Deriving Animations from Recursive Definitions

Alcino Cunha, José Barros, and João Saraiva

Departamento de Informática, Universidade do Minho
4710-057 Braga, Portugal
Tel: +351253604470, Fax: +351253604471
{alcino,jbb,jas}@di.uminho.pt

Abstract. This paper describes a generic method to derive an animation from a recursive definition, with the objective of debugging and understanding this definition by expliciting its control structure. This method is based on a well known algorithm of factorizing a recursive function into the composition of the producer and the consumer of its call tree. We developed a systematic method to transform both the resulting functions in order to draw the tree step by step. The theory of data types as fixed points of functors, generic recursion patterns, and monads, are fundamental to our work and are briefly presented. Using polytypic implementations of monadic recursion patterns and an application to manipulate and generate graph layouts we developed a prototype that, given a recursive function written in a subset of Haskell, returns a function whose execution yields the desired animation.

1 Introduction

The importance of computers and the urge to produce software to assist crucial parts of our everyday tasks have led us to the existence of huge amounts of useful, indispensable, but undocumented programs. This fact emphasizes the need to invest in tools and methods of understanding legacy software.

Refinement techniques usually start from a clear and inefficient specification to reach an obscure and (hopefully) efficient program. One of these techniques is called *deforestation* [21], for it avoids the construction of intermediate tree-like structures. This is usually achieved by coding these structures in the control flow of the program.

The reverse abstraction process tries to get a more understandable specification from a given piece of code. Inspired by deforestation, a possible technique for this process could be *forestation*, aiming at factorizing explicitly recursive definitions into the composition of producers/consumers of tree-like structures, thus expliciting the control structure through these data structures.

This process has many possible applications. In this paper we use it to develop a mechanism for debugging and animating recursive definitions. The novelty of this approach lies in the fact that the control structure of a program is seen as a data structure – the call tree of the program. This structure corresponds to the runtime recursive invocations of functions and differs from the (static) call

graph of a program. Take, for instance the usual recursive definition of factorial function (`fact`). Its call graph has just one node and one arc, whereas, for a given argument `n`, the call tree of `fact n` is a list of `n+1` nodes (for example, the call tree of `fact 4` is presented on the right side of figure 1). If we take a bi-recursive function, like for instance the standard definition of the fibonacci function, the call tree is no longer a list but a binary tree (for example, on the left side of figure 1 is the call tree for `fib 4`).

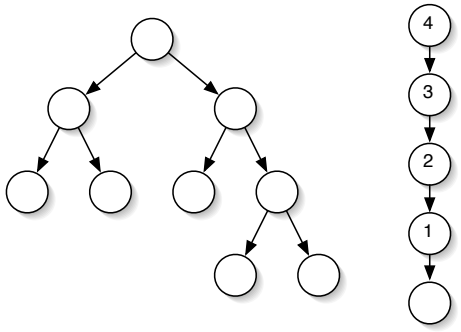


Fig. 1. Call trees of `fib 4` and `fact 4`

From a debugging point of view this approach is substantially richer than the traditional *break-point* strategy where a single node of the call trace is inspected at a time, forgetting the whole of its context in the graph (for a survey of debugging and tracing systems for lazy functional languages see [3]). For example, GHood [18] is a graphical visualization tool for tracing Haskell programs, that uses a simple tree layout algorithm to draw step by step the data structures being produced. In our approach we focus on the control structure of the program, and we do not intend to draw the produced data structures but to animate step by step how they are recursively produced.

All of our work makes intensive use of generic functions that capture typical patterns of recursion (such as the `foldr`). The use of these generic control structures is becoming important for several reasons [19]:

- Abstraction. They allow the specification of algorithms independently of the types of the data structures they should operate on.
- Genericity. They allow the statement, proof and use of generic theorems.
- Structure. Besides understanding and maintainability, in the case of functional languages the use of explicit structure, instead of an implicit one, eases the application of transformation rules.

These recursion patterns arise naturally in a categorical approach to functional programming. In section 2 we will briefly review the theory behind this approach. These presentation will be accompanied by examples and polytypic implementations using Haskell [1] and Generic Haskell [7].

In section 3 we will present a forestation algorithm previously developed by Hu el al [8], and give examples on how it can be used to decompose a recursive function into the composition of a function that builds the call tree and another one that "consumes" it, in order to achieve the desired result. These two functions are instances of generic recursion patterns presented in the preceding section. In section 4 we present the theoretic machinery needed to incorporate the animations into the factorized definition of the recursive function. This will be achieved by defining monadic versions of the recursion patterns. The animation technique is then presented in section 5. Finally, some conclusions are presented in section 6.

2 Data types as fixed points of functors

Given an endofunctor F on a category $\mathcal{C}$, an *F-algebra* is a strict function of type $F\ A \to A$, and an *F-coalgebra* is a, not necessarily strict, function of type $A \to F\ A$. The set A is called the *carrier* of the algebra. An *F-homomorphism* is a function $h : A \to B$ from an F-algebra $\varphi : F\ A \to A$ to an F-algebra $\psi : F\ B \to B$ that makes the following diagram commute.

$$\begin{array}{ccc} A & \xleftarrow{\varphi} & F\ A \\ {\scriptstyle h}\downarrow & & \downarrow{\scriptstyle F\ h} \\ B & \xleftarrow[\psi]{} & F\ B \end{array}$$

Given an endofunctor F we can construct a category $\mathcal{ALG}(\mathcal{F})$, whose objects are F-algebras and whose arrows are F-homomorphisms. Dually we have *F-cohomomorphisms* and the category $\mathcal{COALG}(\mathcal{F})$. In order to guarantee the soundness of some of the recursion patterns presented bellow, we will work in the category $\mathcal{CPO}$ of complete partial orders with continuous functions. An important fact about this category is that the carriers of the initial algebras and final coalgebras coincide. The initial (final) object in $\mathcal{ALG}(\mathcal{F})$ ($\mathcal{COALG}(\mathcal{F})$), whose carrier is denoted by μF, is called the *data type* defined by the endofunctor F. The initial algebra is denoted by $in_F : F(\mu F) \to \mu F$ and the final coalgebra by $out_F : \mu F \to F(\mu F)$.

A *polynomial functor* is either [2]:

- the identity functor I;
- a constant functor $\underline{A}$ for a given type A;
- the pointwise product or coproduct of other polynomial functors, defined by

$$(F + G)\ h = F\ h + G\ h$$
$$(F \times G)\ h = F\ h \times G\ h$$

- the composition of other polynomial functors.

Given a polynomial functor F, the initial object in the category of F-algebras is guaranteed to exist [12] (more precisely, the category must have colimits and countable chains, which is the case of $\mathcal{CPO}$). The most common data types can be expressed as fixed points of polynomial functors, as can be seen in figure 2.

Data Type	Functor
Natural numbers	$\underline{\mathbf{1}} + I$
Lists of type A	$\underline{\mathbf{1}} + \underline{A} \times I$
Non-empty lists of type A	$\underline{A} + \underline{A} \times I$
Binary shape trees	$\underline{\mathbf{1}} + I \times I$
Binary leaf trees of type A	$\underline{A} + I \times I$
Binary trees of type A	$\underline{\mathbf{1}} + \underline{A} \times I \times I$

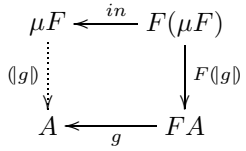
Fig. 2. Some functors of common data types

As shown for example in [14, 9], with Haskell, we can implement data types explicitly as fixed points of functors quite straightforwardly. For example, for the case of lists and binary shape trees we have

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

data FList a x = Nil | Cons (a,x)
instance Functor (FList a) where
    fmap f Nil = Nil
    fmap f (Cons (x,y)) = Cons (x, f y)
type List a = Mu (FList a)

list :: List Int
list = In (Cons (1, In (Cons (2, In Nil))))

data FTree x = Leaf | Branch (x,x)
instance Functor FTree where
    fmap f Leaf = Leaf
    fmap f (Branch (x,y)) = Branch (f x, f y)
type Tree = Mu FTree
```

The initiality of in_F in $\mathcal{ALG}(\mathcal{F})$ means that for any F-algebra $g : FA \to A$, there exists a unique F-homomorphism $h : \mu F \to A$ such that $h \circ in_F = g \circ Fh$. This homomorphism is called a catamorphism (cata) and is denoted by $(\!|g|\!)$[1]. That is, $(\!|g|\!)$ is the only function that makes the following diagram commute.

[1] Sometimes we will append the underlying functor of a recursion pattern explicitly in subscript. In this particular case we would state $(\!|g|\!)_F$.



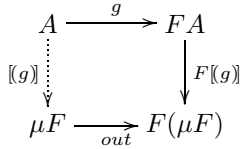
In Haskell, if we adhere to the definition of types as explicit fixed points of functors, we can have a single and straightforward (kind of) polytypic definition of a cata.

```
cata :: Functor f => (f a -> a) -> (Mu f -> a)
cata g = g . fmap (cata g) . out
```

Using this function, we can now define, for example, the sum of a list and the depth of a tree as follows:

```
sum :: List Int -> Int
sum = cata g where
      g Nil = 0
      g (Cons (a,r)) = a+r

depth :: Tree -> Int
depth = cata g where
        g Leaf = 1
        g (Branch (l,r)) = 1+(max l r)
```

Dually to catamorphisms, the finality of out_F in $\mathcal{COALG}(\mathcal{F})$ means that for any F-coalgebra $g : A \to FA$, there exists a unique F-cohomomorphism $h : A \to \mu F$ such that $out_F \circ h = Fh \circ g$. This homomorphism is called an anamorphism (ana) and is denoted by $[\!(g)\!]$. That is, $[\!(g)\!]$ is the only function that makes the following diagram commute.



The Haskell definition of ana and of a function that generates a list with all numbers from a given parameter toward zero follows.

```
ana :: Functor f => (a -> f a) -> (a -> Mu f)
ana g = In . fmap (ana g) . g

downto :: Int -> List Int
downto = ana g where
         g 0 = Nil
         g (n+1) = Cons (n+1, n)
```

An hylomorphism (hylo) is just the composition of a catamorphism and an anamorphism.

$$[\![g, h]\!] = (\!|g|\!) \circ [\!(h)]\!]$$

The definition of hylo and, for example, of the fibonacci function is presented bellow.

```
hylo :: Functor f => (f b -> b) -> (a -> f a) -> (a -> b)
hylo g h = (cata g) . (ana h)

fib :: Int -> Int
fib = hylo g h where
      h 0 = Leaf
      h 1 = Leaf
      h (n+2) = Branch (n+1, n)
      g Leaf = 1
      g (Branch (l,r)) = l+r
```

Generic Haskell [7] is an extension of Haskell that allows us to explicitly define polytypic functions[2]. With this extension we no longer need to explicitly define the maps for the functors, because it is possible to define a generic polytypic map function for polynomial functors (in fact, it is already implemented in the standard distribution as `gmap` in the `Map` library). Using Generic Haskell the definition of `cata` becomes

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

Note that now the definition is parametrized explicitly by the functor `f` that generates the data type. The term `gmap {| f |}` represents the particular instance of the map function to the functor `f`. Using this version of `cata`, the function `depth` can be defined as

```
depth :: Tree -> Int
depth = cata {| FTree |} g where
        g Leaf = 1
        g (Branch (l,r)) = 1+(max l r)
```

where `cata {| FTree |}` represents the instance of `cata` to the type `Tree`.

3 Deriving hylos from recursive definitions

For polynomial functors it is possible to derive automatically hylomorphisms from explicit recursive definitions of functions [8]. This derivation will factorize

[2] We will not present the details of Generic Haskell since we only use it in a very simple and straightforward way, and we believe that our examples are clear given the prior context.

a recursive function into the desired producer (ana) and consumer (cata) of its call tree. The algorithm proposed by these authors is not limited to recursive functions inducting over a single data structure. The language used to describe recursive functions is presented on figure 3. Recursive function calls can not be nested and can only occur in the terms of the alternatives in the definition body. In order to simplify the presentation, the authors restricted themselves to single-recursive datatypes and functions without mutual recursion. A work with similar objectives had already been developed by Launchbury and Sheard [11], but only for recursive definitions that can be specified in build-cata forms.

$$
\begin{array}{rll}
\mathit{decl} ::= & v = b & \text{function definition} \\
b ::= & \lambda v_s.\, \text{case}\, t_0\, \text{of}\, r & \text{definition body} \\
v_s ::= & v | (v_1, \ldots, v_n) & \text{argument} \\
r ::= & p_1 \rightarrow t_1; \ldots; p_n \rightarrow t_n & \text{alternatives} \\
t ::= & v & \text{variable} \\
| & (t_1, \ldots, t_n) & \text{term tuple} \\
| & vt & \text{function application} \\
| & Ct & \text{constructor application} \\
p ::= & Cp & \text{pattern} \\
| & (p_1, \ldots, p_n) & \text{pattern tuple} \\
| & v & \text{variable}
\end{array}
$$

Fig. 3. A language of recursive definitions

For example, using a subset of Haskell similar to the language defined in figure 3, the standard recursive definition of the fibonacci function is

```
fib = \n -> case n of 0 -> 1
                      1 -> 1
                      x -> plus (fib (sub (x,1)), fib (sub (x,2)))
```

where `plus` and `sub` are uncurried versions of operators `+` and `-`.

The derivation algorithm transforms the right hand side of a typical recursive definition

$$f = \lambda v_s.\, \text{case}\, t_0\, \text{of}\, p_1 \rightarrow t_1; \ldots; p_n \rightarrow t_n$$

into the form of $\phi \circ Ff \circ \psi$, so that $f = [\![\phi, \psi]\!]$. The trick is to transform each term t_i into a suitable $g_i t'_i$, in order to extract the functions g_i out and obtain a compositional description of f.

$$f = [g_1, \ldots, g_n] \circ (\lambda v_s.\, \text{case}\, t_0\, \text{of}\, p_1 \rightarrow (1, t'_1); \ldots; p_n \rightarrow (n, t'_n))$$

Additionally, if each g_i can be expressed as $\phi_i \circ F_i f$, where F_i is some functor, we obtain

$$f = [\phi_1, \ldots, \phi_n] \circ (F_1 + \ldots + F_n)\, f \circ (\lambda v_s.\, \text{case}\, t_0\, \text{of}\, p_1 \rightarrow (1, t'_1); \ldots; p_n \rightarrow (n, t'_n))$$

which is equivalent to

$$f = [\![[\phi_1, \ldots, \phi_n], \lambda v_s.\, \mathsf{case}\, t_0\, \mathsf{of}\, p_1 \to (1, t'_1); \ldots; p_n \to (n, t'_n)]\!]$$

The core of the algorithm consists in determining a function ϕ_i, a functor F_i, and a new term t'_i from each term t_i, so that $t_i = (\phi_i \circ F_i\, f)\, t'_i$. Informally, for each t_i, the algorithm proceeds as follows:

1. Identify all occurrences of recursive calls to f in t_i: $f\, t_{i_1}, \ldots, f\, t_{i_l}$.
2. Identify all free variables in t_i, but do not look in $t_{i_1}, \ldots, t_{i_l}$: $v_{i_1}, \ldots, v_{i_k}$.
3. Define t'_i by tupling the free variables obtained in the previous step, and all the terms that are arguments of the recursive calls obtained in step 1: $t'_i = (v_{i_1}, \ldots, v_{i_k}, t_{i_1}, \ldots, t_{i_l})$.
4. Define F_i as
$$F_i = \underline{\Gamma(v_{i_1})} \times \ldots \times \underline{\Gamma(v_{i_k})} \times I_1 \times \ldots \times I_l$$
where $I_1 = \ldots = I_l = I$ and Γ returns the type of the given variable.
5. Define ϕ_i by abstracting all recursive function calls in t_i as
$$\phi_i = \lambda(v_{i_1}, \ldots, v_{i_k}, v'_{i_1}, \ldots, v'_{i_l})\,.\, t_i[f\, t_{i_1} \mapsto v'_{i_1}, \ldots, f\, t_{i_l} \mapsto v'_{i_l}]$$
where $v'_{i_1}, \ldots, v'_{i_l}$ are fresh variables introduced to replace the occurrences of $f\, t_{i_1}, \ldots, f\, t_{i_l}$.

This algorithm is the main step of an automatic transformation system named HYLO [16], which performs fusion transformation in a calculational approach based on the acid rain theorem [20].

The polytypic versions of the recursion patterns are very well suited to be used in conjunction with this algorithm, because it also derives the functor that generates the data type of the call tree. For example, the explicitly recursive definition of the fibonacci function could be automatically transformed into the following equivalent hylo, where `F` is the functor that defines the intermediate data type `D` that models the call tree:

```
data F x = C1 | C2 | C3 (x,x)
type D = Mu F

fib = hylo {| F |} g h where
  g = \p -> case p of C1 -> 1
                      C2 -> 1
                      C3 p3 -> (\(v1,v2) -> plus (v2,v1)) p3
  h = \n -> case n of 0 -> C1
                      1 -> C2
                      x -> C3 (sub (x,2),sub (x,1))
```

Let us compare it with the definition presented in the previous section. The main difference is the existence of two kinds of leaves in the intermediate shape tree due to the existence of two base cases in the recursive definition.

4 Monadic recursion operators

After being proposed by Moggi [15], monads were introduced in functional programming by Wadler [22] as a means to structure programs that produce effects. In this context a monad is usually defined by a so-called Kleisli triple $(M, \mathtt{unit}, -^\star)$ over a category $\mathcal{C}$, where $M : Obj(\mathcal{C}) \to Obj(\mathcal{C})$ is the restriction of a functor M to objects, $\mathtt{unit} : I \dot{\to} M$ is a natural transformation, and $-^\star$ is a extension operator that, given $f : A \to MB$ yields $f^\star : MA \to MB$, such that the following equations hold: $\mathtt{unit}^\star_A = id_{MA}$, $f^\star \circ \mathtt{unit}_A = f$, and $f^\star \circ g^\star = (f^\star \circ g)^\star$.

Assuming that a computation that produces a value of type A has type MA, `unit` is a function that transforms a value into a computation that returns that value without producing effects, and the operator $-^\star$ allows one to compose monadic functions passing the effect around. Given $f : A \to MB$ and $g : B \to MC$, we define the monadic composition $g \bullet f$ as $g^\star \circ f$. The above mentioned laws allow one to form a Kleisli category $\mathcal{C}_M$, with the same objects as $\mathcal{C}$, monadic functions as morphisms, `unit` as identity, and $- \bullet -$ as composition.

Monadic catas where introduced by Fokkinga [5] after lifting most of the concepts presented in section 2 for the Kleisli category (see [13] for a more practical approach). First, we need to define the lifting of a function $f : A \to B$ as $\widehat{f} = \mathtt{unit} \circ f$. Then, we also need to define how to lift a functor F to the Kleisli category. This lifting is called *monadic extension* of F, and will be denoted by $\widehat{F}$.

The operation of $\widehat{F}$ on objects is equal to that of F. Given a monadic function $f : A \to MB$ it yields a function $\widehat{F}f : FA \to M(FB)$. The problem of determining the monadic extension $\widehat{F} : \mathcal{C}_M \to \mathcal{C}_M$ is equivalent to the determination of a natural transformation $\delta^F : FM \dot{\to} MF$, that distributes a monad over a functor: $\widehat{F}f = \delta^F \circ Ff$ and $\delta^F = \widehat{F}id$. For polynomial functors δ^F can be defined by induction on the structure of F, provided that a mechanism to distribute the monad over the product is given [5].

Given a *monadic F-algebra* $g : FA \to MA$, there exists a unique homomorphism $h : \mu F \to MA$ that satisfies the following equation:

$$h \bullet \widehat{in_F} = g \bullet \widehat{F}h$$

This homomorphism is called a *monadic cata* and is denoted by $(\!|g|\!)^M_F$. The existence and uniqueness of h follows from the fact that the previous equation can be reduced to the following homomorphism between to normal algebras.

$$h \circ in_F = (g \circ \delta^F) \circ Fh$$

This equivalence also states that every monadic cata is just a special case of a cata, hence

$$(\!|g|\!)^M_F = (\!|h \bullet \delta^F|\!)_F$$

Once again, using Generic Haskell we can define the monadic cata straightforwardly, because, as mentioned above, the monadic extension of a functor (or

monadic map) can be defined inductively on its structure. Two versions of the monadic map are implemented in the library `MapM`: one that evaluates the products left-to-right (`mapMl`) and another that evaluates them right-to-left (`mapMr`). Technically, if the monad is strong and commutative both yield the same result, but in general that is not the case. Equipped with a polytypic monadic map, the definition of the monadic map is

```
mcata {| f :: * -> * |} :: (Functor m, Monad m) =>
                           (f a -> m a) -> (Mu f -> m a)
mcata {| f |} g =
  g @@ (mapMl {| f |} (mcata {| f |} g)) @@ (return . out)
```

where `@@` is an infix operator that implements the Kleisli composition $- \bullet -$.

Monadic anas and monadic hylos where extensively studied by Pardo [17]. The main problem with monadic anas is that the lifting of normal anas to the Kleisli category does not yield unique solutions, and thus, given a *monadic F-coalgebra* $g : A \to M(FA)$, the *monadic ana* $[\![g]\!]_F^M : A \to M\mu F$ is defined as the least monadic homomorphism between g and $\widehat{out_F}$. This homomorphism can be obtained as the least fixed point of the following function:

$$\lambda f \,.\, \widehat{in_F} \bullet \widehat{F} f \bullet g$$

This least fixed point can be defined in Generic Haskell as follows:

```
mana {| f :: * -> * |} :: (Functor m, Monad m) =>
                          (a -> m (f a)) -> (a -> m (Mu f))
mana {| f |} g =
  (return . In) @@ (mapMl {| f |} (mana {| f |} g)) @@ g
```

In [17], Pardo presents two possible definitions of a monadic hylo. The first, and most general of both, is just the expected Kleisli composition of a monadic cata and a monadic ana. That is, given a monadic algebra $g : FA \to MA$ and a monadic coalgebra $h : B \to M(FB)$, a monadic hylo is defined as

$$[\![g, h]\!]_F^M = (\!|g|\!)_F^M \bullet [\![h]\!]_F^M$$

The problem with this definition is that it can not always be transformed into a function that avoids the construction of the intermediate data structure, as is the case with normal hylos. With some monads the effects produced separately during the construction and the destruction of the intermediate data structure would become interleaved. In order to overcome this problem, Pardo introduced a new definition of monadic hylo as a composition of a monadic ana and the lifting of a cata. Given an algebra $g : FA \to A$ and a monadic coalgebra $g : B \to M(FB)$ this restricted version of a monadic hylo is defined as

$$\langle\!| g, h |\!\rangle_F^M = M(\!|g|\!)_F \circ [\![h]\!]_F^M$$

With this definition it is possible to define a factorization law, that transforms an hylo into a single function. If M is a strictness-preserving functor then $\langle\!| g, h |\!\rangle_F^M$

is equal to the least fixed point of

$$\lambda f \,.\, \widehat{g} \bullet \widehat{F} f \bullet h$$

However, none of these definitions of monadic hylo is general enough to represent the animations that we will derive in the next section. Specifically, what we need is a factorized version similar to the previous one, but with effects both before and after the recursive call of the function. For the monads where that is possible, what we need is the factorized version of $[\![g, h]\!]_F^M$. This kind of monadic hylo will be denoted by $[\![g, h]\!]_F^M$, and, given a monadic algebra $g : FA \to MA$ and a monadic coalgebra $h : B \to M(FB)$, is equal to the least fixed point of

$$\lambda f \,.\, g \bullet \widehat{F} f \bullet h$$

This definition for monadic hylo has been already used by other authors, as for example in [4]. Its definition in Generic Haskell is

```
mhylo {| f :: * -> * |} :: (Functor m, Monad m) =>
                           (f b -> m b) -> (a -> m (f a)) ->
                           (a -> m b)
mhylo {| f |} g h =
  g @@ (mapMl {| f |} (mhylo {| f |} g h)) @@ h
```

5 Animating recursive functions

The derivation of animations for recursive functions proceeds according to the following steps:

- First, given a recursive definition of a function expressed in the syntax of figure 3 we derive an hylomorphism $[\![g, h]\!]_F$ using the algorithm of section 3.
- Then we will define an appropriate monad M and lifting functions $\alpha : (FA \to A) \to (FA \to MA)$ and $\beta : (A \to FA) \to (A \to M(FA))$ that will encode the desired effects of animation. In particular we developed an animation using Graphviz [6], a collection of tools for manipulating graph structures and generating graph layouts.
- Finally, the derived hylo will be transformed into the monadic hylo

  $$[\![\alpha(g), \beta(h)]\!]_F^M$$

  whose execution yields the animation of the calculation process for a given parameter.

The animation in Graphviz is implemented by a script that we developed in its internal language `lefty` [10]. This script needs two files with the following information:

– The first file has all the information needed in order to draw, step by step, the intermediate data structure produced by the anamorphism. For each node generated the file must contain its unique identifier, the value of the parameter that originated it, the value that is contained in it, and, if applicable, the identifier of its parent node (needed to draw an arrow linking both nodes). The nodes are drawn in the order presented in the file. First the node is labeled with the parameter used to generate it, and afterwards that label is replaced by the value in it.
– The second file records the "consumption" of that data structure during the catamorphism. For each visited node it must contain its unique identifier and the outputed value. Once again, the animation proceeds according to the order of the file. For sake of clarity, instead of eliminating each node as the computation proceeds, we opted instead to redraw them with a different color and replace the label by the outputed value.

In order to perform IO and simultaneously carry a state around while traversing the intermediate data structures, we need a monad that combines both functionalities. This can be achieved through the monad transformer `StateT` as shown in [9], that parametrizes a state monad by any other inner monad, that in our case will be the monad `IO`. The definition of `StateT` is

```
newtype StateT s m a = StateT { runStateT :: s -> m (a,s) }
```

The state of the monad has the following components:

– `seed` is used to generate unique identifiers to name the nodes in the trees.
– `path` stores the path along the tree departing from the root node. This path must be recorded in order to enable one to access, at each node, the identifier of its parent.
– `graph` and `moves` are handles to the two files mentioned above.

Its Haskell definition is

```
data MyState = MyState { seed :: Int, path :: [Int],
                         graph :: Handle, moves :: Handle }
```

Given this monad, the transformer of the anamorphism proceeds as follows:

1. Lift the result of applying the gene to the monad.
2. Determine an unique identifier to the current node, by appealing to the seed stored in the state, and increment this seed (function `incSeed`).
3. Write to the file used to draw the intermediate structure the requested information. The identifier of the parent is located at the head of the path. The function `showValue` returns a textual representation of the value that should be contained in the node. Currently, we are using a generic function that besides representing the value contained in the node, also represents the parameters that will be used to generate the node's subtrees delimited by vertical bars.

4. Adjust the path before returning, by inserting the identifier at the head of the path (function `pushNode`).

In Haskell this transformer is implemented as follows:

```
beta :: Show a => (a -> f a) -> (a -> StateT MyState IO (f a))
beta g = \x -> do y <- return (g x)
                  i <- gets seed
                  modify incSeed
                  h <- gets graph
                  liftIO (hPutStrLn h (show i))
                  liftIO (hPutStrLn h (show x))
                  liftIO (hPutStrLn h (showValue y))
                  p <- gets path
                  liftIO (hPutStrLn h (if (null p)
                                       then ""
                                       else (show (head p))))
                  modify (pushNode i)
                  return y
```

The transformer of the catamorphism performs the following actions:

1. Lift the result of applying the gene to the monad.
2. Determine the unique identifier of the current node, stored at the head of the path.
3. Write to the file that records the consumption of the intermediate data structure this identifier and the outputed value at this point.
4. Adjust the path before returning, by eliminating the identifier at the head of the path (function `popNode`).

In Haskell this transformer is implemented as follows:

```
alpha :: (Show a) => (f a -> a) -> (f a -> StateT MyState IO a)
alpha g = \x -> do y <- return (g x)
                   i <- gets (head . path)
                   h <- gets moves
                   liftIO (hPutStrLn h (show i))
                   liftIO (hPutStrLn h (show y))
                   modify popNode
                   return y
```

The need to use a factorized version of the monadic hylo is due only to the need of recording the path from the root node. Since the `IO` operations use different files in the ana and in the cata, their order would be preserved if we used the first definition of monadic hylo. Figures 4 and 5 present the step by step animation produced by our prototype for the recursive fibonacci function presented in section 3, when supplied with the parameter 4.

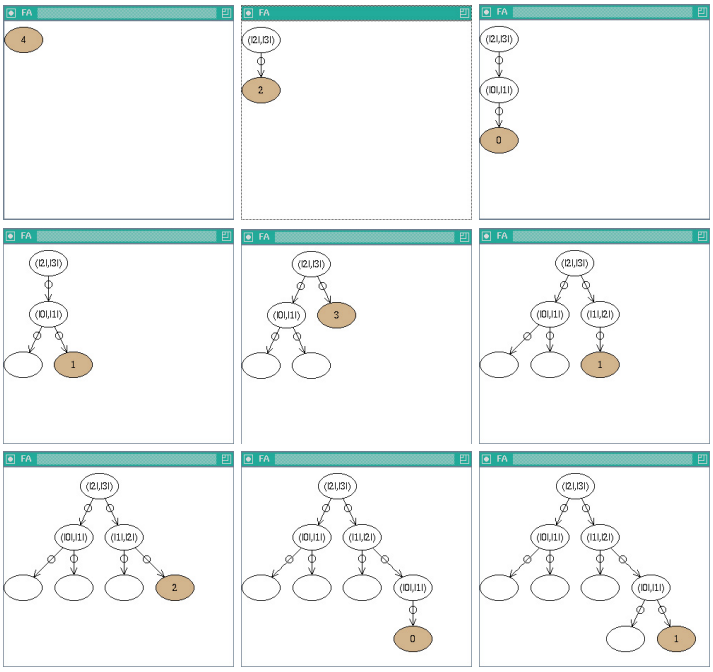


Fig. 4. Anamorphism of `fib 4`

6 Conclusions and future work

Based on well known concepts of the theory of data types as fixed points of functors, generic recursion patterns, and monads, and based on a straightforward implementation of these concepts using Generic Haskell, we developed a very simple mechanism to animate the calculation process of recursive functions. Our mechanism has the advantage that one does not need to rewrite the original program in order to include explicit debugging instructions. Since that the animation implementation is contained only on the (co)algebras' transformers, it is very simple to develop new kinds of animations. For example, we could easily develop a text based tracing system or animations for other visualization tools other then Graphviz.

Our technique relies heavily on the algorithm presented in [8] to derive hylomorphisms from recursive definitions. Therefore, we intend to improve this algorithm in order to cover a wider class of recursive programs, possibly by allowing non-polynomial data types as intermediate structures.

We also need to improve the animation mechanism in order to cover hylomorphisms whose algebras or coalgebras are also defined recursively. For example, the insertion sort algorithm when represented by an hylo has an algebra that is a cata (whose role is to insert an element in an ordered list). One can not understand very well its behavior if this algebra is not animated too. We intend to

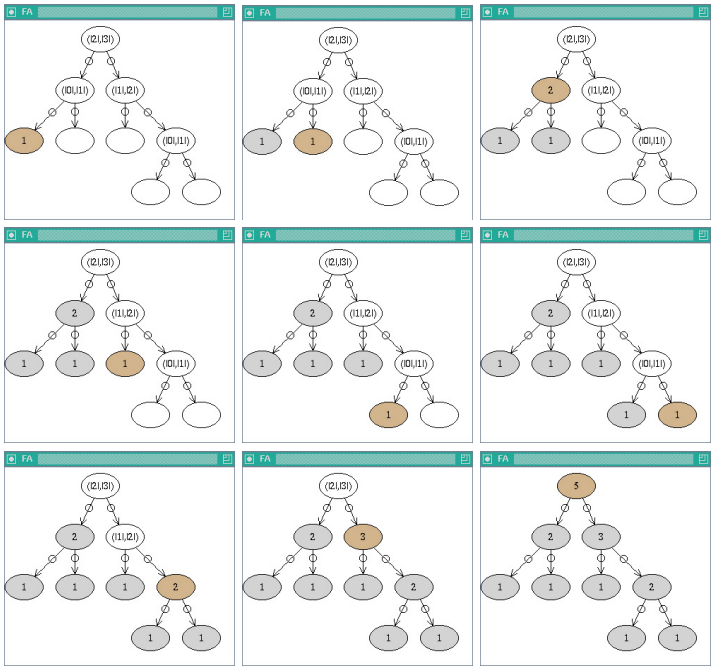


Fig. 5. Catamorphism of `fib 4`

deal with this kind of situations by developing a more interactive animation tool that allows one to see the calculation process of a specific node by clicking on it. In order to increase the understanding of the input function, we will also try to apply the acid rain theorem [20] backwards to the resulting hylomorphism, splitting it in the composition of (hopefully) simpler hylos that communicate through intermediate data structures.

References

1. L. Augustsson, D. Barton, B. Boutel, W. Burton, J. Fasel, K. Hammond, R. Hinze, P. Hudak, T. Johnsson, M. Jones, J. Launchbury, E. Meijer, J. Peterson, A. Reid, C. Runciman, P. Wadler, S. Jones (editor), and J. Hughes (editor). Haskell 98: A non-strict, purely functional language. Technical report, 1998.
2. Richard Bird and Oege de Moor. *Algebra of Programming*. Prentice Hall, 1997.
3. Olaf Chitil, Colin Runciman, and Malcolm Wallace. Freja, Hat and Hood – a comparative evaluation of three systems for tracing and debugging lazy functional programs. In M. Mohnen and P. Koopman, editors, *Proceedings of the 12th International Workshop on Implementation of Functional Languages*, volume 2011 of *LNCS*, pages 176–193. Springer-Verlag, 2001.
4. Tyng-Ruey Chuang and Shin-Cheng Mu. Out-of-core functional programming with type-based primitives. In *Proceedings of the 2nd International Workshop on Practical Aspects of Declarative Languages*, 2000.

5. Maarten Fokkinga. Monadic maps and folds for arbitrary datatypes. Memoranda Informatica 94–28, University of Twente, June 1994.
6. Emden Gansner and Stephen North. An open graph visualization system and its applications to software engineering. *Software - Practice and Experience*, 1999.
7. 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.
8. 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.
9. Mark P. Jones. Functional programming with overloading and higher-order polymorphism. In J. Jeuring and E. Meijer, editors, *First International Spring School on Advanced Functional Programming Techniques*, number 925 in LNCS, pages 97–136. Springer-Verlag, 1995.
10. Eleftherios Koutsofios. *Editing Pictures with lefty*, 1996.
11. John Launchbury and Tim Sheard. Warm fusion: Deriving build-catas from recursive definitions. In *Proceedings of the ACM Conference on Functional Programming Languages and Computer Architecture*, pages 314–323, 1995.
12. Grant Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14(2–3):255–279, October 1990.
13. Erik Meijeir and Johan Jeuring. Merging monads and folds for functional programming. In J. Jeuring and E. Meijer, editors, *First International Spring School on Advanced Functional Programming Techniques*, number 925 in LNCS, pages 228–266. Springer-Verlag, 1995.
14. 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.
15. Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93:55–92, 1991.
16. Yoshiyuki Onoue, Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. A calculational fusion system HYLO. In *Proceedings of the IFIP TC 2 Working Conference on Algorithmic Languages and Calculi*, pages 76–106. Chapman & Hall, 1997.
17. Alberto Pardo. Fusion of recursive programs with computational effects. *Theoretical Computer Science*, 260(1–2):165–207, June 2001.
18. 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.
19. Tim Sheard and Leonidas Fegaras. A fold for all seasons. In *Proceedings of the 6th Conference on Functional Programming Languages and Computer Architecture*, pages 233–242, 1993.
20. Akihiko Takano and Erik Meijer. Shortcut deforestation in calculational form. In *Proceedings of the 7th ACM Conference on Functional Programming Languages and Computer Architecture (FPCA'95)*, pages 306–313. ACM Press, 1995.
21. Philip Wadler. Deforestation: Transforming programs to eliminate trees. In *Proceedings of the European Symposium on Programming*, number 300 in LNCS, pages 344–358. Springer Verlag, 1988.
22. Philip Wadler. Comprehending monads. *Mathematical Structures in Computer Science*, 2:461–493, 1992.