (Maybe a) -> CList a
csnoc y = cefold id v n r where
             v Nothing  = RCons y (Var Nothing)
             v (Just z) = Var z
             n          = Nil
             r x ys     = RCons x ys
```

Notice that folds on cyclic lists do not always make sense as functions on possibly infinite lists. For instance, we can define a function which sums all elements in a cyclic list by

```
newtype K a = K Int

csum = cfold (\ x -> K 0) (K 0) (\ i (K j) -> K (i+j))
```

Obviously, it returns different results on the cyclic lists `RCons 1 (Var Nothing)` and `RCons 1 (RCons 1 (Var Nothing))`, but both correspond to the same infinite list $[1,1,1..]$.

The function $\mathtt{cunfoldL}'$ is an example of an unfold for cyclic lists.

```
cunfold ::
    (forall a. g a -> Either a (Maybe (Int, g (Maybe a))))
                                          -> g a -> CList a
cunfold lht c = case lht c of
          Left z              -> Var z
          Right Nothing       -> Nil
          Right (Just (x, c')) -> RCons x (cunfold lht c')

data St c a = St [c] [a] c

cunfoldL' :: Eq c => [c] -> [a] -> (c -> Maybe (Int, c))
                                           -> c -> CList a
cunfoldL' cs as ht c = cunfold lht (St cs as c) where
   lht (St cs as c) = case lookup c (zip cs as) of
      Nothing -> case ht c of
         Nothing      -> Right Nothing
         Just (x, c') -> let cs' = cs ++ [c]
                                 as' = Nothing : map Just as
                             in Right (Just (x, St cs' as' c'))
      Just a  -> Left a
```

3 CYCLIC BINARY TREES

The approach of the previous section scales up from lists to other datatypes, in particular to all polynomial datatypes.

To see this, we look at cyclic binary trees. Binary trees are non-linear (multiple paths down from the root) and this makes them more general than lists. By a cyclic tree we mean a tree where a path down from the root may end with a pointer back to some position in that path. (Pointers elsewhere are forbidden, we are interested in cycles in this paper, not in sharing.) An infinite tree can contain multiple cycles and the existence in an infinite tree of cycles does not mean it can be represented as a finite cyclic tree. For this, it is necessary that there is a cycle on every path of the tree.

The representation of the type of cyclic binary trees as a nested datatype is a direct generalization of the cyclic list type from the previous section.

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

Again, we can read this definition as inductive or coinductive depending on what we want. As an example, the cyclic tree from Fig. 3 is represented as follows:

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

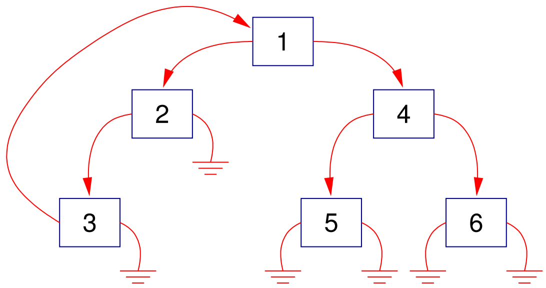

Figure 3. **A cyclic tree**

Cyclic trees are a tree algebra. One of the two operations is `Leaf`, the other is `cbin` defined as

```
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 z)           = VarT (Just z)
shiftT Leaf               = Leaf
shiftT (RBin x xsL xsR) = RBin x (shiftT xsL) (shiftT xsR)
```

Cyclic trees are also a tree coalgebra. Here, however, the situation is more complicated than in the case of lists because of the non-linearity of trees: 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. The left subtree of the cyclic tree from Fig. 3 is depicted on Fig. 4.

The central ingredients of the tree coalgebra structure of cyclic trees are defined as follows:

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

csubR :: CTree Void -> CTree Void
csubR (VarT z)          = void z
csubR (RBin x xsL xsR) = csnocR x xsL xsR

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
```

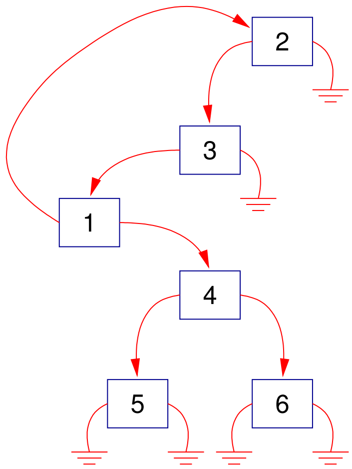

Figure 4. Left subtree of the cyclic tree from Fig. 3

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

4 GENERAL CYCLIC STRUCTURES

To finish, we sketch an implementation of general cyclic structures that is parametric in the base functor defining the depth-one structures.

In Haskell, the general definition of an inductive or coinductive type in terms of its base functor is the following.

```
data Mu f = In (f (Mu f))
```

For instance, the base functors for lists and binary trees can be defined as follows:

```
data L x = Nil  | Cons Int x

instance Functor L where
    fmap f Nil        = Nil
    fmap f (Cons i x) = Cons i (f x)

data T x = Leaf | Bin Int x x

instance Functor T where
```

```
    fmap f Leaf          = Leaf
    fmap f (Bin i xL xR) = Bin i (f xL) (f xR)
```

Hence, lists and binary trees are represented as `Mu L` and `Mu T` respectively.

General cyclic structures can be modelled in a similar way, but with an extra type variable argument for encoding the depth of a local structure in the global substructure (its distance from the root) and an extra constructor for backward pointers:

```
data Cyc f a = Var a
             | RIn (f (Cyc f (Maybe a)))
```

For instance, cyclic lists and binary trees can be represented as follows:

```
type CList = Cyc L
type CTree = Cyc T
```

Obviously, cyclic structures are an algebra of the base functor:

```
shift :: Functor f => Cyc f a -> Cyc f (Maybe a)
shift (Var z) = Var (Just z)
shift (RIn x) = RIn (fmap shift x)

cin :: Functor f => f (Cyc f Void) -> Cyc f Void
cin = RIn . fmap shift
```

They also have a coalgebra structure, but to describe this, we need to be able to work with contexts of positions in a depth-one structure in a general manner. This is achieved by using derivatives [12, 1]. In Generic Haskell, it would be possible to define the derivative of a regular functor by structural recursion over the expression for that functor. Here we take a simpler and more manual approach. We make a class definition which states the requirements on the derivative of a functor.

```
class (Functor f, Functor g) => Ctx f g | f -> g where
   distCtx :: f x -> f (g x, x)
   combCtx :: g x -> x -> f x
```

The member function `distCtx` pairs the value at every position in a depth-one structure with its surrounding one-hole context and the `combCtx` is the 'plug-in' operation which fills a one-hole context with a given value.

For instance, the derivatives of the base functors of list and binary tree types can be defined as follows:

```
data LCtx x = ConsCtx Int

instance Functor LCtx where
    fmap f (ConsCtx i) = ConsCtx i

instance Ctx L LCtx where
```

```
    distCtx Nil        = Nil
    distCtx (Cons i x) = Cons i (ConsCtx i, x)

    combCtx (ConsCtx i) x = Cons i x

data TCtx x = BinCtxL Int x | BinCtxR Int x

instance Functor TCtx where
    fmap f (BinCtxL i x) = BinCtxL i (f x)
    fmap f (BinCtxR i x) = BinCtxR i (f x)

instance Ctx T TCtx where
    distCtx Leaf          = Leaf
    distCtx (Bin i xL xR) = Bin i (BinCtxL i xR, xL)
                                  (BinCtxR i xL, xR)

    combCtx (BinCtxL i xR) xL = Bin i xL xR
    combCtx (BinCtxR i xL) xR = Bin i xL xR
```

Now, the coalgebra structure on cyclic structures is defined as follows:

```
cout :: Ctx f g => Cyc f Void -> f (Cyc f Void)
cout (Var z) = void z
cout (RIn x) = fmap (uncurry csnoc) (distCtx x)

csnoc :: Ctx f g => g (Cyc f (Maybe a)) -> Cyc f (Maybe a)
                                                      -> Cyc f a
csnoc ctx (Var Nothing)  = RIn (combCtx ctx (Var Nothing))
csnoc ctx (Var (Just z)) = Var z
csnoc ctx (RIn x)    = RIn (fmap (csnoc (fmap shift ctx)) x)
```

5 CONCLUSION

In this paper, we have demonstrated that cyclic lists and cyclic trees can be represented as nested datatypes. Our approach is rooted in the idea of Fegaras and Sheard to view cycles as explicit fixpoints, but corrects it, switching from their higher-order abstract syntax style presentation of the variable binding involved to a de Bruijn inspired version. Pleasingly, this simple modification yields representations that are free of the multiple shortcomings of the original proposal of Fegaras and Sheard and easy to manipulate and reason about. In the general case, one is forced to think in terms of of one-hole contexts of positions in depth-one structures. Here, the toolkit of derivatives of functors applies.

As future work, we want to extend our method to sharing. While cycles mean pointers to positions on the path from the root, sharing means pointers to position to the left from the path from the root. And we also want to develop a categorical account of the cyclic representations of rational and coinductive types, building on the groundwork by Ghani, Lüth and de Marchi [11].

Acknowledgements

We are grateful to Zhenjiang Hu for motivating this work.

Makoto Hamana was partially supported by the JSPS Grant-in-Aid for Scientific Research No. 16700005. Tarmo Uustalu and Varmo Vene were partially supported by the Estonian Science Foundation under grant No. 5567.

References

[1] Abbott, M., Altenkirch, T., McBride, C., Ghani, N.: δ for data: differentiating data structures. Fund. Inform. **65**(1–2) (2005) 1–28

[2] Abel, A., Matthes, R., Uustalu, T.: Iteration and coiteration schemes for higher-order and nested datatypes. Theor. Comput. Sci. **333**(1–2) (2005) 3–66

[3] Altenkirch, T., Reus, B.: Monadic presentations of lambda terms using generalized inductive types. CSL 1999: In: Flum, J., Rodrígues-Artalejo, M., Eds., Proc. of 13th Int. Wksh. on Computer Science Logic, CSL '99 (Madrid, Sept. 1999). Vol. 1683 of Lect. Notes in Comput. Sci. Springer-Verlag, Berlin (1999) 453–468

[4] Martin, C. E., Gibbons, J., Bayley, I.: Disciplined, efficient, generalised folds for nested datatypes. Formal Asp. of Comput. **16**(1) (2004) 19–35

[5] Bird, R., Meertens, L.: Nested datatypes. In Jeuring, J., ed.: Proc. of 4th Int. Conf. on Mathematics of Program Construction, MPC '98 (Marstrand, June 1998). Vol. 1422 of Lect. Notes in Comput. Sci. Springer-Verlag, Berlin (1998) 52–67

[6] Bird, R., Paterson, R.: Generalised folds for nested datatypes. Form. Asp. of Comput. **11**(2) (1999) 200–222

[7] Bird, R.S., Paterson, R.: De Bruijn notation as a nested datatype. J. of Funct. Program. **9**(1) (1999) 77–91

[8] Fegaras, L., Sheard, T.: Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space). In: Conf. Record of 23rd ACM SIGPLAN/SIGACT Symp. on Principles of Programming Languages, POPL '96 (St. Petersburg Beach, FL, Jan. 1996). ACM Press, New York (1996) 284–294

[9] Fiore, M., Plotkin, G.D., Turi, D.: Abstract syntax and variable binding. In: Proc. of 14th Ann. IEEE Symp. on Logic in Comp. Sci., LICS '99 (Trento, July 1999), IEEE Comput. Soc. Press, Los Alamitos, CA (1999) 193–202

[10] Freyd, P.J.: Recursive types reduced to inductive types. In: Proc. of 5th IEEE Ann. Symp. on Logic in Computer Science, LICS '90 (Philadelphia, PA, June 1990). IEEE Comput. Soc. Press, Los Alamitos, CA (1990) 498–507

[11] Ghani, N., Lüth, C., de Marchi, F.: Monads of coalgebras: rational terms and term graphs. Math. Struct. in Comput. Sci. **15**(3) (2005) 433–451

[12] McBride, C.: The derivative of a regular type is the type of its one-hole contexts. Manuscript (2000)

[13] Meijer, E., Hutton, G.: Bananas in space: Extending fold and unfold to exponential types. In: Conf. Record of 7th ACM SIGPLAN/SIGARCH and IFIP WG 2.8 Int. Conf. on Functional Programming Languages and Computer Architecture, FPCA '95 (La Jolla, San Diego, CA, June 1995). ACM Press, New York (1995) 324–333

[14] Turbak, F. A., Wells, J. B.: Cycletherapy: A prescription of fold and unfold on regular trees. In: Proc. of 3rd Int. ACM SIGPLAN Conf. on Principles and Practice of Declarative Programming, PPDP 2001 (Florence, Sept. 2001). ACM Press, New York (2001) 137–149