Combining Datatypes and Effects

Alberto Pardo

Instituto de Computación
Universidad de la República
Montevideo - Uruguay
pardo@fing.edu.uy

Abstract. Recursion schemes over datatypes constitute a powerful tool to structure functional programs. Standard schemes, like map and fold, have traditionally been studied in the context of purely-functional programs. In this paper we propose the generalization of well-known recursion schemes with the aim to obtain structuring mechanisms for programs with effects, assuming that effects are modelled by monads. We analyze the definition as well as the algebraic laws associated with the new recursion schemes. The way monads encapsulate effects plays an important role in the definition of the monadic recursion schemes, as it permits to focus on the structure of the recursive programs with effects disregarding the specific details of the effects involved. We illustrate the use of the recursion schemes and their laws with some traversal algorithms on graphs.

1 Introduction

In functional programming, it is common to find programs written using a compositional design, where a program is constructed as a collection of simple and easy to write functions which communicate through function composition. Programs so defined are modular and have many benefits, such as clarity and maintainability, but unfortunately they are inefficient. Each function composition $f \circ g$ implies passing information from one function to the other through an intermediate data structure which is produced by g and consumed by f. This has associated a computational cost, since the nodes of the intermediate data structure need to be allocated, filled, inspected and finally discarded from memory.

Intermediate data structures can be removed by the application of a program transformation technique known as *deforestation* [30]. Diverse approaches to deforestation can be found in the literature [30, 11, 10, 27, 23]. In this paper we follow an approach based on recursive program schemes over data types [18, 6, 4, 8]. By program schemes we mean higher-order functions that capture common patterns of computation over data types and help in structuring programs. Typical examples are functions like *map* and *fold* [3], but there are many others. Recursion schemes have associated algebraic laws, which are useful for formal reasoning about programs as well as for program transformation purposes. In connection with deforestation, there is a particularly relevant subset of these

laws, the so-called *fusion laws*, which involve the elimination of intermediate data structures.

The purpose of this paper is to study recursion schemes for programs with effects, assuming that effects are modelled by *monads* [2]. Most of the standard recursion schemes can only deal with purely-functional programs (i.e. effect-free programs). This means that they fail when we try to use them to represent programs with effects. Basically, the problem is with the shape of recursion that such programs possess, which is different from that of purely-functional ones. This raises the necessity of generalizing the existing recursion schemes to cope with the patterns of computations of programs with effects.

The compositional style of programming still holds in the context of programs with effects. This means that we will be interested in eliminating intermediate data structures generated by the composition of monadic programs, but now produced as the result of monadic computations. Our strategy will be therefore the derivation of fusion laws associated with the program schemes for programs with effects in order to restore deforestation in the presence of effects.

The paper is built on previous work on recursion schemes for programs with effects [20, 25, 24]. In contrast to [25, 24], where a more abstract style of presentation based on category theory was followed, in this paper concepts and definitions are described in a functional programming style, using a Haskell-like notation.

The paper is organized as follows. In Section 2 we review some standard recursion schemes and their associated fusion laws. Section 3 presents background material on monads. Section 4 is devoted to the analysis of recursion schemes for programs with effects. We also present examples which illustrate the use of the program schemes and their laws. In Section 5, we conclude the paper with a brief description of a program fusion tool which integrates many of the ideas discussed in the paper.

2 Recursive program schemes

The program schemes described in the paper encapsulate common patterns of computation of recursive functions and have a strong connection with datatypes. Before presenting well-known recursion schemes for purely-functional programs, we will show a general construction used to capture datatype declarations. Based on that construction, we will be able to give a generic definition of the recursion schemes, parameterised by the structure of some of the datatypes involved.

Throughout we shall assume we are working in the context of a lazy functional language with a *cpo* semantics, in which types are interpreted as pointed cpos (complete partial orders with a least element $\bot$) and functions are interpreted as continuous functions between pointed cpos. As usual, a function f is said to be *strict* if it preserves the least element, i.e. $f\ \bot = \bot$.

2.1 Data types

The general construction relies on the concept of a *functor*. A functor consists of two components, both denoted by F: a type constructor F, and a function $F :: (a \to b) \to (F\ a \to F\ b)$, which preserves identities and compositions:

$$F\ id = id \qquad\qquad F\ (f \circ g) = F\ f \circ F\ g$$

A standard example of a functor is that formed by the *List* type constructor and the well-known *map* function, which applies a function to the elements of a list, building a new list with the results.

$$\begin{aligned} &map && :: (a \to b) \to (List\ a \to List\ b) \\ &map\ f\ Nil && = Nil \\ &map\ f\ (Cons\ a\ as) && = Cons\ (f\ a)\ (map\ f\ as) \end{aligned}$$

We will use functors to capture the structure (or signature) of datatypes. In this paper we will only consider a restricted class of datatypes, called *regular datatypes*. These are datatypes whose declarations contain no function spaces and have recursive occurrences with the same arguments from left-hand sides. The functors corresponding to regular datatypes' signatures will be characterised by an inductive definition, composed by the following basic functors.

Identity functor. The identity functor is defined as the identity type constructor and the identity function (on functions):

$$\textbf{type}\ I\ a = a$$

$$\begin{aligned} &I\ f :: (a \to b) \to (I\ a \to I\ b) \\ &I\ f = f \end{aligned}$$

Constant functor. For any type t, we can construct a constant functor defined as the constant type constructor and the constant function that maps any function to the identity on t:

$$\textbf{type}\ \underline{t}\ a = t$$

$$\begin{aligned} &\underline{t}\ f :: (a \to b) \to (\underline{t}\ a \to \underline{t}\ b) \\ &\underline{t}\ f = id \end{aligned}$$

Product functor. The product functor is an example of a *bifunctor* (a functor on two arguments). The product type constructor gives the type of pairs as result. The mapping function takes two functions which are applied to each component of the input pair.

$$\begin{aligned} &\textbf{data}\ a \times b && = (a, b) \\ &(\times) && :: (a \to c) \to (b \to d) \to (a \times b \to c \times d) \\ &(f \times g)\ (a, b) && = (f\ a, g\ b) \end{aligned}$$

The elements of a product can be inspected using the projection functions.

$$\pi_1 \qquad :: a \times b \to a$$

$$\pi_1\ (a, b) = a$$

$$\begin{aligned}\pi_2 \quad & :: a \times b \to b\\ \pi_2\ (a, b) &= b\end{aligned}$$

The split operation allows us to construct a product from a given object.

$$\begin{aligned}(\vartriangle) \quad & :: (c \to a) \to (c \to b) \to (c \to a \times b)\\ (f \vartriangle g)\ x &= (f\ x, g\ x)\end{aligned}$$

Among others, the following laws hold:

$$(f \vartriangle g) \circ h = (f \circ h) \vartriangle (g \circ h)$$

$$(f \times g) \circ (h \vartriangle k) = (f \circ h) \vartriangle (g \circ k)$$

Sum functor. The sum functor builds the disjoint sum of two types, which are unions of tagged elements.

$$\mathbf{data}\ a + b = \mathit{Left}\ a \mid \mathit{Right}\ b$$

$$\begin{aligned}(+) \quad & :: (a \to c) \to (b \to d) \to (a + b \to c + d)\\ (f + g)\ (\mathit{Left}\ a) &= \mathit{Left}\ (f\ a)\\ (f + g)\ (\mathit{Right}\ b) &= \mathit{Right}\ (g\ b)\end{aligned}$$

Associated with sums we can define a case analysis operator:

$$\begin{aligned}(\triangledown) \quad & :: (a \to c) \to (b \to c) \to (a + b \to c)\\ (f \triangledown g)\ (\mathit{Left}\ a) &= f\ a\\ (f \triangledown g)\ (\mathit{Right}\ b) &= g\ b\end{aligned}$$

which satisfies the following properties:

$$f \text{ strict} \Rightarrow f \circ (g \triangledown h) = f \circ g \triangledown f \circ h$$

$$(f \triangledown g) \circ (h + k) = f \circ h \triangledown g \circ k$$

Functor composition. The composition of two functors F and G is denoted by $F\ G$. In particular, we can define the composition of the bifunctors $\times$ and $+$ with functors F and G, written $F \times G$ and $F + G$, as follows:

$$\begin{aligned}\mathbf{type}\ (F \times G)\ a &= F\ a \times G\ a\\ (F \times G)\ f &= F\ f \times G\ f\end{aligned}$$

$$\begin{aligned}\mathbf{type}\ (F + G)\ a &= F\ a + G\ a\\ (F + G)\ f &= F\ f + G\ f\end{aligned}$$

Regular functors. Regular functors are functors built from identities, constants, products, sums, compositions and type functors.

$$F ::= I \mid \underline{t} \mid F \times F \mid F + F \mid F\ F \mid D$$

D stands for type functors, which are functors corresponding to polymorphic recursive datatypes (the *List* functor is an example). Their definition is given in Section 2.3.

The general construction. The idea is to describe the top level structure of a datatype by means of a functor. Consider a regular datatype declaration,

$$\textbf{data}\ \tau = C_1\ \tau_{1,1} \cdots \tau_{1,k_1}\ \mid \cdots \mid\ C_n\ \tau_{n,1} \cdots \tau_{n,k_n}$$

The assumption that τ is regular implies that each $\tau_{i,j}$ is restricted to the following forms: some constant type t (like *Int*, *Char*, or even a type variable); a type constructor D (e.g. *List*) applied to a type $\tau'_{i,j}$; or τ itself.

The derivation of a functor from a datatype declaration then proceeds as follows:

- pack the arguments of the constructors in tuples; for constant constructors (i.e. those with no arguments) we place the empty tuple ();
- regard alternatives as sums, replacing | by +; and
- substitute the occurrences of τ by a type variable a in every $\tau_{i,j}$.

As a result, we obtain the following type constructor:

$$F\ a = \sigma_{1,1} \times \cdots \times \sigma_{1,k_1}\ + \cdots +\ \sigma_{n,1} \times \cdots \times \sigma_{n,k_n}$$

where $\sigma_{i,j} = \tau_{i,j}[\tau := a]$[1]. The body of the mapping function $F :: (a \to b) \to (F\ a \to F\ b)$ is similar to that of $F\ a$, with the difference that now we substitute the occurrences of the type variable a by a function $f :: a \to b$, and write identities in the other positions:

$$F f = \overline{\sigma}_{1,1} \times \cdots \times \overline{\sigma}_{1,k_1}\ + \cdots +\ \overline{\sigma}_{n,1} \times \cdots \times \overline{\sigma}_{n,k_n}$$

with

$$\overline{\sigma}_{i,j} = \begin{cases} f & \text{if } \sigma_{i,j} = a \\ id & \text{if } \sigma_{i,j} = t, \text{ for some type } t \\ D\ \overline{\sigma}'_{i,j} & \text{if } \sigma_{i,j} = D\ \sigma'_{i,j} \end{cases}$$

Example 1.

- For the datatype of natural numbers,

  $\textbf{data}\ \mathit{Nat} = \mathit{Zero} \mid \mathit{Succ}\ \mathit{Nat}$

  we can derive a functor N given by

  $\textbf{type}\ N\ a = () + a$

  $$\begin{array}{ll} N & :: (a \to b) \to (N\ a \to N\ b) \\ N\ f & = id + f \end{array}$$

  As a functorial expression, $N = \underline{()} + I$.

[1] By $s[t := a]$ we denote the replacement of every occurrence of t by a in s.

– For a datatype of arithmetic expressions:

  data $Exp = Num\ Int \mid Add\ Exp\ Exp$

  we can derive a functor E given by

  type $E\ a = Int + Exp\ \times\ Exp$

  $$\begin{array}{ll} E & :: (a \to b) \to (E\ a \to E\ b) \\ E\ f & = id + f\ \times\ f \end{array}$$

  As a functorial expression, $E = \underline{Int} + I\ \times\ I$.

– For the datatype of lists,

  $List\ a = Nil \mid Cons\ a\ (List\ a)$

  we can derive a functor L_a given by

  type $L_a\ b = () + a\ \times\ b$

  $$\begin{array}{ll} L_a & :: (b \to c) \to (L_a\ b \to L_a\ c) \\ L_a\ f & = id + id\ \times\ f \end{array}$$

  As a functorial expression, $L_a = \underline{()} + \underline{a}\ \times\ I$. Notice that in this case the functor is parameterised. This happens with the signature of every polymorphic datatype, since it is necessary to reflect in the functor the presence of the type parameter. A parameterised functor F_a is actually the partial application of a bifunctor F: **type** $F_a\ b = F\ a\ b$ and $F_a\ f = F\ id\ f$. □

Every (recursive) regular datatype is then understood as a solution of an equation $Fx \cong x$, being F the functor that captures its signature. A solution to this equation corresponds to a fixed point of the functor F, given by a type t and an isomorphism between $F\ t$ and t. The underlying semantics in terms of cpos ensures the existence of a unique (up to isomorphism) fixed point to every regular functor F whose type is denoted by μF. The isomorphism is provided by the strict functions,

$$F\mu F \underset{out_F}{\overset{in_F}{\rightleftarrows}} \mu F$$

each the inverse of the other, such that in_F (out_F) packs the constructors (destructors) of the datatype. The type μF contains partial, finite as well as infinite values. Further details can be found in [1, 8].

Example 2.

– In the case of the datatype of natural numbers, the corresponding isomorphism is given by the type $\mu N = Nat$ and the functions in_N and out_N:

  $$\begin{array}{ll} in_N & :: N\ Nat \to Nat \\ in_N & = const\ Zero \triangledown Succ \end{array}$$

$$
\begin{aligned}
&out_N && :: Nat \to N\ Nat \\
&out_N\ Zero && = Left\ () \\
&out_N\ (Succ\ n) && = Right\ n
\end{aligned}
$$

$$
\begin{aligned}
&const && :: a \to b \to a \\
&const\ a\ b && = a
\end{aligned}
$$

– In the case of the datatype of lists, the corresponding isomorphism is given by the type $\mu L_a = List\ a$ and the functions in_{L_a} and out_{L_a}:

$$
\begin{aligned}
&in_{L_a} && :: L_a\ (List\ a) \to List\ a \\
&in_{L_a} && = const\ Nil \triangledown uncurry\ Cons
\end{aligned}
$$

$$
\begin{aligned}
&out_{L_a} && :: List\ a \to L_a\ (List\ a) \\
&out_{L_a}\ Nil && = Left\ () \\
&out_{L_a}\ (Cons\ a\ as) && = Right\ (a, as)
\end{aligned}
$$

$$
\begin{aligned}
&uncurry && :: (a \to b \to c) \to (a\ \times\ b \to c) \\
&uncurry\ f\ (a, b) && = f\ a\ b
\end{aligned}
$$

□

2.2 Fold

Fold is a pattern of recursion that captures funcion definitions by structural recursion. The best known example of fold is the definition for lists,

$$
\begin{aligned}
&fold_L && :: (b, a \to b \to b) \to List\ a \to b \\
&fold_L\ (h_1, h_2) && = f_L \\
&\quad \textbf{where} \\
&\quad f_L\ Nil && = h_1 \\
&\quad f_L\ (Cons\ a\ as) && = h_2\ a\ (f_L\ as)
\end{aligned}
$$

which corresponds to the *foldr* operator [3], but the same construction can be generalized to any regular datatype.

The general definition of *fold* can be represented by the following diagram:

$$
\begin{array}{ccc}
\mu F & \xrightarrow{fold\ h} & a \\
{\scriptstyle in_F}\uparrow & & \uparrow{\scriptstyle h} \\
F\mu F & \xrightarrow[F\ (fold\ h)]{} & F\ a
\end{array}
$$

Since out_F is the inverse of the isomorphism in_F, we can write:

$$
\begin{aligned}
&fold && :: (F\ a \to a) \to \mu F \to a \\
&fold\ h && = h \circ F\ (fold\ h) \circ out_F
\end{aligned}
$$

A function $h :: F\ a \to a$ is called an *F-algebra*. The functor F plays the role of signature of the algebra, as it encodes the information about the operations of

the algebra. The type a is called the carrier of the algebra. An *F-homomorphism* between two algebras $h :: F\ a \to a$ and $k :: F\ b \to b$ is a function $f :: a \to b$ between the carriers that commutes with the operations. This is specified by the condition $f \circ h = k \circ F\ f$. Notice that fold is a homomorphism from in_F to h.

Remark 1. When writing the instances of the program schemes, we will adopt the following notational convention for algebras: We will write $(h'_1, \ldots, h'_n)$ instead of $h_1 \triangledown \cdots \triangledown h_n :: F\ a \to a$, such that, $h'_i = v$ when $h_i = const\ v :: () \to a$, or $h'_i :: \tau_1 \to \cdots \to \tau_k \to a$ is the curried version of $h_i :: \tau_1 \times \cdots \times \tau_k \to a$. For example, given an algebra $const\ v \triangledown f :: L_a\ b \to b$ we will write $(e, curry\ f) :: (b, a \to b \to b)$. □

Example 3. The following are instances of fold for different datatypes.

Natural numbers

$$
\begin{aligned}
&fold_N && :: (a, a \to a) \to Nat \to a \\
&fold_N\ (h_1, h_2) &&= f_N \\
&\quad \textbf{where} \\
&\quad\quad f_N\ Zero && = h_1 \\
&\quad\quad f_N\ (Succ\ n) && = h_2\ (f_N\ n)
\end{aligned}
$$

For instance, addition can be defined as:

$$
\begin{aligned}
&add && :: Nat \to Nat \to Nat \\
&add\ m &&= fold_N\ (m, Succ)
\end{aligned}
$$

Leaf-labelled binary trees

$$\textbf{data}\ Btree\ a = Leaf\ a \mid Join\ (Btree\ a)\ (Btree\ a)$$

$$\textbf{type}\ B_a\ b = a + b \times b$$

$$
\begin{aligned}
&B_a && :: (b \to c) \to (B_a\ b \to B_a\ c) \\
&B_a\ f &&= id + f \times f
\end{aligned}
$$

$$
\begin{aligned}
&fold_B && :: (a \to b, b \to b \to b) \to Btree\ a \to b \\
&fold_B\ (h_1, h_2) &&= f_B \\
&\quad \textbf{where} \\
&\quad\quad f_B\ (Leaf\ a) && = h_1\ a \\
&\quad\quad f_B\ (Join\ t\ t') && = h_2\ (f_B\ t)\ (f_B\ t')
\end{aligned}
$$

For instance,

$$
\begin{aligned}
&mirror && :: Btree\ a \to Btree\ a \\
&mirror\ (Leaf\ a) && = Leaf\ a \\
&mirror\ (Join\ t\ t') && = Join\ (mirror\ t')\ (mirror\ t)
\end{aligned}
$$

can be defined as:

$$mirror = fold_B\ (Leaf, \lambda t\ t' \to Join\ t'\ t)$$

□

Fold enjoys some algebraic laws that are useful for program transformation. A law that plays an important role is *fold fusion*. It states that the composition of a fold with a homomorphism is again a fold.

$$f \text{ strict } \;\wedge\; f \circ h = k \circ F\ f \;\;\Rightarrow\;\; f \circ \mathit{fold}\ h = \mathit{fold}\ k$$

The next law is known as *acid rain* or *fold-fold fusion*. The goal of acid rain is to combine functions that produce and consume elements of an intermediate data structure. The intermediate datatype is required to be generated by a fold whose algebra is given in terms of a polymorphic function.

$$\begin{array}{ll} & \tau :: \forall\ a\ .\ (F\ a \to a) \to (G\ a \to a) \\ \Rightarrow & \\ & \mathit{fold}\ h \circ \mathit{fold}\ (\tau\ \mathit{in}_F) = \mathit{fold}\ (\tau\ h) \end{array}$$

Remark 2. We will adopt a samilar notational convention as for algebras for writing functions τ as part of the instances of programs schemes. Concretely, in the instances we will regard a function τ as a function between tuples. That is, if $h_1 \triangledown \cdots \triangledown h_m = \tau(k_1 \triangledown \cdots \triangledown k_n)$, then we will understand this transformation between algebras as $(h'_1, \ldots, h'_m) = \tau(k'_1, \ldots, k'_n)$, where each h'_i and k'_j is obtained from h_i and k_j, respectively, by the convention for algebras. The following example uses this convention. □

Example 4. We use acid-rain to show that

$$\mathit{size}_B \circ \mathit{mirror} = \mathit{size}_B$$

where

$$\begin{array}{l} \mathit{size}_B :: \mathit{Btree}\ a \to \mathit{Int} \\ \mathit{size}_B = \mathit{fold}_B\ (\mathit{const}\ 1, (+)) \end{array}$$

counts the number of leaves of a binary tree. The proof proceeds as follows:

$$\begin{array}{ll} & \mathit{size}_B \circ \mathit{mirror} \\ = & \quad \{\ \text{definition of } \mathit{size}_B \text{ and } \mathit{mirror}\ \} \\ & \mathit{fold}_B\ (\mathit{const}\ 1, (+)) \circ \mathit{fold}_B\ (\mathit{Leaf}, \lambda t\ t' \to \mathit{Join}\ t'\ t) \\ = & \quad \{\ \text{define } \tau\ (h_1, h_2) = (h_1, \lambda x\ x' \to h_2\ x'\ x)\ \} \\ & \mathit{fold}_B\ (\mathit{const}\ 1, (+)) \circ \mathit{fold}_B\ (\tau\ \mathit{in}_{B_a}) \\ = & \quad \{\ \text{acid rain}\ \} \\ & \mathit{fold}_B\ (\tau\ (\mathit{const}\ 1, (+))) \\ = & \quad \{\ \text{definition of } \tau\ \} \\ & \mathit{fold}_B\ (\mathit{const}\ 1, \lambda x\ x' \to x' + x) \\ = & \quad \{\ \text{commutativity of } +, \text{ section } (+)\ \} \\ & \mathit{fold}_B\ (\mathit{const}\ 1, (+)) \\ = & \quad \{\ \text{definition of } \mathit{size}_B\ \} \\ & \mathit{size}_B \end{array}$$

□

2.3 Type functors

Every polymorphic regular datatype gives rise to a polymorphic type constructor $D\,a = \mu F_a$, which can be made into a functor (called a type functor) by defining its mapping function:

$$\begin{aligned} &D \quad :: (a \to b) \to (D\ a \to D\ b) \\ &D\ f = fold\ (in_{F_b} \circ F\ f\ id) \end{aligned}$$

Example 5.

Lists The list type functor corresponds to the standard *map* function [3]:

$$\begin{aligned} &List \quad :: (a \to b) \to (List\ a \to List\ b) \\ &List\ f = fold_L\ (Nil, \lambda a\ bs \to Cons\ (f\ a)\ bs) \end{aligned}$$

that is,

$$\begin{aligned} &List\ f\ Nil \qquad\quad = Nil \\ &List\ f\ (Cons\ a\ as) = Cons\ (f\ a)\ (List\ f\ as) \end{aligned}$$

Leaf-labelled binary trees

$$\begin{aligned} &Btree \quad :: (a \to b) \to (Btree\ a \to Btree\ b) \\ &Btree\ f = fold_B\ (Leaf \circ f, Join) \end{aligned}$$

that is,

$$\begin{aligned} &Btree\ f\ (Leaf\ a) \quad = Leaf\ (f\ a) \\ &Btree\ f\ (Join\ t\ t') = Join\ (Btree\ f\ t)\ (Btree\ f\ t') \end{aligned}$$

□

Example 6. Rose trees are multiway branching structures:

$$\textbf{data}\ Rose\ a = Fork\ a\ (List\ (Rose\ a))$$

The signature of rose trees is captured by a functor R_a given by

$$\textbf{type}\ R_a\ b = a\ \times\ List\ b$$

$$\begin{aligned} &R_a \qquad :: (b \to c) \to (R_a\ b \to R_a\ c) \\ &R_a\ f \qquad = id\ \times\ List\ f \end{aligned}$$

As a functorial expression, $R_a = \underline{a}\ \times\ List$. The fold operator is defined by,

$$\begin{aligned} &fold_R \quad :: (R_a\ b \to b) \to Rose\ a \to b \\ &fold_R\ h = f_R \\ &\quad \textbf{where} \\ &\quad\ f_R\ (Fork\ a\ rs) = h\ a\ (List\ f_R\ rs) \end{aligned}$$

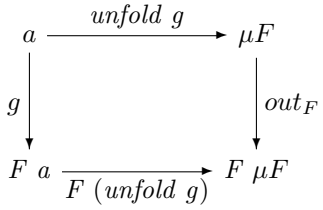
□

A standard property of type functors is *map-fold fusion*. This law states that a map followed by a fold is a fold.

$$fold\ h \circ D\ f = fold\ (h \circ F\ f\ id)$$

2.4 Unfold

Let us now analyze the dual case. The corresponding pattern of recursion, called *unfold* [9, 12], captures function definitions by structural corecursion. By corecursion we understand functions whose structure is dictated by that of the values produced as result. Unfold has a pattern of recursion given by the following scheme:



Proceeding as with fold, since in_F is the inverse of out_F, we can write:

$$\begin{aligned}&unfold \;\; :: (a \to F\ a) \to a \to \mu F\\ &unfold\ g = in_F \circ F\ (unfold\ g) \circ g\end{aligned}$$

Example 7. The following are instances of unfold for different datatypes.

Natural numbers

$$\begin{aligned}&unfold_N \;\; :: (a \to N\ a) \to a \to Nat\\ &unfold_N\ g\ a = \mathbf{case}\ (g\ a)\ \mathbf{of}\\ &\qquad\qquad Left\ () \;\; \to Zero\\ &\qquad\qquad Right\ a' \to Succ\ (unfold_N\ g\ a')\end{aligned}$$

Lists

$$\begin{aligned}&unfold_L \;\; :: (b \to L_a\ b) \to b \to List\ a\\ &unfold_L\ g\ b = \mathbf{case}\ (g\ b)\ \mathbf{of}\\ &\qquad\qquad Left\ () \;\; \to Nil\\ &\qquad\qquad Right\ (a, b') \to Cons\ a\ (unfold_L\ g\ b')\end{aligned}$$

Leaf-labelled binary trees

$$\begin{aligned}&unfold_B \;\; :: (b \to B_a\ b) \to b \to Btree\ a\\ &unfold_B\ g\ b = \mathbf{case}\ (g\ b)\ \mathbf{of}\\ &\qquad\qquad Left\ a \;\; \to Leaf\ a\\ &\qquad\qquad Right\ (b1, b2) \to Join\ (unfold_B\ g\ b1)\ (unfold_B\ g\ b2)\end{aligned}$$

Rose trees

$$\begin{aligned}&unfold_R \;\; :: (b \to R_a\ b) \to b \to Rose\ a\\ &unfold_R\ g\ b = \mathbf{let}\ (a, bs) = g\ b\\ &\qquad\qquad \mathbf{in}\ Fork\ a\ (List\ (unfold_R\ g)\ bs)\end{aligned}$$

□

A function $g :: a \to F\ a$ is called an *F-coalgebra*. A *F-homomorphism* between two coalgebras $g :: a \to F\ a$ and $g' :: b \to F\ b$ is a function $f :: a \to b$ such that $g' \circ f = F\ f \circ g$.

There is a corresponding *fusion* law for unfold, which states that the composition of a homomorphism with an unfold is again an unfold.

$$g' \circ f = F\ f \circ g \quad \Rightarrow \quad unfold\ g' \circ f = unfold\ g$$

There is also an acid rain law, called *unfold-unfold fusion*.

$$\begin{array}{l} \quad \sigma :: \forall\ a\ .\ (a \to F\ a) \to (a \to G\ a) \\ \Rightarrow \\ \quad unfold\ (\sigma\ out_F) \circ unfold\ g = unfold\ (\sigma\ g) \end{array}$$

2.5 Hylomorphism

Now we look at functions given by the composition of a fold with an unfold. They capture the idea of general recursive functions whose structure is dictated by that of a virtual data structure.

Given an algebra $h :: F\ b \to b$ and a coalgebra $g :: a \to F\ a$, a *hylomorphism* [18, 19, 27, 23] is a function $hylo\ h\ g :: a \to b$ defined by

$$hylo\ h\ g\ =\ a \xrightarrow{unfold\ g} \mu F \xrightarrow{fold\ h} b \tag{1}$$

An alternative definition of hylomorphism shows that it is not necessary to construct the intermediate data structure:

$$\begin{array}{l} hylo :: (F\ b \to b) \to (a \to F\ a) \to a \to b \\ hylo\ h\ g = h \circ F\ (hylo\ h\ g) \circ g \end{array}$$

that is,

$$\begin{array}{ccc} a & \xrightarrow{hylo\ h\ g} & b \\ {\scriptstyle g}\downarrow & & \uparrow{\scriptstyle h} \\ F\ a & \xrightarrow[F\ (hylo\ h\ g)]{} & F\ b \end{array}$$

From this definition it is easy to see that fold and unfold are special cases of hylomorphism.

$$fold\ h = hylo\ h\ out_F \qquad\qquad unfold\ g = hylo\ in_F\ g$$

Example 8. We show the definition of hylomorphism for different datatypes.

Lists

$$\begin{array}{l} hylo_L \qquad\qquad :: (c, a \to c \to c) \to (b \to L_a\ b) \to b \to c \\ hylo_L\ (h_1, h_2)\ g\ b = hylo \end{array}$$

$$
\begin{array}{l}
\quad \textbf{where} \\
\quad\quad hylo\ b = \textbf{case}\ (g\ b)\ \textbf{of} \\
\quad\quad\quad\quad\quad\quad Left\ () \quad\quad\ \ \rightarrow h_1 \\
\quad\quad\quad\quad\quad\quad Right\ (a, b') \rightarrow h_2\ a\ (hylo\ b')
\end{array}
$$

For example, the function that computes the factorial of a number

$$
\begin{array}{l}
fact \quad\quad\quad\quad\quad\quad\quad :: Int \rightarrow Int \\
fact\ n \mid n < 1 \quad\quad\ = 1 \\
\quad\quad\ \mid otherwise = n * fact\ (n-1)
\end{array}
$$

can be written as:

$$
\begin{array}{l}
fact = hylo_L\ (1, (*))\ g \\
\quad \textbf{where} \\
\quad\quad g\ n \mid n < 1 \quad\quad\ = Left\ () \\
\quad\quad\quad\ \mid otherwise = Right\ (n, n-1)
\end{array}
$$

The reason of presenting *fact* as a hylomorphism associated with lists is because there is a virtual list that can be seen reflected in the form of the call-tree. Such a list can be made explicit by using (1):

$$
fact = prod \circ upto
$$

$$
\begin{array}{l}
prod :: List\ Int \rightarrow Int \\
prod = fold_L\ (1, (*))
\end{array}
$$

$$
\begin{array}{l}
upto \quad\quad\quad\quad\quad\quad :: Int \rightarrow List\ Int \\
upto\ n \mid n < 1 \quad\quad\ = Nil \\
\quad\quad\ \mid otherwise = Cons\ n\ (upto\ (n-1))
\end{array}
$$

Internally-labelled binary trees

$$
\textbf{data}\ Tree\ a \quad\quad = Empty \mid Node\ (Tree\ a)\ a\ (Tree\ a)
$$

$$
\textbf{type}\ T_a\ b \quad\quad = () + b\ \times\ a\ \times\ b
$$

$$
\begin{array}{l}
T_a \quad\quad\quad\quad\quad\quad :: (b \rightarrow c) \rightarrow (T_a\ b \rightarrow T_a\ c) \\
T_a\ f \quad\quad\quad\quad\quad\quad = id + f\ \times\ id\ \times\ f
\end{array}
$$

$$
\begin{array}{l}
hylo_T \quad\quad\quad\quad\quad\quad :: (c, c \rightarrow a \rightarrow c \rightarrow c) \rightarrow (b \rightarrow T_a\ b) \rightarrow b \rightarrow c \\
hylo_T\ (h_1, h_2)\ g = hylo \\
\quad \textbf{where} \\
\quad\quad hylo\ b = \textbf{case}\ (g\ b)\ \textbf{of} \\
\quad\quad\quad\quad\quad\quad Left\ () \quad\quad\quad\quad\ \rightarrow h_1 \\
\quad\quad\quad\quad\quad\quad Right\ (b1, a, b2) \rightarrow h_2\ (hylo\ b1)\ a\ (hylo\ b2)
\end{array}
$$

For example, the usual definition of quicksort

$$
\begin{array}{l}
qsort \quad\quad\quad\quad\quad\quad :: Ord\ a \Rightarrow List\ a \rightarrow List\ a \\
qsort\ Nil \quad\quad\quad\quad\quad\quad = Nil
\end{array}
$$

$$
\begin{aligned}
qsort\ (Cons\ a\ as) &= qsort\ [x \mid x \leftarrow as; x \leqslant a] \\
&\quad +\!\!+\ wrap\ a\ +\!\!+ \\
&\quad qsort\ [x \mid x \leftarrow as; x > a]
\end{aligned}
$$

$$
\begin{aligned}
wrap &:: a \to List\ a \\
wrap\ a &= Cons\ a\ Nil
\end{aligned}
$$

can be written as a hylomorphism as follows:

$$
\begin{aligned}
&qsort = hylo_T\ (Nil, h)\ g \\
&\quad \textbf{where} \\
&\qquad h\ ys\ a\ zs = ys\ +\!\!+\ wrap\ a\ +\!\!+\ zs \\
&\qquad g\ Nil = Left\ () \\
&\qquad g\ (Cons\ a\ as) = Right\ ([x \mid x \leftarrow as; x \leqslant a], a, [x \mid x \leftarrow as; x > a])
\end{aligned}
$$

□

The following fusion laws are a direct consequence of (1).

Hylo Fusion

$$f \text{ strict } \wedge\ f \circ h = k \circ F\ f \quad \Rightarrow \quad f \circ hylo\ h\ g = hylo\ k\ g$$

$$g' \circ f = F\ f \circ g \quad \Rightarrow \quad hylo\ h\ g' \circ f = hylo\ h\ g$$

Hylo-Fold Fusion

$$
\begin{aligned}
&\tau :: \forall\ a\ .\ (F\ a \to a) \to (G\ a \to a) \\
\Rightarrow\ & \\
&fold\ h \circ hylo\ (\tau\ in_F)\ g = hylo\ (\tau\ h)\ g
\end{aligned}
$$

Unfold-Hylo Fusion

$$
\begin{aligned}
&\sigma :: \forall\ a\ .\ (a \to F\ a) \to (a \to G\ a) \\
\Rightarrow\ & \\
&hylo\ h\ (\sigma\ out_F) \circ unfold\ g = hylo\ h\ (\sigma\ g)
\end{aligned}
$$

3 Monads

It is well-known that computational effects, such as exceptions, side-effects, or input/output, can be uniformly modelled in terms of algebraic structures called monads [21, 2]. In functional programming, monads are a powerful mechanism to structure functional programs that produce effects [31].

A *monad* is usually presented as a *Kleisli triple* $(m, return, \gg\!=)$ composed by a type constructor m, a polymorphic function *return* and a polymorphic operator $(\gg\!=)$ often pronounced *bind*. The natural way to define a monad in Haskell is by means of a class.

$$\begin{aligned}&\textbf{class } \mathit{Monad}\ m \textbf{ where}\\ &\quad \mathit{return} :: a \to m\ a\\ &\quad (\gg\!=)\ \ :: m\ a \to (a \to m\ b) \to m\ b\end{aligned}$$

Computations delivering values of type a are regarded as objects of type $m\ a$, and can be understood as terms with remaining computation steps. The bind operator describes how computations are combined. An expression of the form $m \gg\!= \lambda x \to m'$ is read as follows: evaluate computation m, bind the variable x to the resulting value of this computation, and then continue with the evaluation of computation m'. How the effect is passed around is a matter for each monad. In some cases, we may not be interested in binding the result of the first computation to be used in the second. This can be performed by an operator pronounced *then*,

$$\begin{aligned}&(\gg) \qquad :: \mathit{Monad}\ m \Rightarrow m\ a \to m\ b \to m\ b\\ &m \gg m' = m \gg\!= \lambda\_ \to m'\end{aligned}$$

Formally, to be a monad, the components of the triple must satisfy the following equations:

$$m \gg\!= \mathit{return} = m \tag{2}$$

$$\mathit{return}\ a \gg\!= \lambda x \to m = m\ [x := a] \tag{3}$$

$$(m \gg\!= \lambda x \to m') \gg\!= \lambda y \to m'' = m \gg\!= \lambda x \to (m' \gg\!= \lambda y \to m'') \tag{4}$$

In (4) x cannot appear free in m''. The expression $m\ [x := a]$ means the substitution of all free occurrences of x by a in m.

With the introduction of monads the focus of attention is now on functions of type $a \to m\ b$, often referred to as *monadic functions*, which produce an effect when applied to an argument. Given a monadic function $f :: a \to m\ b$, we define $f^\star :: m\ a \to m\ b$ as $f^\star\ m = m \gg\!= f$. Using the same idea it is possible to define the *Kleisli composition* of two monadic functions,

$$\begin{aligned}&(\bullet) \qquad :: \mathit{Monad}\ m \Rightarrow (b \to m\ c) \to (a \to m\ b) \to (a \to m\ c)\\ &(f \bullet g)\ a = g\ a \gg\!= f\end{aligned}$$

Now we can assign a meaning to the laws of Kleisli triples. The first two laws amount to say that *return* is a left and right identity with respect to Kleisli composition, whereas the last one expresses that composition is associative. Note that $f \bullet g = f^\star \circ g$.

Associated with every monad we can define also a map function, which applies a function to the result yielded by a computation, and a lifting operator, which turns an arbitrary function into a monadic function.

$$\begin{aligned}&\mathit{mmap} \qquad :: \mathit{Monad}\ m \Rightarrow (a \to b) \to (m\ a \to m\ b)\\ &\mathit{mmap}\ f\ m = m \gg\!= \lambda a \to \mathit{return}\ (f\ a)\end{aligned}$$

$$\begin{aligned}&(\widehat{\ }) \qquad :: (a \to b) \to (a \to m\ b)\\ &\widehat{f} \qquad\ \ = \mathit{return} \circ f\end{aligned}$$

Using the Kleisli triple's laws it can be easily verified that both *mmap* and $(\widehat{\ })$ happen to be functorial on functions:

$$\begin{aligned} mmap\ id &= id & \widehat{id} &= return \\ mmap\ (f \circ g) &= mmap\ f \circ mmap\ g & \widehat{f \circ g} &= \widehat{f} \bullet \widehat{g} \end{aligned}$$

Example 9. The *exception monad* models the occurrence of exceptions in a program.

```
data Exc a       = Ok a | Fail Exception
type Exception = String

instance Monad Exc where
  return a       = Ok a
  (Ok a)  >>= f = f a
  (Fail e) >>= f = Fail e
```

This monad captures computations which either succeed returning a value, or fail raising a specific exception signaled by a value of type *Exception*. The *return* operation takes a value and returns a computation that always succeeds, whereas bind may be thought of as a kind of *strict* function application that propagates an exception if one occurs.

When there is a unique exception value, the exception monad is often referred to as the *maybe monad*.

```
data Maybe a = Just a | Nothing

instance Monad Maybe where
  return a       = Just a
  (Just a) >>= f = f a
  Nothing  >>= f = Nothing
```

□

Example 10. State-based computations are modelled by the *state monad*. These are computations that take an initial state and return a value and a possibly modified state.

```
newtype State s a = State (s → (a, s))

instance Monad (State s) where
  return x       = State (λs → (x, s))
  State c >>= f = State (λs → let (a, s')  = c s
                                  State c' = f a
                              in c' s')
```

The bind operator combines two computations in sequence so that the state and value resulting from the first computation are supplied to the second one.

The state monad has been used as an effective tool for encapsulating actual imperative features, such as, mutable variables, destructive data structures, and

input/output, while retaining fundamental properties of the language (see [26, 14, 13]). The idea is to hide the *real* state in an abstract data type (based on the monad) which is equipped with primitive operations that internally access the real state [31, 5, 13]. □

Example 11. The *list monad* enables us to describe computations that produce a list of results, which can be used to model a form of nondeterminism.

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Monad}\ \mathit{List}\ \textbf{where} \\
\quad \begin{array}{ll}
\mathit{return} & = \mathit{wrap} \\
\mathit{Nil} \gg\!= f & = \mathit{Nil} \\
(\mathit{Cons}\ a\ as) \gg\!= f & = f\ a +\!\!+ (as \gg\!= f)
\end{array}
\end{array}
$$

This monad can be seen as a generalization of the maybe monad: a computation of type $\mathit{List}\ a$ may succeed with several outcomes, or fail by returning no result at all. □

With the aim at improving readability of monadic programs, Haskell provides a special syntax called the *do notation*. It is defined by the following translation rules:

$$
\begin{aligned}
\textbf{do}\,\{x \leftarrow m; m'\} &= m \gg\!= \lambda x \rightarrow \textbf{do}\,\{m'\} \\
\textbf{do}\,\{m; m'\} &= m \gg \textbf{do}\,\{m'\} \\
\textbf{do}\,\{m\} &= m
\end{aligned}
$$

4 Recursion with monadic effects

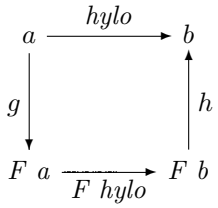
Recursion and monads turn out to be two important structuring devices in functional programming. In this section we combine them with the aim to obtain structuring mechanisms for recursive programs with effects. A natural result of this combination will be the generalization of the existing recursion schemes to work with programs with effects. The way monads encapsulate effects turns out to be essential for this integration, since it permits to focus on the relevant structure of recursive programs disregarding the specific details of the effects they produce.

The fusion laws associated with the monadic program schemes are particularly interesting because they encapsulate new cases of deforestation. However, as we will see later, some of the fusion laws require very strong conditions for their application, reducing dramatically their possibilities to be considered in practice. To overcome this problem we will introduce alternative fusion laws, which, though not so powerful, turn out to be useful in practice.

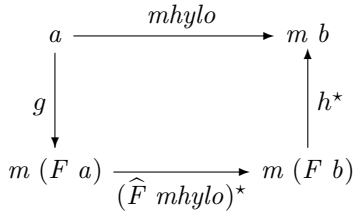
Two alternative approaches can be adopted to the definition of monadic program schemes. One of them, to be presented first, is a strictly structural approach based on a *lifting* construction. This means to translate to the monadic universe the constructions that characterize the recursion schemes, as well as the concepts that take part in them. The other approach, to be presented in Subsection 4.8, is more pragmatical and turns out to be more useful in practice.

4.1 Lifting

Let us start explaining the notion of lifting. Our goal is to define program schemes that capture the recursion structure of functions with effects. Consider the pattern of recursion captured by hylomorphism:



By lifting we mean that we view each arrow of this diagram as an effect-producing function (a somehow *imperative* view). By thinking functionally, we make the effects explicit, giving rise to the following recursion scheme:



(5)

where $h :: F\ b \to m\ b$, $g :: a \to m\ (F\ a)$ and $\widehat{F} :: (a \to m\ b) \to (F\ a \to m\ (F\ b))$, for an arbitrary monad m. These ingredients are monadic versions of the notions of algebra, coalgebra and functor, respectively. Before introducing the monadic versions of fold, unfold and hylomorphism, we analyze first the $\widehat{F}$ construction, since it plays an essential role in the strictly structural approach to the definition of the monadic recursive program schemes.

4.2 Monadic extension of a functor

The *monadic extension* of a functor F [7, 28, 24] is a function

$$\widehat{F} :: (a \to m\ b) \to (F\ a \to m\ (F\ b))$$

whose action embodies that of F. Monadic extensions are used to express the structure of the recursive calls in monadic functions. Every monadic extension $\widehat{F}$ is in one-to-one correspondence with a *distributive law*

$$dist_F :: F\ (m\ a) \to m\ (F\ a)$$

a polymorphic function that performs the distribution of a functor over a monad. In fact, for each distributive law $dist_F$, the action of $\widehat{F}$ on a function $f :: a \to m\ b$ can be defined by

$$\widehat{F}\ f\ =\ F\ a \xrightarrow{F\ f} F\ (m\ b) \xrightarrow{dist_F} m\ (F\ b) \tag{6}$$

Hence, $\widehat{F}\ f$ first applies f to each argument position of type a within a compound value of type $F\ a$, and then joins the monadic effects produced in each function application into a computation that delivers a compound value of type $F\ b$. Conversely, given a monadic extension $\widehat{F}$, the corresponding distributive law is given by $dist_F = \widehat{F}\ id$.

A definition of the distributive law $dist_F :: F\ (m\ a) \to m\ (F\ a)$ for each regular functor F can be given by induction on the structure of F:

$$\begin{aligned} dist_I &= id & dist_{F\times G} &= dist_\times \circ (dist_F \times dist_G) \\ dist_{\underline{t}} &= return & dist_{F+G} &= dist_+ \circ (dist_F + dist_G) \\ dist_{FG} &= dist_F \circ F\ dist_G & dist_D &= fold\ (mmap\ in_{Fa} \circ dist_F) \end{aligned}$$

where

$$\begin{aligned} &dist_+ :: m\ a + m\ b \to m\ (a + b) \\ &dist_+ = \lambda s \to \mathbf{case}\ s\ \mathbf{of} \\ &\qquad\qquad Left\ ma \ \to \mathbf{do}\ a \leftarrow ma \\ &\qquad\qquad\qquad\qquad\quad return\ (Left\ a) \\ &\qquad\qquad Right\ mb \to \mathbf{do}\ b \leftarrow mb \\ &\qquad\qquad\qquad\qquad\quad return\ (Right\ b) \end{aligned}$$

In the case of $dist_D$, with $D\ a = \mu F_a$, $dist_F :: F\ (m\ a)\ (m\ b) \to m\ (F\ a\ b)$ represents a distributive law for the bifunctor F. Distributive laws for regular bifunctors can be defined analogously by induction.

The inductive definition of $dist_F$ given above is parametric in the distributive law for the product functor

$$dist_\times :: (m\ a, m\ b) \to m\ (a, b)$$

Here we have two equally valid alternatives to choose. One is to define $dist_\times$ as a left-to-right product distribution,

$$dist_\times\ (m, m') = \mathbf{do}\ \{\, a \leftarrow m; b \leftarrow m'; return\ (a, b)\,\}$$

combining a pair of computations by first evaluating the first one and then the second. The other alternative is to evaluate the computations from right-to-left,

$$dist_\times\ (m, m') = \mathbf{do}\ \{\, b \leftarrow m'; a \leftarrow m; return\ (a, b)\,\}$$

A monad is said to be *commutative* if both alternatives produce the same result on the same input. Monads like identity or state reader [31] are commutative. Examples of noncommutative monads are state and list.

Example 12. Assuming that $dist_\times$ proceeds from left-to-right, the following are examples of distributive laws:

$$\begin{aligned} &dist_N\ :: Monad\ m \Rightarrow N\ (m\ a) \to m\ (N\ a) \\ &dist_N\ = \lambda x \to \mathbf{case}\ x\ \mathbf{of} \\ &\qquad\qquad Left\ () \quad \to return\ (Left\ ()) \\ &\qquad\qquad Right\ ma \to \mathbf{do}\ a \leftarrow ma \\ &\qquad\qquad\qquad\qquad\quad return\ (Right\ a) \end{aligned}$$

$$
\begin{aligned}
&dist_{L_a} :: Monad\ m \Rightarrow L\ a\ (m\ b) \to m\ (L\ a\ b)\\
&dist_{L_a} = \lambda x \to \mathbf{case}\ x\ \mathbf{of}\\
&\qquad\qquad Left\ () \qquad\quad \to return\ (Left\ ())\\
&\qquad\qquad Right\ (a, mb) \to \mathbf{do}\ b \leftarrow mb\\
&\qquad\qquad\qquad\qquad\qquad return\ (Right\ (a, b))
\end{aligned}
$$

$$
\begin{aligned}
&dist_{B_a} :: Monad\ m \Rightarrow B\ a\ (m\ b) \to m\ (B\ a\ b)\\
&dist_{B_a} = \lambda x \to \mathbf{case}\ x\ \mathbf{of}\\
&\qquad\qquad Left\ a \qquad\quad\ \to return\ (Left\ a)\\
&\qquad\qquad Right\ (mb, mb') \to \mathbf{do}\ b\ \leftarrow mb\\
&\qquad\qquad\qquad\qquad\qquad\quad b' \leftarrow mb'\\
&\qquad\qquad\qquad\qquad\qquad\quad return\ (Right\ (b, b'))
\end{aligned}
$$

$$
\begin{aligned}
&dist_{R_a} :: Monad\ m \Rightarrow R\ a\ (m\ b) \to m\ (R\ a\ b)\\
&dist_{R_a} = \lambda(a, mbs) \to \mathbf{do}\ bs \leftarrow sequence\ mbs\\
&\qquad\qquad\qquad\qquad\quad return\ (a, bs)
\end{aligned}
$$

where *sequence* is a distributive law corresponding to the list type functor:

$$
\begin{aligned}
&sequence &&:: Monad\ m \Rightarrow List\ (m\ a) \to m\ (List\ a)\\
&sequence\ Nil &&= return\ Nil\\
&sequence\ (Cons\ m\ ms) &&= \mathbf{do}\ a\ \leftarrow m\\
&&&\qquad as \leftarrow sequence\ ms\\
&&&\qquad return\ (Cons\ a\ as)
\end{aligned}
$$

□

An inductive definition of $\widehat{F}$ can be derived from the definition of $dist_F$:

$$
\begin{aligned}
&\widehat{I}\ f = f && \widehat{(F + G)}\ f = dist_+ \circ (\widehat{F}\ f + \widehat{G}\ f)\\
&\widehat{\underline{t}}\ f = return && \widehat{(F \times G)}\ f = dist_\times \circ (\widehat{F}\ f\ \times\ \widehat{G}\ f)\\
&\widehat{FG}\ f = \widehat{F}\ \widehat{G}\ f && \widehat{D}\ f = fold\ (mmap\ in_{Fa} \circ \widehat{F}\ (f, id))
\end{aligned}
$$

In the case of $\widehat{D}$, $\widehat{F}$ is the monadic extension of the bifunctor F, where $\mu F_a = Da$.

Example 13. Assuming that $dist_\times$ proceeds from left to right, the following are examples of monadic extensions:

$$
\begin{aligned}
&\widehat{N} &&:: Monad\ m \Rightarrow (a \to m\ b) \to (N\ a \to m\ (N\ b))\\
&\widehat{N}\ f &&= \lambda x \to \mathbf{case}\ x\ \mathbf{of}\\
&&&\qquad Left\ () \to return\ (Left\ ())\\
&&&\qquad Right\ a \to \mathbf{do}\ b \leftarrow f\ a\\
&&&\qquad\qquad\qquad return\ (Right\ b)
\end{aligned}
$$

$$
\begin{aligned}
&\widehat{L_a} &&:: Monad\ m \Rightarrow (b \to m\ c) \to (L\ a\ b \to m\ (L\ a\ c))\\
&\widehat{L_a}\ f &&= \lambda x \to \mathbf{case}\ x\ \mathbf{of}\\
&&&\qquad Left\ () \to return\ (Left\ ())
\end{aligned}
$$

$$
\begin{array}{l}
\quad\quad\quad\quad Right\ (a,b) \rightarrow \textbf{do}\ c \leftarrow f\ b \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad return\ (Right\ (a,c))
\end{array}
$$

$$
\begin{array}{l}
\widehat{B_a} \quad :: Monad\ m \Rightarrow (b \rightarrow m\ c) \rightarrow (B\ a\ b \rightarrow m\ (B\ a\ c)) \\
\widehat{B_a}\ f = \lambda x \rightarrow \textbf{case}\ x\ \textbf{of} \\
\quad\quad\quad Left\ a \rightarrow return\ (Left\ a) \\
\quad\quad\quad Right\ (b,b') \rightarrow \textbf{do}\ c \leftarrow f\ b \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad c' \leftarrow f\ b' \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad return\ (Right\ (c,c'))
\end{array}
$$

$$
\begin{array}{l}
\widehat{R_a} \quad :: Monad\ m \Rightarrow (b \rightarrow m\ c) \rightarrow (R\ a\ b \rightarrow m\ (R\ a\ c)) \\
\widehat{R_a}\ f = \lambda(a,bs) \rightarrow \textbf{do}\ cs \leftarrow mapM\ f\ bs \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad return\ (a,cs)
\end{array}
$$

where $mapM$ is a monadic extension $\widehat{List}$ of the list type functor:

$$
\begin{array}{ll}
mapM & :: Monad\ m \Rightarrow (a \rightarrow m\ b) \rightarrow (List\ a \rightarrow m\ (List\ b)) \\
mapM\ f\ Nil & = return\ Nil \\
mapM\ f\ (Cons\ a\ as) & = \textbf{do}\ b \leftarrow f\ a \\
& \quad\quad\quad bs \leftarrow mapM\ f\ as \\
& \quad\quad\quad return\ (Cons\ b\ bs)
\end{array}
$$

□

A monadic extension is said to be a *lifting* whenever it behaves like a functor with respect to monadic functions. That is, when it preserves identities (returns) and Kleisli composition.

$$\widehat{F}\ return = return \qquad\qquad \widehat{F}\ (f \bullet g) = \widehat{F}\ f \bullet \widehat{F}\ g$$

As established by Mulry [22], a monadic extension is a lifting iff its associated distributive law satisfies the following conditions:

$$dist_F \circ F\ return = return \tag{7}$$

$$dist_F \circ F\ join = dist_F \bullet dist_F \tag{8}$$

where

$$
\begin{array}{l}
join \quad :: Monad\ m \Rightarrow m\ (m\ a) \rightarrow m\ a \\
join\ m = \textbf{do}\ \{ m' \leftarrow m; m' \}
\end{array}
$$

Equation (7) ensures the preservation of identities, while (8) makes $\widehat{F}$ distribute over Kleisli composition.

An interesting case to analyze is that of the product functor.[2] It is easy to verify that (7) is valid for every monad:

$$dist_\times \circ (return \times return) = return$$

[2] A detailed analysis for all regular functors can be found in [25, 24].

For example, assuming that $dist_\times$ proceeds from left to right, we have that:

$$
\begin{array}{ll}
 & (dist_\times \circ (return \ \times \ return))\ (a, b) \\
= & \quad \{ \ \text{definition } dist_\times \ \} \\
 & \mathbf{do}\ \{ x \leftarrow return\ a; y \leftarrow return\ b; return\ (x, y) \} \\
= & \quad \{ \ (3) \ \} \\
 & \mathbf{do}\ \{ y \leftarrow return\ b; return\ (a, y) \} \\
= & \quad \{ \ (3) \ \} \\
 & return\ (a, b)
\end{array}
$$

The same holds if $dist_\times$ is right to left. However, equation (8),

$$
\begin{array}{ccc}
(m^2\ a, m^2\ b) & \xrightarrow{join \ \times \ join} & (m\ a, m\ b) \\
\Big\downarrow{\scriptstyle dist_\times} & & \Big\downarrow{\scriptstyle dist_\times} \\
m\ (m\ a, m\ b) & \xrightarrow[dist^\star_\times]{} & m\ (a, b)
\end{array}
$$

does not always hold, since it requires the monad to be commutative. To see the problem, let us calculate the expressions corresponding to each side of the equation. Again, assume that $dist_\times$ is left to right. We start with the left-hand side:

$$
\begin{array}{ll}
 & (dist_\times \circ (join \ \times \ join))\ (m2, m2') \\
= & \quad \{ \ \text{definition of } dist_\times \ \} \\
 & \mathbf{do}\ \{ a \leftarrow join\ m2; b \leftarrow join\ m2'; return\ (a, b) \} \\
= & \quad \{ \ \text{definition of } join \ \} \\
 & \mathbf{do}\ \{ a \leftarrow \mathbf{do}\ \{ m \leftarrow m2; m \}; b \leftarrow \mathbf{do}\ \{ m' \leftarrow m2'; m' \}; return\ (a, b) \} \\
= & \quad \{ \ (4) \ \} \\
 & \mathbf{do}\ \{ m \leftarrow m2; a \leftarrow m; m' \leftarrow m2'; b \leftarrow m'; return\ (a, b) \}
\end{array}
$$

Now, the right-hand side:

$$
\begin{array}{ll}
 & (dist_\times \ \bullet \ dist_\times)\ (m2, m2') \\
= & \quad \{ \ \text{definition of } dist_\times \text{ and Kleisli composition} \ \} \\
 & \mathbf{do}\ \{ (n, n') \leftarrow \mathbf{do}\ \{ m \leftarrow m2; m' \leftarrow m2'; return\ (m, m') \}; \\
 & \qquad a \leftarrow n; b \leftarrow n'; return\ (a, b) \} \\
= & \quad \{ \ (3) \text{ and } (4) \ \} \\
 & \mathbf{do}\ \{ m \leftarrow m2; m' \leftarrow m2'; a \leftarrow m; b \leftarrow m'; return\ (a, b) \}
\end{array}
$$

Both expressions involve exactly the same computations, but they are executed in different order. If we were working with the state monad, for example, the order in which computations are performed is completely relevant both for the side-effects produced and for the values delivered by the computations.

The failure of (8) for functors containing products makes it necessary to add the hypothesis of preservation of Kleisli composition in those fusion laws in which that condition is required. There are some functors involving products for which (8) holds. These are functors containing product expressions of the form $F = \underline{t} \times I$ (or symmetric). For example, for that F, the distributive law $dist_F :: (t, m\ a) \to m\ (t, a)$, given by,

$$
\begin{array}{ll}
 & dist_F\ (t, m) \\
= & \quad \{\ \text{inductive definition}\ \} \\
 & (dist_\times \circ (return\ \times\ id))\ (t, m) \\
= & \quad \{\ dist_\times \text{ left-to-right}\ \} \\
 & \mathbf{do}\ \{x \leftarrow return\ t; a \leftarrow m; return\ (x, a)\} \\
= & \quad \{\ (3)\ \} \\
 & \mathbf{do}\ \{a \leftarrow m; return\ (t, a)\}
\end{array}
$$

satisfies (8) for every monad, as can be verified:

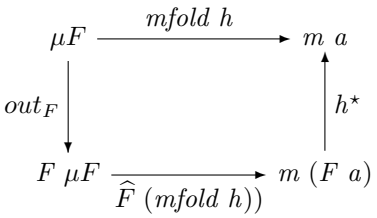
$$
\begin{array}{ll}
 & (dist_F \circ (id\ \times\ join))\ (t, m2) \\
= & \quad \{\ \text{definition of } dist_F\ \} \\
 & \mathbf{do}\ \{a \leftarrow join\ m2; return\ (t, a)\} \\
= & \quad \{\ \text{definition of } join\ \} \\
 & \mathbf{do}\ \{a \leftarrow \mathbf{do}\ \{m \leftarrow m2; m\}; return\ (t, a)\} \\
= & \quad \{\ (4)\ \} \\
 & \mathbf{do}\ \{m \leftarrow m2; a \leftarrow m; return\ (t, a)\} \\
= & \quad \{\ (3)\ \} \\
 & \mathbf{do}\ \{(x, n) \leftarrow \mathbf{do}\ \{m \leftarrow m2; return\ (t, m)\}; a \leftarrow n; return\ (x, a)\} \\
= & \quad \{\ \text{definition of } dist_F \text{ and Kleisli composition}\ \} \\
 & (dist_F\ \bullet\ dist_F)\ (t, m2)
\end{array}
$$

4.3 Monadic Fold

Monadic fold [7] is a pattern of recursion that captures structural recursive functions with monadic effects. A definition of monadic fold is obtained by instantiating (5) with $g = \widehat{out_F}$:

$$
\begin{array}{ccc}
\mu F & \xrightarrow{\quad mfold\ h\quad} & m\ a \\
\Big\downarrow{\scriptstyle \widehat{out_F}} & & \Big\uparrow{\scriptstyle h^\star} \\
m\ (F\ \mu F) & \xrightarrow[(\widehat{F}\ (mfold\ h))^\star]{} & m\ (F\ a)
\end{array}
$$

By (3) this can be simplified to:



Therefore,

$$\begin{aligned}&mfold \quad :: Monad\ m \Rightarrow (F\ a \to m\ a) \to \mu F \to m\ a\\ &mfold\ h = h \bullet \widehat{F}\ (mfold\ h) \circ out_F\end{aligned}$$

Example 14. The following are instances of monadic fold for different datatypes. We assume a left-to-right product distribution $dist_\times$.

Lists

$$\begin{aligned}&mfold_L \qquad\qquad :: Monad\ m \Rightarrow (m\ b, a \to b \to m\ b) \to List\ a \to m\ b\\ &mfold_L\ (h_1, h_2) = mf_L\\ &\quad \textbf{where}\\ &\qquad mf_L\ Nil \qquad\quad = h_1\\ &\qquad mf_L\ (Cons\ a\ as) = \textbf{do}\ y \leftarrow mf_L\ as\\ &\qquad\qquad\qquad\qquad\quad\ \ h_2\ a\ y\end{aligned}$$

For instance, the function that sums the numbers produced by a list of computations (performed from right to left),

$$\begin{aligned}&msum_L \qquad\qquad :: Monad\ m \Rightarrow List\ (m\ Int) \to m\ Int\\ &msum_L\ Nil \qquad\quad = return\ 0\\ &msum_L\ (Cons\ m\ ms) = \textbf{do}\ \{y \leftarrow msum_L\ ms; x \leftarrow m; return\ (x + y)\}\end{aligned}$$

can be defined as:

$$msum_L = mfold_L\ (return\ 0, \lambda m\ y \to \textbf{do}\ \{x \leftarrow m; return\ (x + y)\})$$

Leaf-labelled binary trees

$$\begin{aligned}&mfold_B \qquad\qquad :: Monad\ m \Rightarrow (a \to m\ b, b \to b \to m\ b) \to Btree\ a \to m\ b\\ &mfold_B\ (h_1, h_2) = mf_B\\ &\quad \textbf{where}\\ &\qquad mf_B\ (Leaf\ a) \quad = h_1\ a\\ &\qquad mf_B\ (Join\ t\ t') = \textbf{do}\ y\ \leftarrow mf_B\ t\\ &\qquad\qquad\qquad\qquad\quad\ \ y' \leftarrow mf_B\ t'\\ &\qquad\qquad\qquad\qquad\quad\ \ h_2\ y\ y'\end{aligned}$$

For instance, the function that sums the numbers produced by a tree of computations (performed from left to right),

$$msum_B \qquad\qquad :: Monad\ m \Rightarrow Btree\ (m\ Int) \to m\ Int$$

$$
\begin{aligned}
&msum_B\ (Leaf\ m) &&= m\\
&msum_B\ (Join\ t\ t') &&= \mathbf{do}\ \{y \leftarrow msum_B\ t; y' \leftarrow msum_B\ t'; return\ (y + y')\}
\end{aligned}
$$

can be defined as:

$$msum_B = mfold_B\ (id, \lambda y\ y' \rightarrow return\ (y + y'))$$

Rose trees

$$
\begin{aligned}
&mfold_R &&:: Monad\ m \Rightarrow (a \rightarrow List\ b \rightarrow m\ b) \rightarrow Rose\ a \rightarrow m\ b\\
&mfold_R\ h &&= mf_R\\
&\quad \mathbf{where}\\
&\qquad mf_R\ (Fork\ a\ rs) &&= \mathbf{do}\ ys \leftarrow mapM\ mf_R\ rs\\
&\qquad\qquad &&\quad\ \ h\ a\ ys
\end{aligned}
$$

In this case, the function that sums the numbers produced by a tree of computations,

$$
\begin{aligned}
&msum_R &&:: Monad\ m \Rightarrow Rose\ (m\ Int) \rightarrow m\ Int\\
&msum_R\ (Fork\ m\ rs) &&= \mathbf{do}\ ys \leftarrow mapM\ msum_R\ rs\\
&\qquad\qquad &&\quad\ \ x \leftarrow m\\
&\qquad\qquad &&\quad\ \ return\ (x + sum_L\ ys)
\end{aligned}
$$

$$
\begin{aligned}
&sum_L &&:: List\ Int \rightarrow Int\\
&sum_L &&= fold_L\ (0, (+))
\end{aligned}
$$

can be defined as:

$$msum_R = mfold_R\ (\lambda m\ ys \rightarrow \mathbf{do}\ \{x \leftarrow m; return\ (x + sum_L\ ys)\})$$

□

Functions of type $F\ a \rightarrow m\ a$ are called *monadic F-algebras*; the type a is called the carrier of the algebra. Like purely-functional algebras, monadic algebras may be thought of as structures. The difference is that they return a computation instead of simply a value. As could be seen in Example 14, we adopt a similar notational convention as for algebras to write monadic algebras in instances of the schemes.

A structure-preserving mapping between two monadic algebras is a function between their carriers that preserves their structures, and is *compatible* with their monadic effects. We identify two forms of structure-preserving mappings.

A *F-homomorphism* between two monadic algebras $h :: F\ a \rightarrow m\ a$ and $k :: F\ b \rightarrow m\ b$ is a monadic function $f :: a \rightarrow m\ b$ such that $f \bullet h = k \bullet \widehat{F}\ f$. The use of $\widehat{F}$ in the definition of homomorphism is essential, since it is necessary to join the effects produced by the occurrences of f within the expression $F\ f$. Homomorphisms are closed under composition provided $\widehat{F}$ preserves Kleisli compositions.

A weaker notion of mapping between two monadic algebras $h :: F\ a \rightarrow m\ a$ and $k :: F\ b \rightarrow m\ b$ is what we call a *pure homomorphism*: a function $f :: a \rightarrow b$ such that $mmap\ f \circ h = k \circ F\ f$. A pure homomorphism may be thought of as a

means of changing the 'representation' of a monadic algebra while maintaining the effects that it produces.

The following are fusion laws for monadic fold. In all of them it is necessary to assume that function $mmap :: (a \to b) \to (m\ a \to m\ b)$ is strictness-preserving, in the sense that it maps strict functions to strict functions.

MFold Fusion If $\widehat{F}$ preserves Kleisli compositions,

$$f \text{ strict } \wedge\ f \bullet h = k \bullet \widehat{F}\, f \quad \Rightarrow \quad f \bullet mfold\ h = mfold\ k$$

MFold Pure Fusion

$$f \text{ strict } \wedge\ mmap\ f \circ h = k \circ F\ f \quad \Rightarrow \quad mmap\ f \circ mfold\ h = mfold\ k$$

MFold-Fold Fusion

$$\begin{array}{ll} & \tau :: \forall\ a\ .\ (F\ a \to a) \to (G\ a \to m\ a) \\ \Rightarrow & \\ & mmap\ (fold\ h) \circ mfold\ (\tau\ in_F) = mfold\ (\tau\ h) \end{array}$$

We will adopt a similar notational convention as for the case of algebras to write this kind of functions τ in instances of the program schemes.

Example 15. In Example 14, we showed that $msum_L$ can be defined as a monadic fold. Assuming that *mmap* is strictness-preserving, we use fusion to show that:

$$msum_L = mmap\ sum_L \circ lsequence$$

being *lsequence* the function that performs a list of computations from right to left:

$$\begin{array}{lll} lsequence & :: Monad\ m \Rightarrow List\ (m\ a) \to m\ (List\ a) \\ lsequence\ Nil & = return\ Nil \\ lsequence\ (Cons\ m\ ms) & = \mathbf{do}\ as \leftarrow lsequence\ ms \\ & \qquad\quad a\ \leftarrow m \\ & \qquad\quad return\ (Cons\ a\ as) \end{array}$$

We can express *lsequence* as a monadic fold,

$$\begin{array}{l} lsequence = mfold_L\ (return\ Nil, \\ \qquad\qquad \lambda m\ as \to \mathbf{do}\ \{\, a \leftarrow m; return\ (Cons\ a\ as)\,\}) \end{array}$$

such that it is possible to write its monadic algebra as $\tau\ (Nil, Cons)$, where

$$\begin{array}{lll} \tau & :: (b, a \to b \to b) \to (b, m\ a \to b \to m\ b) \\ \tau\ (h_1, h_2) & = (return\ h_1, \\ & \qquad \lambda m\ b \to \mathbf{do}\ \{\, a \leftarrow m; return\ (h_2\ a\ b)\,\}) \end{array}$$

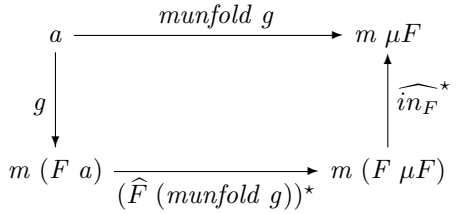
Finally, we calculate

$$
\begin{aligned}
& mmap\ sum_L \circ lsequence \\
= & \quad \{\ \text{definition of } sum_L \text{ and } lsequence\ \} \\
& mmap\ (fold_L\ (0,(+))) \circ mfold_L\ (\tau\ (Nil, Cons)) \\
= & \quad \{\ \text{mfold-fold fusion}\ \} \\
& mfold_L\ (\tau\ (0,(+))) \\
= & \quad \{\ \text{definition of } \tau \text{ and } msum_L\ \} \\
& msum_L
\end{aligned}
$$

□

4.4 Monadic Unfold

Now we turn to the analysis of corecursive functions with monadic effects. Like monadic fold, the definition of monadic unfold can be obtained from (5), now taking $h = \widehat{in_F}$.



that is,

$$
\begin{aligned}
& munfold \quad :: Monad\ m \Rightarrow (a \to m\ (F\ a)) \to (a \to m\ \mu F) \\
& munfold\ g\ a = (return \circ in_F) \bullet \widehat{F}\ (unfold\ g) \bullet g
\end{aligned}
$$

Example 16. We show the definition of monadic unfold for different datatypes. Again, we assume a left to right product distribution $dist_\times$.

Lists

$$
\begin{aligned}
& munfold_L \quad :: Monad\ m \Rightarrow (b \to m\ (L\ a\ b)) \to (b \to m\ (List\ a)) \\
& munfold_L\ g\ b = \mathbf{do}\ x \leftarrow g\ b \\
& \qquad\qquad \mathbf{case}\ x\ \mathbf{of} \\
& \qquad\qquad\quad Left\ () \qquad\ \to return\ Nil \\
& \qquad\qquad\quad Right\ (a, b') \to \mathbf{do}\ as \leftarrow munfold_L\ g\ b' \\
& \qquad\qquad\qquad\qquad\qquad\quad return\ (Cons\ a\ as)
\end{aligned}
$$

Leaf-labelled binary trees

$$
\begin{aligned}
& munfold_B \quad :: Monad\ m \Rightarrow (b \to m\ (B\ a\ b)) \to (b \to m\ (Btree\ a)) \\
& munfold_B\ g\ b = \mathbf{do}\ x \leftarrow g\ b \\
& \qquad\qquad \mathbf{case}\ x\ \mathbf{of} \\
& \qquad\qquad\quad Left\ a \qquad\quad \to return\ (Leaf\ a) \\
& \qquad\qquad\quad Right\ (b1, b2) \to \mathbf{do}\ t1 \leftarrow munfold_B\ g\ b1
\end{aligned}
$$

$t2 \leftarrow munfold_B\ g\ b2$
$return\ (Join\ t1\ t2)$

Rose trees

$$
\begin{aligned}
&munfold_R \quad :: Monad\ m \Rightarrow (b \to m\ (R\ a\ b)) \to (b \to m\ (Rose\ a)) \\
&munfold_R\ g\ b = \mathbf{do}\ (a, bs) \leftarrow g\ b \\
&\qquad\qquad\qquad\quad rs \quad\ \leftarrow mapM\ (munfold_R\ g)\ bs \\
&\qquad\qquad\qquad\quad return\ (Fork\ a\ rs)
\end{aligned}
$$

□

A function $g :: a \to m\ (F\ a)$ is called a *monadic F-coalgebra*. Structure-preserving mappings between monadic coalgebras play an important role in the fusion laws for monadic unfold. A *F-homomorphism* between two monadic coalgebras $g :: a \to m\ (F\ a)$ and $g' :: b \to m\ (F\ b)$ is a function $f :: a \to m\ b$ such that $g' \bullet f = \widehat{F}\ f \bullet g$. Homomorphisms between monadic coalgebras are closed under composition provided $\widehat{F}$ preserves Kleisli compositions.

Like with monadic algebras, we can define a weaker notion of structure-preserving mapping. A *pure homomorphism* between two coalgebras $g :: a \to m\ (F\ a)$ and $g' :: b \to m\ (F\ b)$ is a function $f :: a \to b$ between their carriers such that $g' \circ f = mmap\ (F\ f) \circ g$. Again, a pure homomorphism may be regarded as a representation changer.

The following are fusion laws for monadic unfold.

MUnfold Fusion If $\widehat{F}$ preserves Kleisli compositions,

$$g' \bullet f = \widehat{F}\ f \bullet g \quad \Rightarrow \quad munfold\ g' \bullet f = munfold\ g$$

MUnfold Pure Fusion

$$g' \circ f = mmap\ (F\ f) \circ g \quad \Rightarrow \quad munfold\ g' \circ f = munfold\ g$$

Unfold-MUnfold Fusion

$$
\begin{aligned}
&\sigma :: \forall\ a\ .\ (a \to F\ a) \to (a \to m\ (G\ a)) \\
\Rightarrow\ & \\
&munfold\ (\sigma\ out_F) \circ unfold\ g = munfold\ (\sigma\ g)
\end{aligned}
$$

4.5 Graph traversals

A *graph traversal* is a function that takes a list of roots (entry points to a graph) and returns a list containing the vertices met along the way. In this subsection we show that classical graph traversals, such as DFS or BFS, can be formulated as a monadic unfold.

We assume a representation of graphs that provides a function *adj* which returns the *adjacency list* for each vertex.

type *Graph v* = ...

$$adj :: Eq\ v \Rightarrow Graph\ v \to v \to List\ v$$

In a graph traversal vertices are visited at most once. Hence, it is necessary to maintain a set where to keep track of vertices already visited in order to avoid repeats. Let us assume an abstract data type of finite sets over a, with operations

$$\begin{aligned}
&emptyS :: Set\ a \\
&insS \quad :: Eq\ a \Rightarrow a \to Set\ a \to Set\ a \\
&memS \ :: Eq\ a \Rightarrow a \to Set\ a \to Bool
\end{aligned}$$

where *emptyS* denotes the empty set, *insS* is set insertion and *memS* is a membership predicate.

We handle the set of visited nodes in a state monad. A standard technique to do so is to encapsulate the set operations in an abstract data type based on the monad [31]:

$$\begin{aligned}
&\textbf{type}\ M\ a\ b \qquad = State\ (Set\ a)\ b \\
&\\
&runMS \qquad\quad :: M\ a\ b \to b \\
&runMS\ (State\ f) = \pi_1\ (f\ emptyS) \\
&\\
&insMS \qquad\quad :: Eq\ a \Rightarrow a \to M\ a\ () \\
&insMS\ a \qquad\quad = State\ (\lambda s \to ((), insS\ a\ s)) \\
&\\
&memMS \qquad\quad :: Eq\ a \Rightarrow a \to M\ a\ Bool \\
&memMS\ a \qquad\quad = State\ (\lambda s \to (memS\ a\ s, s))
\end{aligned}$$

Such a technique makes it possible to consider, if desired, an imperative representation of sets, like e.g. a *characteristic vector* of boolean values, which allows $O(1)$ time insertions and lookups when implemented by a mutable array. In that case the monadic abstract data type has to be implemented in terms of the ST monad [14].

Now, we define graph traversal:

$$\begin{aligned}
&\textbf{type}\ Policy\ v = Graph\ v \to v \to List\ v \to List\ v \\
&\\
&graphtrav \qquad :: Eq\ v \Rightarrow Policy\ v \to Graph\ v \to List\ v \to List\ v \\
&graphtrav\ pol\ g = runMS \circ gtrav\ pol\ g \\
&\\
&gtrav \qquad\quad :: Eq\ v \Rightarrow Policy\ v \to Graph\ v \to List\ v \to M\ v\ (List\ v) \\
&gtrav\ pol\ g\ vs \ = \textbf{do}\ xs \leftarrow mdropS\ vs \\
&\qquad\qquad\qquad\quad \textbf{case}\ xs\ \textbf{of} \\
&\qquad\qquad\qquad\qquad Nil \qquad\quad \to return\ Nil \\
&\qquad\qquad\qquad\qquad Cons\ v\ vs \to \textbf{do}\ insMS\ v \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad zs \leftarrow gtrav\ pol\ g\ (pol\ g\ v\ vs) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad return\ (Cons\ v\ zs) \\
&\\
&mdropS \qquad\quad :: Eq\ v \Rightarrow List\ v \to M\ v\ (List\ v) \\
&mdropS\ Nil \qquad\quad = return\ Nil \\
&mdropS\ (Cons\ v\ vs) = \textbf{do}\ b \leftarrow memMS\ a
\end{aligned}$$

```
                    if b then mdropS vs
                    else return (Cons v vs)
```

Given an initial list of roots, *graphtrav* first creates an empty set, then executes *gtrav*, obtaining a list of vertices and a set, and finally discards the set and returns the resulting list. In each iteration, the function *gtrav* starts with an exploration of the current list of roots in order to find a vertex that has not been visited yet. To this end, it removes from the front of that list every vertex u that is marked as visited until, either an unvisited vertex is met, or the end of the list is reached. This task is performed by the function *mdropS*.

After the application of *mdropS*, we visit the vertex at the head of the input list, if still there is any, and mark it (by inserting it in the set). A new 'state' of the list of roots is also computed. This is performed by an auxiliary function, called *pol*, which encapsulates the administration policy used for the list of pending roots. That way, we obtain a formulation of graph traversal parameterized by a strategy.

Function *gtrav* can be expressed as a monadic unfold:

```
gtrav pol g = munfold_L k
  where
    k     :: List v → M v (L v (List v))
    k vs = do xs ← mdropS vs
              case xs of
                Nil        → return (Left ())
                Cons v ys → do insMS v
                               return (Right (v, pol g v ys))
```

Particular traversal strategies are obtained by providing specific policies:

Depth-first traversal. This is achieved by managing the list of pending roots as a stack.

```
dfsTrav     :: Eq v ⇒ Graph v → List v → List v
dfsTrav g   = graphtrav dfsPol g

dfsPol g v vs = adj g v ++ vs
```

Breath-first traversal. This is achieved by managing the list of pending roots as a queue.

```
bfsTrav     :: Eq v ⇒ Graph v → List v → List v
bfsTrav g   = graphtrav bfsPol g

bfsPol g v vs = vs ++ adj g v
```

4.6 Monadic Hylomorphism

Monadic hylomorphism is a pattern of recursion that represents general recursive monadic functions.

$$
\begin{aligned}
&\mathit{mhylo} \quad :: \mathit{Monad}\ m \Rightarrow (F\ b \to m\ b) \to (a \to m\ (F\ a)) \to (a \to b)\\
&\mathit{mhylo}\ h\ g = h \bullet \widehat{F}\ (\mathit{mhylo}\ h\ g) \bullet g
\end{aligned}
$$

Example 17. The following are instances of monadic hylomorphism for specific datatypes. Again, we assume a left to right product distribution $\mathit{dist}_\times$.

Lists

$$
\begin{aligned}
&\mathit{mhylo}_L :: \mathit{Monad}\ m \Rightarrow\\
&\qquad (m\ c, a \to c \to m\ c) \to (b \to m\ (L\ a\ b)) \to (b \to m\ c)\\
&\mathit{mhylo}_L\ (h_1, h_2)\ g = \mathit{mh}_L\\
&\quad \textbf{where}\\
&\qquad \mathit{mh}_L\ b = \textbf{do}\ x \leftarrow g\ b\\
&\qquad\qquad\qquad \textbf{case}\ x\ \textbf{of}\\
&\qquad\qquad\qquad\quad \mathit{Left}\ () \qquad\ \to h_1\\
&\qquad\qquad\qquad\quad \mathit{Right}\ (a, b') \to \textbf{do}\ c \leftarrow \mathit{mh}_L\ b'\\
&\qquad\qquad\qquad\qquad\qquad\qquad\ \ h_2\ a\ c
\end{aligned}
$$

Leaf-labelled binary trees

$$
\begin{aligned}
&\mathit{mhylo}_B :: \mathit{Monad}\ m \Rightarrow\\
&\qquad (a \to m\ c, c \to c \to m\ c) \to (b \to m\ (B\ a\ b)) \to (b \to c)\\
&\mathit{mhylo}_B\ (h_1, h_2)\ g = \mathit{mh}_B\\
&\quad \textbf{where}\\
&\qquad \mathit{mh}_B\ b = \textbf{do}\ x \leftarrow g\ b\\
&\qquad\qquad\qquad \textbf{case}\ x\ \textbf{of}\\
&\qquad\qquad\qquad\quad \mathit{Left}\ a \qquad\quad \to h_1\ a\\
&\qquad\qquad\qquad\quad \mathit{Right}\ (b1, b2) \to \textbf{do}\ c1 \leftarrow \mathit{mh}_B\ b1\\
&\qquad\qquad\qquad\qquad\qquad\qquad\ \ c2 \leftarrow \mathit{mh}_B\ b2\\
&\qquad\qquad\qquad\qquad\qquad\qquad\ \ h_2\ c1\ c2
\end{aligned}
$$

Rose trees

$$
\begin{aligned}
&\mathit{mhylo}_R :: \mathit{Monad}\ m \Rightarrow\\
&\qquad (a \to [c] \to m\ c) \to (b \to m\ (R\ a\ b)) \to (b \to m\ c)\\
&\mathit{mhyloh}\ h\ g\ b = \textbf{do}\ (a, bs) \leftarrow g\ b\\
&\qquad\qquad\qquad\quad cs \qquad \leftarrow \mathit{mapM}\ (\mathit{mhylo}_R\ h\ g)\ bs\\
&\qquad\qquad\qquad\quad h\ a\ cs
\end{aligned}
$$

□

The fusion laws for monadic hylomorphism are a consequence of those for monadic fold and monadic unfold.

MHylo Fusion If $\widehat{F}$ preserves Kleisli compositions,

$$f \text{ strict } \wedge\ f \bullet h = k \bullet \widehat{F}\, f \ \Rightarrow\ f \bullet \mathit{mhylo}\ h\ g = \mathit{mhylo}\ k\ g$$

$$g' \bullet f = \widehat{F}\, f \bullet g \ \Rightarrow\ \mathit{mhylo}\ h\ g' \bullet f = \mathit{mhylo}\ h\ g$$

In the first law *mmap* needs to be strictness-preserving.

MHylo Pure Fusion

$$f \text{ strict } \wedge \ mmap\ f \circ h = k \circ F\ f \ \Rightarrow \ mmap\ f \circ mhylo\ h\ g = mhylo\ k\ g$$

$$g' \circ f = mmap\ (F\ f) \circ g \ \Rightarrow \ mhylo\ h\ g' \circ f = mhylo\ h\ g$$

In the first law *mmap* needs to be strictness-preserving.

MHylo-Fold Fusion If *mmap* is strictness-preserving,

$$\begin{array}{l} \quad \tau :: \forall\ a\ .\ (F\ a \to a) \to (G\ a \to m\ a) \\ \Rightarrow \\ \quad mmap\ (fold\ h) \circ mhylo\ (\tau\ in_F)\ g = mhylo\ (\tau\ h)\ g \end{array}$$

Unfold-MHylo Fusion

$$\begin{array}{l} \quad \sigma :: \forall\ a\ .\ (a \to F\ a) \to (a \to m\ (G\ a)) \\ \Rightarrow \\ \quad mhylo\ h\ (\sigma\ out_F) \circ unfold\ g = mhylo\ h\ (\sigma\ g) \end{array}$$

4.7 Depth-first search algorithms

The references [16, 17, 15] show the advantages of explicitly maintaining the *depth-first spanning forest* of a graph when implementing DFS algorithms in a lazy functional language. The construction of the depth-first forest is performed in two stages. In the first phase a forest of (possibly infinite) trees is generated. Each tree is rooted with a vertex from a given list of entry points to the graph and contains all vertices in the graph reachable from that root. The second phase runs a prune process, which traverses the forest in depth-first order, discarding all subtrees whose roots have occurred previously. This *generate-then-prune* strategy turns out to be the natural solution in the context of a lazy functional language. Indeed, because of lazy evaluation, deeper levels of the trees are generated only if and when demanded by the prune process.

In this subsection, we show that the depth-first forest construction can be structured using monadic recursion schemes.

Generation Like in Subsection 4.5, we assume a graph representation that supports a function $adj :: Eq\ v \Rightarrow Graph\ v \to v \to List\ v$ which returns the adjacency list of each node of a graph.

The generation of a (rose) tree containing all vertices in the graph reachable from a vertex v is defined by,

$$\begin{array}{l} gen \quad :: Eq\ v \Rightarrow Graph\ v \to v \to Rose\ v \\ gen\ g\ v = Fork\ v\ (List\ (gen\ g)\ (adj\ g\ v)) \end{array}$$

This function is naturally an unfold

$$gen\ g = unfold_R\ (id\ \triangle\ adj\ g)$$

The generation of a forest from a given list of vertices is then obtained by mapping each vertex of the list with function *gen*.

$$\begin{aligned} &fgen\quad :: Eq\ v \Rightarrow Graph\ v \to List\ v \to List\ (Rose\ a) \\ &fgen\ g = List\ (gen\ g) \end{aligned}$$

Pruning Pruning traverses the forest in depth-first order, discarding all subtrees whose roots have occurred previously. Analogous to graph traversals, pruning needs to maintain a set (of *marks*) to keep track of the already visited nodes. This suggest the use of the same monadic abstract data type.

In the pruning process we will use a datatype of rose trees extended with an empty tree constructor.

$$\mathbf{data}\ ERose\ a = ENull \mid EFork\ a\ (List\ (ERose\ a))$$

$$\begin{aligned} &\mathbf{type}\ ER_a\ b \quad = () + a\ \times\ List\ b \\ &ER_a \qquad\qquad :: (b \to c) \to (ER_a\ b \to ER_a\ c) \\ &ER_a\ f \qquad\quad = id + id\ \times\ List\ f \end{aligned}$$

When we find a root that has occurred previously, we prune the whole subtree. The function that prunes an individual rose tree is defined by

$$\begin{aligned} &prune_R :: \qquad\qquad Eq\ v \Rightarrow Rose\ v \to M\ v\ (ERose\ v) \\ &prune_R\ (Fork\ v\ rs) = \mathbf{do}\ b \leftarrow memMS\ v \\ &\qquad\qquad\qquad\qquad\quad \mathbf{if}\ b \\ &\qquad\qquad\qquad\qquad\quad \mathbf{then}\ return\ ENull \\ &\qquad\qquad\qquad\qquad\quad \mathbf{else\ do}\ insMS\ v \\ &\qquad\qquad\qquad\qquad\qquad\qquad rs' \leftarrow mapM\ prune_R\ rs \\ &\qquad\qquad\qquad\qquad\qquad\qquad return\ (EFork\ v\ rs') \end{aligned}$$

This function can be written as a monadic unfold:

$$\begin{aligned} &prune_R = munfold_{ER}\ g \\ &\quad \mathbf{where} \\ &\qquad g\ (Fork\ v\ rs) = prStep\ (v, rs) \\ &\qquad prStep\ (v, rs) = \mathbf{do}\ b \leftarrow memMS\ v \\ &\qquad\qquad\qquad\qquad\quad \mathbf{if}\ b \\ &\qquad\qquad\qquad\qquad\quad \mathbf{then}\ return\ (Left\ ()) \\ &\qquad\qquad\qquad\qquad\quad \mathbf{else\ do}\ insMS\ v \\ &\qquad\qquad\qquad\qquad\qquad\qquad return\ (Right\ (v, rs)) \end{aligned}$$

such that its coalgebra can be written as $g = prStep \circ out_{R_v}$.

Pruning a forest just consists of pruning the trees in sequence:

$$fpruneR :: Eq\ v \Rightarrow List\ (Rose\ a) \to M\ a\ (List\ (ERose\ v))$$

$$fpruneR = mapM\ prune_R$$

A drawback of this solution is that the resulting forest contains many unnecessary empty trees, which could be dropped if we convert the resulting extended rose trees into rose trees again. The conversion is performed by simply traversing the forest of extended rose tress, *cleaning* all occurrences of the empty tree:

$$\begin{aligned}
&fclean :: List\ (ERose\ a) \to List\ (Rose\ a)\\
&fclean = collect \circ List\ clean
\end{aligned}$$

$$\begin{aligned}
&clean\ &&:: ERose\ a \to Maybe\ (Rose\ a)\\
&clean\ ENull\ &&= Nothing\\
&clean\ (EFork\ a\ rs) &&= Just\ (Fork\ a\ (fclean\ rs))
\end{aligned}$$

$$\begin{aligned}
&collect\ &&:: List\ (Maybe\ a) \to List\ a\\
&collect\ Nil &&= Nil\\
&collect\ (Cons\ m\ ms) &&= \mathbf{case}\ m\ \mathbf{of}\\
&&&\qquad Nothing \to collect\ ms\\
&&&\qquad Just\ a\ \ \to Cons\ a\ (collect\ ms)
\end{aligned}$$

Clearly, both *clean* and *collect* are folds, $clean = fold_{ER}\ cl$ and $collect = fold_L\ coll$, for suitable algebras cl and $coll = (coll1, coll2)$, respectively.

Finally, we define the function that prunes a forest of rose trees, returning the rose trees that remain:

$$\begin{aligned}
&prune :: Eq\ v \Rightarrow List\ (Rose\ a) \to M\ a\ (List\ (Rose\ a))\\
&prune = mmap\ fclean \circ fpruneR
\end{aligned}$$

Computing the depth-first forest Now we define a function *dfs* that computes the depth-first spanning forest of a graph reachable from a given list of vertices.

$$\begin{aligned}
&dfs\ \ :: Eq\ v \Rightarrow Graph\ v \to List\ v \to List\ (Rose\ v)\\
&dfs\ g = runMS \circ prune \circ fgen\ g
\end{aligned}$$

We use function *runMS* to hide the monadic state from the outside world. That way, *dfs* is externally regarded as a purely functional. The internal components of *dfs* can be fused as the following calculation shows.

$$\begin{aligned}
&\quad prune \circ fgen\ g\\
&= \quad \{\ \text{function definitions}\ \}\\
&\quad mmap\ (collect \circ List\ clean) \circ mapM\ (prune_R) \circ List\ (gen\ g)\\
&= \quad \{\ mapM = \widehat{List}\ \text{and property: } \widehat{F}\ f \circ F\ g = \widehat{F}\ (f \circ g)\ \}\\
&\quad mmap\ (collect \circ List\ clean) \circ mapM\ (prune_R \circ gen\ g)\\
&= \quad \{\ \text{functor } mmap\ \}\\
&\quad mmap\ collect \circ mmap\ (List\ clean) \circ mapM\ (prune_R \circ gen\ g)
\end{aligned}$$

$$
\begin{aligned}
&= \quad \{ \text{ property: } mmap\ (F\ f) \circ \widehat{F}\ g = \widehat{F}\ (mmap\ f \circ g) \ \} \\
&\quad mmap\ collect \circ mapM\ (mmap\ clean \circ prune_R \circ gen\ g) \\
&= \quad \{ \text{ define: } gpc\ g = mmap\ clean \circ prune_R \circ gen\ g \ \} \\
&\quad mmap\ collect \circ mapM\ (gpc\ g) \\
&= \quad \{ \text{ property: } mmap\ (fold\ h) \circ \widehat{D}\ f = fold\ (mmap\ h \circ \widehat{F}\ f\ id) \ \} \\
&\quad fold_L\ (mmap\ coll \circ \widehat{L}\ (gpc\ g)\ id)
\end{aligned}
$$

We call *gp* (for generate then prune) the resulting fold. Inlining, we get the following recursive definition:

$$
\begin{aligned}
&gp &&:: Eq\ v \Rightarrow Graph\ v \to List\ v \to M\ v\ (List\ (Rose\ v)) \\
&gp\ g\ Nil &&= return\ Nil \\
&gp\ g\ (Cons\ v\ vs) &&= \mathbf{do}\ x \leftarrow gpc\ g\ v \\
&&&\qquad rs \leftarrow gp\ g\ vs \\
&&&\qquad return\ (\mathbf{case}\ x\ \mathbf{of} \\
&&&\qquad\qquad Left\ () \to rs \\
&&&\qquad\qquad Right\ r \to Cons\ r\ rs)
\end{aligned}
$$

Now, let us analyze function *gpc*, which expresses how individual trees are generated, pruned and cleaned in a shot.

$$
\begin{aligned}
&gpc :: Eq\ v \Rightarrow Graph\ g \to v \to M\ v\ (Maybe\ (Rose\ a)) \\
&gpc\ g = mmap\ clean \circ prune_R \circ gen\ g
\end{aligned}
$$

This definition can also be simplified:

$$
\begin{aligned}
&\quad mmap\ clean \circ prune_R \circ gen\ g \\
&= \quad \{ \text{ function definitions } \} \\
&\quad mmap\ (fold_{ER}\ cl) \circ munfold_{ER}\ (prStep \circ out_{R_v}) \circ unfold_R\ (id\ \vartriangle\ adj\ g) \\
&= \quad \{ \text{ define: } \sigma\ j = prStep \circ j \ \} \\
&\quad mmap\ (fold_{ER}\ cl) \circ munfold_{ER}\ (\sigma\ out_{R_v}) \circ unfold_R\ (id\ \vartriangle\ adj\ g) \\
&= \quad \{ \text{ unfold-munfold fusion } \} \\
&\quad mmap\ (fold_{ER}\ cl) \circ munfold_{ER}\ (\sigma\ (id\ \vartriangle\ adj\ g)) \\
&= \quad \{ \text{ factorization prop.: } mmap\ (fold\ h) \circ munfold\ g = mhylo\ \widehat{h}\ g \ \} \\
&\quad mhylo_{ER}\ \widehat{cl}\ (prStep \circ (id\ \vartriangle\ adj\ g))
\end{aligned}
$$

Inlining, we obtain:

$$
\begin{aligned}
&gpc\ g\ v = \mathbf{do}\ b \leftarrow memMS\ v \\
&\qquad\qquad \mathbf{if}\ b \\
&\qquad\qquad \mathbf{then}\ return\ Nothing \\
&\qquad\qquad \mathbf{else\ do}\ insMS\ v \\
&\qquad\qquad\qquad ms \leftarrow mapM\ (gpc\ g)\ (adj\ g\ v) \\
&\qquad\qquad\qquad return\ (Just\ (Fork\ v\ (collect\ ms)))
\end{aligned}
$$

Depth-first traversal To illustrate the use of the depth-first forest, we compute the depth-first traversal of a graph by traversing the forest in preorder. Other DFS algorithms under the same approach can be found in [16, 17, 15, 24].

The preorder of a forest can be defined by

$$\begin{aligned}
&fpreorder :: List\ (Rose\ v) \to List\ v\\
&fpreoredr = concat \circ List\ preorder
\end{aligned}$$

$$\begin{aligned}
&preorder\ \ :: Rose\ v \to List\ v\\
&preorder\ \ = fold_R\ (Cons \circ (id\ \times\ concat))
\end{aligned}$$

We compute the depth-first traversal of a graph by listing the depth-first forest in preorder:

$$\begin{aligned}
&dfsTrav\ \ \ :: Eq\ v \Rightarrow Graph\ g \to List\ v \to List\ v\\
&dfsTrav\ g = fpreorder \circ dfs\ g
\end{aligned}$$

We show now how the generation of the intermediate depth-first forest can be eliminated using fusion.

$$\begin{aligned}
&fpreorder \circ dfs\ g\\
=\quad &\{\ \text{function definitions}\ \}\\
&concat \circ List\ preorder \circ runMS \circ mmap\ collect \circ mapM\ (gpc\ g)\\
=\quad &\{\ \text{parametricity property: } f \circ runMS = runMS \circ mmap\ f\ \ \}\\
&runMS \circ mmap\ (concat \circ List\ preorder) \circ mmap\ collect \circ mapM\ (gpc\ g)\\
=\quad &\{\ \text{functor } mmap\ \}\\
&runMS \circ mmap\ (concat \circ List\ preorder \circ collect) \circ mapM\ (gpc\ g)\\
=\quad &\{\ \text{map-fold fusion, define } pjoin = uncurry\ (+\!\!+) \circ (preorder\ \times\ id)\ \}\\
&runMS \circ mmap\ (fold_L\ (Nil, pjoin) \circ collect) \circ mapM\ (gpc\ g)\\
=\quad &\{\ \text{define: } \tau \text{ (see below)}\ \}\\
&runMS \circ mmap\ (fold_L\ (Nil, pjoin) \circ fold_L\ (\tau\ (Nil, Cons))) \circ mapM\ (gpc\ g)\\
=\quad &\{\ \text{fold-fold fusion}\ \}\\
&runMS \circ mmap\ (fold_L\ (\tau\ (Nil, pjoin))) \circ mapM\ (gpc\ g)\\
=\quad &\{\ \text{property: } mmap\ (fold\ h) \circ \widehat{D}\ f = fold\ (M\ h \circ \widehat{F}\ f\ id)\ \}\\
&runMS \circ fold_L\ (mmap\ (\tau\ (Nil, pjoin)) \circ \widehat{L}\ (gpc\ g)\ id)
\end{aligned}$$

Function τ is given by:

$$\begin{aligned}
&\tau\ \ \ \ \ \ \ \ :: (b, a \to b \to b) \to (b, Maybe\ a \to b \to b)\\
&\tau\ (h_1, h_2) = (h_1,\\
&\qquad\qquad \lambda m\ b \to \textbf{case}\ m\ \textbf{of}\\
&\qquad\qquad\qquad\qquad Nothing \to b\\
&\qquad\qquad\qquad\qquad Just\ a\ \ \ \to h_2\ a\ b)
\end{aligned}$$

The property $f \circ runMS = runMS \circ mmap\ f$ is an example of a *parametricity property* or *free theorem* [29], which are properties that can be directly derived from the type of polymorphic functions.

Calling *mtrav* the $fold_L$ obtained in the derivation and inlining, we obtain this program:

$$
\begin{array}{lll}
mtrav & :: Eq\ v \Rightarrow Graph\ g \to List\ v \to M\ v\ (List\ v) \\
mtrav\ g\ Nil & = return\ Nil \\
mtrav\ g\ (Cons\ v\ vs) & = \mathbf{do}\ x \leftarrow gpc\ g\ v \\
 & \qquad\qquad as \leftarrow mtrav\ g\ vs \\
 & \qquad\qquad return\ (\mathbf{case}\ x\ \mathbf{of} \\
 & \qquad\qquad\qquad\qquad Nothing \to as \\
 & \qquad\qquad\qquad\qquad Just\ r \quad \to preorder\ r +\!\!+ as)
\end{array}
$$

4.8 A more practical approach

The monadic program schemes shown so far were all derived from the lifting construction presented in Subsection 4.1.

$$
\begin{array}{ccc}
a & \xrightarrow{\quad mhylo \quad} & m\ b \\
{\scriptstyle g}\downarrow & & \uparrow{\scriptstyle h^\star} \\
m\ (F\ a) & \xrightarrow[(\widehat{F}\ mhylo)^\star]{} & m\ (F\ b)
\end{array}
$$

However, despite its theoretical elegance, this construction suffers from an important drawback that hinders the practical use of the program schemes derived from it. The origin of the problem is the compulsory use of the distributive law $dist_F$ associated with $\widehat{F}$ as unique way of joining the effects produced by the recursive calls. It is not hard to see that this structural requirement introduces a restriction in the kind of functions that can be formulated in terms of the monadic program schemes. To see a simple example, consider the following function that prints the values contained in a leaf-labelled binary tree, with a `'+'` symbol in between.

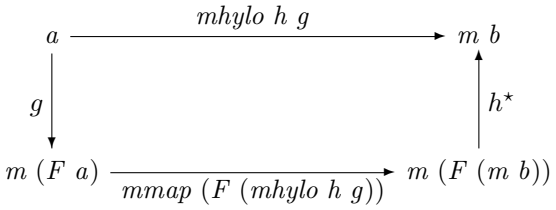
$$
\begin{array}{lll}
printTree & :: Show\ a \Rightarrow Btree\ a \to IO\ () \\
printTree\ (Leaf\ a) & = putStr\ (show\ a) \\
printTree\ (Join\ t\ t') & = \mathbf{do}\ \{printTree\ t; putStr\ \texttt{"+"}; printTree\ t'\}
\end{array}
$$

For instance, when applied to the tree $Join\ (Join\ (Leaf\ 1)\ (Leaf\ 2))\ (Leaf\ 3)$, *printTree* returns an I/O action that, when performed, prints the string `"1+2+3"` on the standard output. Since it is a monadic function defined by structural recursion on the input tree, one could expect that it can be written as a monadic fold. However, this is impossible. To see why, recall that the definition of monadic fold for binary trees follows a pattern of recursion of this form:

$$mf_B\ (Leaf\ a) \quad = h_1\ a$$
$$mf_B\ (Join\ t\ t') = \mathbf{do}\ \{ y \leftarrow mf_B\ t; y' \leftarrow mf_B\ t'; h_2\ y\ y' \}$$

when a left to right product distribution $dist_\times$ is assumed. According to this pattern, in every recursive step the computations returned by the recursive calls must be performed in sequence, one immediately after the other. This means that there is no way of interleaving additional computations between the recursive calls, precisely the contrary of what *printTree* does. This limitation is a consequence of having fixed the use of a monadic extension $\widehat{F}$ as unique alternative to structure the recursive calls in monadic program schemes. In other words, the fault is in the lifting construction itself.

This problem can be overcome by introducing a more flexible construction for the definition of monadic hylomorphism:



There are two differences between this definition and the one shown previously. First, this definition avoids the use of a monadic extension $\widehat{F}$, and second, the type of h has changed with respect to the type it had previously. Now, its type is $F\ (m\ b) \to m\ b$. Therefore, strictly speaking, h is not more a monadic F-algebra, but an F-algebra with monadic carrier. As a consequence of these modifications, in the new scheme the computations returned by the recursive calls are not performed apart in a separate unit any more. Instead, they are provided to the algebra h, which will specify the order in that these computations are performed, as well as their possible interleaving with other computations.

It is easy to see that this new version of monadic hylomorphism subsumes the previous one. In fact, a previous version of monadic hylomorphism (with monadic algebra $h :: F\ b \to m\ b$) can be represented in terms of the new one by taking $h \bullet dist_F$ as algebra, that is, $mhylo_{old}\ h\ g = mhylo\ (h \bullet dist_F)\ g$. This means that the definitions, examples and laws based on the lifting construction can all be regarded as special cases of the new construction.

Of course, we can derive new definitions of monadic fold and unfold from the new construction. For monadic unfold, the algebra of the monadic hylomorphism should only join the effects of the computations returned by the recursive calls, and build the values of the data structure using the constructors. Therefore,

$$munfold \quad :: Monad\ m \Rightarrow (a \to m\ (F\ a)) \to a \to m\ \mu F$$
$$munfold\ g = mhylo\ (\widehat{in_F} \bullet dist_F)\ g$$

Interestingly, this definition turns out to be equivalent to the one presented in Subsection 4.4. A definition of monadic fold is obained by taking $g = \widehat{out_F}$. By applying simplifications concerning the monad operations, we obtain:

$$
\begin{aligned}
&\mathit{mfold} \quad :: \mathit{Monad}\ m \Rightarrow (F\ (m\ a) \to m\ a) \to \mu F \to m\ a \\
&\mathit{mfold}\ h = h \circ F\ (\mathit{mfold}\ h) \circ \mathit{out}_F
\end{aligned}
$$

Observe that this is nothing but the definition of fold (see Subsection 2.2) with the additional restriction that the algebra must be of monadic carrier. For instance, for leaf-labelled binary trees, $h :: (a \to m\ b, m\ b \to m\ b \to mb)$. Now, we can write *printTree* as a monadic fold:

$$
\mathit{printTree} = \mathit{mfold}_B\ (\mathit{putStr} \circ \mathit{show}, \lambda m\ m' \to \mathbf{do}\ \{m; \mathit{putStr}\ \texttt{"+"}; m'\})
$$

Finally, we present a pair of fusion laws for the new version of monadic hylomorphism.

MHylo-Fold Fusion If *mmap* is strictness-preserving,

$$
\begin{aligned}
& \tau :: \forall\ a\ .\ (F\ a \to a) \to (G\ (m\ a) \to m\ a) \\
\Rightarrow & \\
& \mathit{mmap}\ (\mathit{fold}\ h) \circ \mathit{mhylo}\ (\tau\ \mathit{in}_F)\ g = \mathit{mhylo}\ (\tau\ h)\ g
\end{aligned}
$$

Unfold-MHylo Fusion

$$
\begin{aligned}
& \sigma :: \forall\ a\ .\ (a \to F\ a) \to (a \to m\ (G\ a)) \\
\Rightarrow & \\
& \mathit{mhylo}\ h\ (\sigma\ \mathit{out}_F) \circ \mathit{unfold}\ g = \mathit{mhylo}\ h\ (\sigma\ g)
\end{aligned}
$$

5 A Program Fusion Tool

The research presented in this paper motivated the development of an interactive program fusion tool that performs the automatic elimination of intermediate data structures from both purely-functional programas and programs with effects. The system accepts as input standard functional programs written in a subset of Haskell and translates them into an internal representation in terms of (monadic) hylomorphism. The tool is based on ideas and algorithms used in the design of the HYLO system [23]. In addition to the manipulation of programs with effects, our system extends HYLO with the treatment of some other shapes of recursion for purely-functional programs.

The following web page contains documentation and versions of the tool:

`http://www.fing.edu.uy/inco/proyectos/fusion`

References

1. S. Abramsky and A. Jung. Domain theory. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 3, pages 1–168. Clarendon Press, 1994.
2. N. Benton, J. Hughes, and E. Moggi. Monads and Effects. In *APPSEM 2000 Summer School,* LNCS 2395. Springer-Verlag, 2002.

3. R. Bird. *Introduction to Functional Programming using Haskell,* 2nd edition. Prentice-Hall, UK, 1998.
4. R.S. Bird and O. de Moor. *Algebra of Programming.* Prentice Hall, UK, 1997.
5. Chih-Ping Chen and P. Hudak. Rolling Your Own Mutable ADT—A Connection Between Linear Types and Monads. In *24th Symposium on Principles of Programming Languages*, pages 54–66. ACM, January 1997.
6. M.M. Fokkinga. *Law and Order in Algorithmics.* PhD thesis, Universiteit Twente, The Netherlands, 1992.
7. M.M. Fokkinga. Monadic maps and folds for arbitrary datatypes. Memoranda Informatica 94-28, University of Twente, June 1994.
8. J. Gibbons. Calculating Functional Programs. In *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction,* LNCS 2297, pages 148–203. Springer-Verlag, January 2002.
9. J. Gibbons and G. Jones. The Under-Appreciated Unfold. In *3rd. International Conference on Functional Programming.* ACM, September 1998.
10. A. Gill. *Cheap Deforestation for Non-strict Functional Languages.* PhD thesis, Department of Computing Science, University of Glasgow, UK, 1996.
11. A. Gill, J. Launchbury, and S. Peyton Jones. A Shortcut to Deforestation. In *Conference on Functional Programming and Computer Architecture*, 1993.
12. G. Hutton. Fold and Unfold for Program Semantics. In *3rd. International Conference on Functional Programming.* ACM, September 1998.
13. S. Peyton Jones. Tackling the awkward squad: monadic input/output, concurrency, exceptions, and foreign language calls in Haskell. In *Engineering theories of software construction, Marktoberdorf Summer School 2000.* NATO ASI Series, IOS press, 2001.
14. S. Peyton Jones and J. Launchbury. Lazy functional state threads. In *Symposium on Programming Language Design and Implementation (PLDI'94)*, pages 24–35. ACM, 1994.
15. D. King. *Functional Programming and Graph Algorithms.* PhD thesis, Department of Computing Science, University of Glasgow, UK, March 1996.
16. D. King and J. Launchbury. Structuring depth-first search algorithms in Haskell. In *22nd Symposium on Principles of Programming Languages*, pages 344–354. ACM, 1995.
17. J. Launchbury. Graph Algorithms with a Functional Flavour. In *Advanced Functional Programming,* LNCS 925. Springer-Verlag, 1995.
18. E. Meijer, M. Fokkinga, and R. Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *Functional Programming Languages and Computer Architecture'91,* LNCS 523. Springer-Verlag, August 1991.
19. E. Meijer and G. Hutton. Bananas in space: Extending fold and unfold to exponential types. In *Functional Programming Languages and Computer Architecture'95*, pages 324–333, 1995.
20. E. Meijer and J. Jeuring. Merging Monads and Folds for Functional Programming. In *Advanced Functional Programming,* LNCS 925, pages 228–266. Springer-Verlag, 1995.
21. E. Moggi. Notions of Computation and Monads. *Information and Computation*, 93:55–92, 1991.
22. P.S. Mulry. Lifting Theorems for Kleisli Categories. In *9th International Conference on Mathematical Foundations of Programming Semantics,* LNCS 802, pages 304–319. Springer-Verlag, 1993.

23. Y. Onoue, Z. Hu, H. Iwasaki, and M. Takeichi. A Calculational Fusion System HYLO. In *IFIP TC 2 Working Conference on Algorithmic Languages and Calculi, Le Bischenberg, France*, pages 76–106. Chapman & Hall, February 1997.
24. A. Pardo. *A Calculational Approach to Recursive Programs with Effects.* PhD thesis, Technische Universität Darmstadt, October 2001.
25. A. Pardo. Fusion of Recursive Programs with Computational Effects. *Theoretical Computer Science*, 260:165–207, 2001.
26. S. Peyton-Jones and P. Wadler. Imperative Functional Programming. In *20th Annual Symposium on Principles of Programming Languages*, Charlotte, North Carolina, 1993. ACM.
27. A. Takano and E. Meijer. Shortcut to Deforestation in Calculational Form. In *Functional Programming Languages and Computer Architecture'95*, 1995.
28. D. Tuijnman. *A Categorical Approach to Functional Programming.* PhD thesis, Fakultät für Informatik, Universität Ulm, Germany, January 1996.
29. P. Wadler. Theorems for free! In *4th International Conference on Functional Programming and Computer Architecture*, London, 1989.
30. P. Wadler. Deforestation: transforming programs to eliminate trees. *Theoretical Computer Science*, 73:231–248, 1990.
31. P. Wadler. Monads for functional programming. In *Advanced Functional Programming,* LNCS 925. Springer-Verlag, 1995.