Electronic Notes in Theoretical Computer Science 86 No. 3 (2003)
URL: http://www.elsevier.nl/locate/entcs/volume86.html 15 pages

Automatic Visualization of Recursion Trees: a Case Study on Generic Programming [1]

Alcino Cunha [2]

Departamento de Informática
Universidade do Minho
4710-057 Braga, Portugal

Abstract

Although the principles behind generic programming are already well understood, this style of programming is not widespread and examples of applications are rarely found in the literature. This paper addresses this shortage by presenting a new method, based on generic programming, to automatically visualize recursion trees of functions written in Haskell. Crucial to our solution is the fact that almost any function definition can be automatically factorized into the composition of a fold after an unfold of some intermediate data structure that models its recursion tree. By combining this technique with an existing tool for graphical debugging, and by extensively using Generic Haskell, we achieve a rather concise and elegant solution to this problem.

1 Introduction

A generic or polytypic function is defined by induction on the structure of types. It is defined once and for all, and can afterwards be re-used for any specific data type. The principles behind generic programming are already well understood [2], and several languages supporting this concept have been developed, such as PolyP [13] or Generic Haskell [3]. Unfortunately, this style of programming is not widespread, and we rarely find in the literature descriptions of applications developed in these languages.

This paper addresses this shortage by presenting a case study on generic programming. The problem that we are trying to solve is to automatically and graphically visualize the recursion tree of a Haskell [14] function definition. This problem does not rise any particular difficulties, however it will be shown that the use of generic programming allows us to achieve a rather concise and

[1] This work was partially supported by FCT project POSI/CHS/44304/2002.
[2] Email: alcino@di.uminho.pt

elegant solution. This solution was integrated in a tool that is of practical interest for the area of program understanding, specially on an educational setting.

At the core of our solution lies an algorithm that automatically factorizes a recursive definition into the composition of a fold and an unfold of an intermediate data structure. This algorithm was first presented by Hu, Iwasaki, and Takeichi in [12], where it was applied to program deforestation. A well known side-effect of the factorization is that the intermediate data structure models the recursion tree of the original definition. The visualization of this structure is defined generically on top of GHood [19], a system to graphically trace Haskell programs.

This paper is structured as follows. In section 2 we informally show how the well-known fold and unfold functions on lists can be generalized to arbitrary data types. This generalization is essential for understanding the factorization algorithm. Section 3 introduces some theoretical concepts behind the idea of programming with recursion patterns. In section 4 we show how these recursion patterns can be defined once and for all using Generic Haskell. These generic definitions simplify the implementation, since we no longer have to derive the specific recursion patterns to operate on the intermediate data types. In section 5 we present the factorization algorithm very briefly. In section 6 we show how we can use GHood to observe intermediate data structures, and in section 7 we generalize the observation mechanism in order to animate any possible recursion tree. The last section presents some concluding remarks. We assume that the reader has some familiarity with the language Haskell.

2 Recursion Patterns Informally

Each inductive data type is characterized by a standard way of recursively consuming and producing its values according to its shape. The standard recursion pattern for consuming values is usually known as fold or catamorphism. In the case of lists this corresponds to the standard Haskell function `foldr`.

```
foldr :: (a -> b -> b) -> b -> [a] -> b
foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```

The generalization to other data types is straightforward. Let us suppose that we want to fold over binary trees.

```
data Tree a = Leaf | Node a (Tree a) (Tree a)
```

Similarly to `foldr`, this fold must receive as an argument the value to return when we reach a leaf, and a function to apply when consuming a node. This function has to process the result of two recursive calls since the data type is birecursive. In order to make explicit the duality with the (yet to be presented) unfolds, we group both parameters in a single function with the help of the data type Maybe.

```
foldT :: (Maybe (a,b,b) -> b) -> Tree a -> b
foldT g Leaf         = g Nothing
foldT g (Node x l r) = g (Just (x, foldT g l, foldT g r))
```

The dual of fold is the unfold or anamorphism. Although already known for a long time it is still not very used by programmers [6]. This recursion pattern encodes a standard way of producing a value of a given data type, and for lists it is defined as `unfoldr` in one of the standard libraries of Haskell.

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

The argument function `f` dictates the generation of the list. If it returns `Nothing` the generation stops yielding the empty list. Otherwise it should return both the value to put at the head of the list being constructed, and a seed to carry on with the generation. The generalization to trees is again straightforward.

```
unfoldT :: (b -> Maybe (a,b,b)) -> b -> Tree a
unfoldT h x = case (h x) of
                Nothing      -> Leaf
                Just (x,l,r) -> Node x (unfoldT h l) (unfoldT h r)
```

The composition of a fold after an unfold is known as a hylomorphism [17]. Note that since the fixed point operator can be defined as a hylomorphism, this recursion pattern can express any recursive function [18]. For a wide class of function definitions the intermediate data structures correspond to their recursion trees. For example, quick-sort can be easily implemented as a hylomorphism by using a binary search tree as intermediate data structure [1]. The unfold should build a search tree containing the elements of the input list, and the fold just traverses it inorder.

```
qsort :: (Ord a) => [a] -> [a]
qsort = (foldT g) . (unfoldT h)
    where h []     = Nothing
          h (x:xs) = Just (x, filter (<=x) xs, filter (>x) xs)
          g Nothing        = []
          g (Just (x,l,r)) = l ++ [x] ++ r
```

3 Data Types as Fixed Points of Functors

One of the tasks of the method presented in the previous section, is the explicit definition of the folds and unfolds for each new data type. However, it is possible to avoid this task by appealing to the theoretical concepts behind data types and recursion patterns. The framework of the following presentation is the category $\mathcal{CPO}$ of complete partial orders and continuous functions.

The first insight is that data types can be modeled as least fixed points of functors. Given a monofunctor F which is locally continuous, there exists a data type μF and two strict functions $\texttt{in}_F : F(\mu F) \to \mu F$ and $\texttt{out}_F : \mu F \to F(\mu F)$ which are each others inverse. The data type μF is the least fixed point of F, the functor that captures the signature of its constructors. The functions $\texttt{in}_F$ and $\texttt{out}_F$ are used, respectively, to construct and destruct values of the data type μF. At least since the work of Meijer and Hutton presented in [18] it is well known how these concepts can be implemented directly in Haskell. First we define an explicit fixpoint operator with the keyword `newtype` to enforce the isomorphism.

```
newtype Mu f = In {out :: f (Mu f)}
```

For each new data type it is necessary to define the functor that captures its signature (and declare it as an instance of the class `Functor`), and then apply `Mu` in order to obtain the data type. For example, the `Tree` data type presented in the previous section could be defined as follows.

```
data FTree a b = Leaf | Node a b b

instance Functor (FTree a)
    where fmap f Leaf = Leaf
          fmap f (Node x l r) = Node x (f l) (f r)

type Tree a = Mu (FTree a)
```

A possible example of a tree with just two integer elements is

```
tree :: Tree Int
tree = In (Node 2 (In (Node 1 (In Leaf) (In Leaf))) (In Leaf))
```

Under some particular strictness conditions, the F-algebra $(\mu F, \texttt{in}_F)$ is initial, and its carrier matches with the carrier of the final F-coalgebra $(\mu F, \texttt{out}_F)$. This means that given any other F-algebra (B, g) or F-coalgebra (A, h), `fold` g and `unfold` h are the unique functions that make the following diagrams commute.

$$\begin{tikzcd}
\mu F \arrow[d, dotted, "\texttt{fold}\, g"'] & F(\mu F) \arrow[l, "\texttt{in}_F"'] \arrow[d, "F(\texttt{fold}\, g)"] \\
B & FB \arrow[l, "g"]
\end{tikzcd}
\qquad
\begin{tikzcd}
A \arrow[r, "h"] \arrow[d, dotted, "\texttt{unfold}\, h"'] & FA \arrow[d, "F(\texttt{unfold}\, h)"] \\
\mu F \arrow[r, "\texttt{out}_F"'] & F(\mu F)
\end{tikzcd}$$

The unique definitions for folds and unfolds given by the diagrams can be directly translated to Haskell (in order to simplify the presentation we also give an explicit definition for hylomorphisms).

```
fold :: (Functor f) => (f b -> b) -> Mu f -> b
fold g = g . fmap (fold g) . out
```

```
unfold :: (Functor f) => (a -> f a) -> a -> Mu f
unfold h = In . fmap (unfold h) . h

hylo :: (Functor f) => (f b -> b) -> (a -> f a) -> a -> b
hylo g h = (fold g) . (unfold h)
```

These definitions can be seen as polytypic functions because they work for any data type, provided that it is modeled explicitly by the fixed point of a functor. The quick-sort example presented at the end of the previous section must be (slightly) adapted to this new methodology.

```
qsort :: (Ord a) => [a] -> [a]
qsort = hylo g h
    where h []     = Leaf
          h (x:xs) = Node x (filter (<=x) xs) (filter (>x) xs)
          g Leaf         = []
          g (Node x l r) = l ++ [x] ++ r
```

4 Generic Recursion Patterns

In order to use these recursion patterns, we still have to implement the map function for each functor. In order to avoid this task we will use Generic Haskell [3]. This language extends Haskell with polytypic features and originates on work by Ralf Hinze [8,9], where generic functions are defined by induction on the structure of types, by providing equations for the base types and type constructors (like products and sums). Given this information, a generic definition can be specialized to any Haskell data type (these are internally converted into sums of products). The specialization proceeds inductively over the structure of the type, with type abstraction, type application, and type-level fixed-point being interpreted as their value-level counterparts.

Generic Haskell automatically converts each data type to an isomorphic type that captures its sum of products structure and records information about the presence of constructors. This structure type only represents the top-level structure of the original type. The types in the constructors do not change, including the recursive occurrences of the original type. The functions that convert data types into their structure types (and the other way round) are also automatically created. The constructors of structure types are

```
data a :+: b = Inl a | Inl b -- sum
data a :*: b = a :*: b       -- product
data Unit = Unit             -- unit
data Con a = Con a           -- constructor
```

For example, `Tree'` is the structure type of our first declaration of `Tree`.

```
data Tree a  = Leaf | Node a (Tree a) (Tree a)
type Tree' a = Con Unit :+: Con (a :*: (Tree a :*: Tree a))
```

Generic functions are then defined over the constructors of structure types and base types. The specific function for a data type is obtained by composing the specialization of the generic function to its structure type with the respective conversion functions. For example, a generic map function is predefined in the Generic Haskell libraries as follows[3].

```
gmap {| Unit |} = id
gmap {| Int |}  = id
gmap {| :+: |} gmapA gmapB (Inl a)   = Inl (gmapA a)
gmap {| :+: |} gmapA gmapB (Inr b)   = Inr (gmapB b)
gmap {| :*: |} gmapA gmapB (a :*: b) = (gmapA a) :*: (gmapB b)
gmap {| Con c |} gmapA (Con a)       = Con (gmapA a)
```

Notice that different cases in `gmap` have different types (more precisely, the number of arguments equals the number of arguments of the type constructor). This is due to the fact that *polytypic values possess polykinded types* [9], a restriction that ensures that generic functions can be used with types of arbitrary kind. The type of `gmap` is specified in Generic Haskell as follows [4].

```
type Map {[ * ]} t1 t2 = t1 -> t2
type Map {[ k -> l ]} t1 t2 = forall u1 u2.
         Map {[ k ]} u1 u2 -> Map {[ l ]} (t1 u1) (t2 u2)
gmap {| t :: k |} :: Map {[ k ]} t t
```

For example, since the kind of `Tree` is `* -> *` (a constructor that receives a type as argument and produces a type), the type of the map function for binary trees can be determined by the following sequence of expansions.

```
Map {[ * -> * ]} Tree Tree
forall u1 u2 . Map {[ * ]} u1 u2 -> Map {[ * ]} (Tree u1) (Tree u2)
forall u1 u2 . (u1 -> u2) -> (Tree u1 -> Tree u2)
```

Since in Haskell all type variables are implicitly universally quantified, this type equals the expected one.

```
gmap {| Tree |} :: (a -> b) -> Tree a -> Tree b
```

In the case of the bifunctor `FTree`, of kind `* -> * -> *`, the map must receive two functions as arguments, one to be applied to the elements in the node, and another one for the recursive parameter.

```
gmap {| FTree |} :: (a -> b) -> (c -> d) -> FTree a c -> FTree b d
```

[3] The delimiters `{| |}` enclose a type argument. In a generic function definition they enclose the type index for a given case. They are also used in the so-called generic application to require the specialization of a generic function to a specific type. When `Con` is supplied as a type index argument it includes an extra argument that is bound to a value providing information about the constructor name, its arity, etc. This information is necessary to implement, for example, a generic `show` function. Similarly, the delimiters `{[ ]}` used in type definitions enclose a kind argument.

[4] In Generic Haskell, the `type` keyword allows one to define a type by induction on the structure of its kind.

Besides generic functions, we can also define generic abstractions when a type variable (of fixed kind) can be abstracted from an expression. Typically this involves applying predefined generic functions to the type variable. For example, our recursion patterns can be defined by generic abstractions as follows [10].

```
fold {| f :: * -> * |} :: (f b -> b) -> Mu f -> b
fold {| f |} g = g . gmap {| f |} (fold {| f |} g) . out

unfold {| f :: * -> * |} :: (a -> f a) -> a -> Mu f
unfold {| f |} h = In . gmap {| f |} (unfold {| f |} h) . h

hylo {| f :: * -> * |} :: (f b -> b) -> (a -> f a) -> a -> b
hylo {| f |} g h = (fold {| f |} g) . (unfold {| f |} h)
```

Given these recursion patterns we no longer have to define the map functions for the data types we declare. For example, a quick-sort for integer lists can now be defined as follows (`g` and `h` are exactly the same as before).

```
qsort :: [Int] -> [Int]
qsort = hylo {| FTree Int |} g h
    where ...
```

However, there is a problem in defining the original polymorphic `qsort`. Given a list of type `[a]`, the intermediate data structure should be a binary tree with elements of type `a`, defined as the fixed point of `FTree a`. If Generic Haskell allowed the definition of scoped type variables (an extension to Haskell 98 described in [15]), one could define `qsort` as follows.

```
qsort :: (Ord a) => [a] -> [a]
qsort (l::[a]) = hylo {| FTree a |} g h l
    where ...
```

Unfortunately, since Generic Haskell does not support this extension we had to resort to a less elegant solution in order to allow for polymorphism. Instead of having a single definition for hylomorphisms, we define different functions to be used with monofunctors (`hylo1`), bifunctors (`hylo2`), and so on. As we have seen in the previous section for the `FTree` example, a polymorphic data type is obtained as the fixed point of a monofunctor, which in turn results from sectioning a bifunctor with the type variable. This means that the `Functor` instances should treat the type variable as a type constant (notice that in the case of `FTree` the contents of the node were left intact). When defining the generic recursion patterns for bifunctors this implies that we will have to pass the identity function as first argument to the generic map. For example, the fold for bifunctors is defined as follows.

```
fold2 {| f :: * -> * -> * |} :: (f c b -> b) -> Mu (f c) -> b
fold2 {| f |} g = g . gmap {| f |} id (fold2 {| f |} g) . out
```

The polymorphic qsort can then be defined as

```
qsort :: (Ord a) => [a] -> [a]
qsort = hylo2 {| FTree |} g h
    where ...
```

5 Deriving Hylomorphisms from Recursive Definitions

It is possible to derive automatically a hylomorphism from almost any explicit recursive definition of a function [12]. The main restrictions to the function definitions are that no mutual recursion is allowed, the recursive function calls should not be nested (thus excluding, for example, the usual definition of the Ackermann function), and, if the function has more than one argument, it should only induct over the last one (although it can be a tuple), leaving the remaining unchanged. This restrictions guarantee that the intermediate data type is a fixed point of a polynomial functor (sum of products), and, as a side-effect, that it models the recursion tree of the original definition.

We will informally explain how the algorithm works by applying it to the explicitly recursive definition of the quick-sort function. Our presentation assumes that all bounded variables are uniquely named. This restriction is trivially verified in this case.

```
qsort []     = []
qsort (x:xs) = qsort (filter (<=x) xs) ++ [x] ++
               qsort (filter (> x) xs)
```

The goal is to derive a functor F, and functions g and h in order to obtain the following hylomorphism (note that after determining F one should choose the appropriate hylo according to its kind).

```
qsort = hylo {| F |} g h
```

The first step is to identify, for the right hand side of each clause, the recursive calls (shown in italic) and all variables that occur free in those terms outside of the recursive calls (shown underlined).

```
qsort []     = []
qsort (x:xs) = qsort (filter (<=x) xs) ++ [x] ++
               qsort (filter (> x) xs)
```

The definition of h is similar to that of qsort, but the right hand sides are replaced by new data constructors applied to the free variables and the arguments of recursive calls.

```
h []     = F1
h (x:xs) = F2 x (filter (<=x) xs) (filter (>x) xs)
```

In order to define g we first replace the recursive calls in the right hand sides of qsort by fresh variables, and use as arguments the right hand sides of h, but with the arguments of recursive calls replaced by the new variables.

```
g F1            = []
g (F2 x r1 r2) = r1 ++ [x] ++ r2
```

The arguments of the functor `F` are all the free variables plus a single recursive variable `r`. The constructors have the same definition as the arguments of `g`, but with all the fresh variables replaced by `r`.

```
data F x r = F1 | F2 x r r
```

Putting it all together, and using the adequate `hylo` (in this case `F` is a bifunctor), we get the same definition as before.

```
qsort = hylo2 {| F |} g h
    where h []     = F1
          h (x:xs) = F2 x (filter (<=x) xs) (filter (>x) xs)
          g F1           = []
          g (F2 x r1 r2) = r1 ++ [x] ++ r2
```

As expected, the intermediate data type models the recursion tree of the original definition. In this example we got binary trees since quick-sort is a birecursive function.

We implemented this algorithm straightforwardly using some libraries for parsing and pretty-printing Haskell, combined with a state monad for managing unique names and other global information. Note that the generic definitions of the recursion patterns can be used for every new functor `F`.

6 Observing Recursion Trees

GHood [19] is a graphical animation tool built on top of Hood [7] (*Haskell Object Observation Debugger*). Hood is a portable debugger for full Haskell, based on the observation of intermediate data structures. Essentially, it introduces the following combinator with a similar signature to `trace` (a debugging primitive offered by all major Haskell distributions), but with a more complex behavior.

```
observe :: (Observable a) => String -> a -> a
```

This function just returns the second argument, but as a side-effect it stores it into some persistent structure for later rendering. It behaves like an identity function that can remember its argument. The string parameter is just a label that allows one to distinguish between different observations in the same program. The main advantage of `observe` over `trace` is that it can be effectively used without changing the strictness properties of the observed program.

Instances of `Observable` for the standard types are predefined. Implementing new instances of this class is very simple due to the high-level combinators and monads included in the library. As an example, we present the implementation for lists that is predefined in the Hood libraries.

```
instance (Observable a) => Observable [a] where
  observer (a:as) = send ":"  (return (:) << a << as)
  observer []     = send "[]" (return [])
```

The function `send` collects temporal information (when the observation was done) that is not used by Hood. GHood uses this information to produce animations. Its graphical visualization system is based on a simple layout algorithm.

Since the derivation of hylomorphisms exposes the recursion tree as an intermediate data structure, it is enough to place an observation point in the hylomorphism definition in order to visualize it.

```
hylo {| f :: * -> * |} :: (f b -> b) -> (a -> f a) -> a -> b
hylo {| f |} g h = (fold   {| f |} g).(observe "Recursion Tree").
                   (unfold {| f |} h)
```

It is also necessary to implement instances of the class `Observable` for all possible intermediate data types. Since it is not known beforehand what kind of functor will be derived, it is necessary to provide a generic definition for `observer`. In the next section we provide that definition. For example, in the case of the functor `F` derived in the previous section, the final instances should behave as follows.

```
instance (Observable a) => Observable (Mu (F a)) where
    observer (In x) p = In (observer x p)

instance (Observable a, Observable b) => Observable (F a b) where
    observer F1          = send "" (return F1)
    observer (F2 x l r) = send "" (return F2 << x << l << r)
```

There are a couple of remarks to be made about these definitions. We bypass the observation of the fixpoint operator `Mu`. The goal is that the final animation should look the same as if the data type had been explicitly declared in a recursive fashion. The constructors of the data type are not displayed because their derived names are meaningless. Using these definitions, the recursion tree of `qsort [3,2,4,3,1]` is visualized as shown on figure 1.

Another problem with the previous instances is that they imply the definition of a particular instance for `Mu` applied to any possible derived functor, even if the implementation of the `observer` function is always the same. In the presence of an extension to the type system allowing for the specification of polymorphic predicates in an instance declaration, as presented in [11], we could have a single definition that would look like

```
instance (forall b . (Observable b) => Observable (f b))
                                    => Observable (Mu f) where
    observer (In x) p = In (observer x p)
```

Recently, Valery Trifonov has shown how this type of class constrains could, in general, be simulated in standard Haskell 98 [20]. However, for this par-

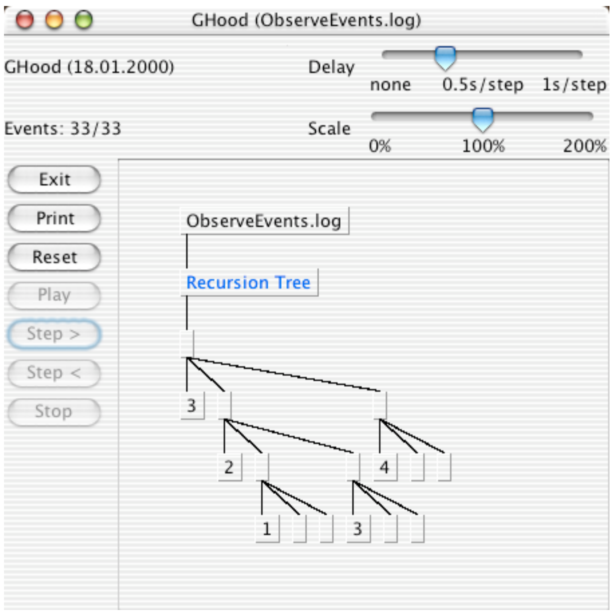

Fig. 1. Recursion tree of `qsort [3,2,4,3,1]`.

ticular example we will rely in a simpler solution suggested by Oleg Kiselyov [16]. First, we define a single instance of the `Observable` class for `Mu` as

```
instance (Observable (f (Mu f))) => Observable (Mu f) where
    observer (In x) p = In (observer x p)
```

This implies that, for a particular functor `f`, the instance of `Observable` must be made for `f (Mu f)`. The implementation of the `observer` function is the same as before. The final instance for the functor `F` is then

```
instance (Observable a) => Observable (F a (Mu (F a))) where
    observer F1         = send "" (return F1)
    observer (F2 x l r) = send "" (return F2 << x << l << r)
```

7 Generic Observations

In order to explain how we can define a generic observer we will first expand `<<` in the previous instance.

```
instance (Observable a) => Observable (F a (Mu (F a))) where
    observer F1         = send "" (return Leaf)
    observer (F2 x l r) = send "" (do {x’ <- thunk x;
                                       l’ <- thunk l;
                                       r’ <- thunk r;
                                       return (Node x’ l’ r’)})
```

In this definition we can see that `send` runs a state monad (`ObserverM`) in order to evaluate a term and simultaneously collect information for the renderer, and `thunk` is invoked for each child in a node. Even without presenting

```
type Observer {[ * ]} t = t -> ObserverM t
type Observer {[ k -> l ]} t = forall u . Observer {[ k ]} u ->
                                          Observer {[ l ]} (t u)

gobserverm {| t :: k |} :: Observer {[ k ]} t
gobserverm {| Unit |} = return
gobserverm {| Int |}  = return
gobserverm {| :+: |} oA oB (Inl a)    = do {a' <- (oA a);
                                            return (Inl a')}
gobserverm {| :+: |} oA oB (Inr b)    = do {b' <- (oB b);
                                            return (Inr b')}
gobserverm {| :*: |} oA oB (a :*: b) = do {a' <- (oA a);
                                            b' <- (oB b);
                                            return (a' :*: b')}
gobserverm {| Con c |} oA (Con a)     = do {a' <- (oA a);
                                            return (Con a')}
```

Fig. 2. Generic embedding into `ObserverM`

more details, it is clear that the monadic code follows the structure of the type, and so it is possible to define generically a function to embed a value into `ObserverM`, as shown in figure 2.

Notice that if we parameterize this function with a monofunctor `f` we get a function with type `(a -> ObserverM a) -> f a -> ObserverM (f a)`. By passing `thunk` as parameter we can get a generic definition of `observer` for monofunctors that behaves as expected.

```
gobserver {| f :: *->* |} :: Observable a => f a -> Parent -> f a
gobserver {| f |} x p = send "" (gobserverm {| f |} thunk x) p
```

Technically speaking, `gobserverm` is a monadic map for `ObserverM`. Given a functor F, and a monad M, the monadic map should transform functions of type $A \to M\ B$ into functions of type $F\ A \to M\ (F\ B)$. This concept was introduced by Fokkinga in [5]. For example, in the standard `Prelude` of Haskell, the function `mapM` implements the monadic map for lists. However, likewise to the regular map function, given a polynomial functor the monadic map can be defined by induction on its structure, thus being suitable for polytypic implementation. In fact, Generic Haskell provides a library `MapM` where this function is implemented with two different versions: one that evaluates the products left-to-right (`mapMl`) and another that evaluates them right-to-left (`mapMr`). If the monad is strong and commutative both yield the same result, but in general that is not the case. If we abstract the monad `ObserverM` in `gobserverm` we obtain the function `mapMl`. Given this equivalence, we can implement the generic observer as follows.

```
gobserver {| f :: *->* |} :: Observable a => f a -> Parent -> f a
gobserver {| f |} x p = send "" (mapMl {| f |} thunk x) p
```

As was the case for recursion patterns, we have to define different generic observers for monofunctors (`gobserver1`), bifunctors (`gobserver2`), etc. The instance implementation for functor `F` presented in the previous section can now be simply obtained as follows.

```
instance (Observable a) => Observable (F a (Mu (F a))) where
    observer = gobserver2 {| F |}
```

8 Conclusions and Future Work

The techniques presented in this paper were included in an application that, given a Haskell module, tries to derive hylomorphisms for all the functions declared in that module. When successful, each original definition is replaced by the corresponding new one, and the data type that models the recursion tree is declared, together with the appropriate instances of `Observable`. The resulting module is written in Generic Haskell and should be compiled into regular Haskell. The execution of each transformed function triggers, as a side-effect, a visualization of its recursion tree.

The main contribution of this paper is a new approach to visualizing recursion trees, that makes intensive use of generic programming in order to make the task of putting together previously developed tools and techniques easier. Other specific contributions are the clarification of the use of generic recursion patterns in polymorphic definitions, and the generic definition of observations to be used with `GHood`. In the past [4], we developed a preliminary solution to this problem, with a proprietary observation mechanism based on monadic recursion patterns. However, it had several problems, namely, it changed the strictness properties of the original definitions and it had no support for polymorphism.

Essentially, we have used generic programming to overcome the limitations of the Haskell `deriving` mechanism, which currently supports only a limited range of classes. There has been some research towards developing specific mechanisms to overcome these limitations. These mechanisms could have been used to achieve a similar effect. Hinze and Peyton Jones proposed *Derivable Type Classes* [11], a system based on the same theoretical concepts as Generic Haskell, but that only allows generic definitions in instance declarations. Unfortunately, it is not yet fully implemented in any Haskell distribution. An older system is DrIFT [21], a preprocessor that parses a Haskell module for special commands that trigger the generation of new code. This is a rather *ad hoc* mechanism, which is not as theoretically sound as generic programming.

In the future we intend to develop a new animation backend to replace GHood, in order to increase the understanding of the functions being observed. First, it should allow us to visualize the consumption of the intermediate data structure, showing how the final result is obtained from the recursion tree. It should also allow more control on the view of nested hylomorphisms. Sometimes, the parameters of a hylomorphism are also hylomorphisms, and to simplify the presentation these should only be visualized as requested. We also intend to develop more tailored animations to use when one determines that the function is an instance of a more specific recursion pattern, such as a function defined with an accumulation parameter.

Acknowledgments

I would like to thank José Barros, Jorge Sousa Pinto, José Bacelar Almeida, Alexandra Barros, and all the anonymous referees for their helpful comments on previous versions of this paper. Thanks also to João Saraiva for calling up the idea of hylomorphism visualization, and Andres Löh for his help with Generic Haskell.

References

[1] Lex Augusteijn. Sorting morphisms. In D. Swierstra, P. Henriques, and J. Oliveira, editors, *3rd International Summer School on Advanced Functional Programming*, volume 1608 of *LNCS*, pages 1–27. Springer Verlag, 1999.

[2] Roland Backhouse, Patrik Jansson, Johan Jeuring, and Lambert Meertens. Generic programming – an introduction. In D. Swierstra, P. Henriques, and J. Oliveira, editors, *3rd International Summer School on Advanced Functional Programming*, volume 1608 of *LNCS*, pages 28–115. Springer Verlag, 1999.

[3] Dave Clarke, Ralf Hinze, Johan Jeuring, Andres Löh, and Jan de Wit. The Generic Haskell user's guide. Technical Report UU-CS-2001-26, Utrecht University, 2001.

[4] Alcino Cunha, José Barros, and João Saraiva. Deriving animations from recursive definitions. In *Draft Proceedings of the 14th International Workshop on the Implementation of Functional Languages (IFL'02)*, 2002.

[5] Maarten Fokkinga. Monadic maps and folds for arbitrary datatypes. Memoranda Informatica 94–28, University of Twente, June 1994.

[6] Jeremy Gibbons and Geraint Jones. The under-appreciated unfold. In *Proceedings of the 3rd ACM SIGPLAN International Conference on Functional Programming (ICFP'98)*, pages 273–279. ACM Press, 1998.

[7] Andy Gill. Debugging Haskell by observing intermediate data structures. In G. Hutton, editor, *Proceedings of the 4th ACM SIGPLAN Haskell Workshop*, 2000.

[8] Ralf Hinze. A new approach to generic functional programming. In *Proceedings of the 27th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'00)*, pages 119–132. ACM Press, 2000.

[9] Ralf Hinze. Polytypic values possess polykinded types. In Roland Backhouse and José Nuno Oliveira, editors, *Mathematics of Program Construction (proceedings of MPC'00)*, volume 1837 of *LNCS*, pages 2–27. Springer-Verlag, 2000.

[10] Ralf Hinze, Johan Jeuring, and Andres Löh. Type-indexed data types. In Eerke Boiten and Bernhard Möller, editors, *Mathematics of Program Construction (proceedings of MPC'02)*, volume 2386 of *LNCS*, pages 148–174. Springer-Verlag, 2002.

[11] Ralf Hinze and Simon Peyton Jones. Derivable type classes. In Graham Hutton, editor, *Proceedings of the 2000 ACM SIGPLAN Haskell Workshop*, volume 41.1 of *ENTCS*. Elsevier, 2001.

[12] Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. Deriving structural hylomorphisms from recursive definitions. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP'96)*, pages 73–82. ACM Press, 1996.

[13] Patrik Jansson and Johan Jeuring. Polyp – a polytypic programming language extension. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 470–482. ACM Press, 1997.

[14] Simon Peyton Jones and John Hughes, editors. *Haskell 98: A Non-strict, Purely Functional Language.* February 1999.

[15] Simon Peyton Jones and Mark Shields. Lexically-scoped type variables. To be submitted to The Journal of Functional Programming, March 2002.

[16] Oleg Kiselyov. Re: Type class problem. Message posted on the Haskell mailing list, August 2003. http://www.mail-archive.com/haskell@haskell.org/msg13213.html.

[17] Erik Meijer, Maarten Fokkinga, and Ross Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In J. Hughes, editor, *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture (FPCA'91)*, volume 523 of *LNCS*. Springer-Verlag, 1991.

[18] Erik Meijer and Graham Hutton. Bananas in space: Extending fold and unfold to exponential types. In *Proceedings of the 7th ACM Conference on Functional Programming Languages and Computer Architecture (FPCA'95)*. ACM Press, 1995.

[19] Claus Reinke. GHood - graphical visualisation and animation of Haskell object observations. In Ralf Hinze, editor, *Proceedings of the 2001 ACM SIGPLAN Haskell Workshop*, volume 59 of *ENTCS*. Elsevier, 2001.

[20] Valery Trifonov. Simulating quantified class constrains. In *Proceedings of the ACM SIGPLAN 2003 Haskell Workshop*, pages 98–102. ACM Press, 2003.

[21] Noel Winstanley and John Meacham. *DrIFT User Guide (version 2.0rc3)*, 2002.