PEDECIBA Informática
Instituto de Computación – Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay

Tesis de Maestría
en Informática

Hfusion : a fusion tool based on acid rain plus extensions

Facundo Domínguez Laumann

Supervisor y Orientador: Dr. Alberto Pardo

Montevideo, agosto de 2009

Hfusion: a fusion tool based on acid rain plus extensions
Domínguez Laumann, Facundo
ISSN 0797-6410
Tesis de Maestría en Informática
Reporte Técnico RT 09-16
PEDECIBA Area Informática
Instituto de Computación – Facultad de Ingeniería
Universidad de la República.
Montevideo, Uruguay, agosto de 2009

Abstract

When constructing programs, it is a usual practice to compose algorithms that solve simpler problems to solve a more complex one. This principle adapts so well to software development because it provides a structure to understand, design, reuse and test programs.

In functional languages, algorithms are usually connected through the use of intermediate data structures, which carry the data from one algorithm to another one. The data structures impose a load on the algorithms to allocate, traverse and deallocate them. To alleviate this inefficiency, automatic program transformations have been studied, which produce equivalent programs that make less use of intermediate data structures.

We present a set of automatic program transformation techniques based on algebraic laws known as Acid Rain. These techniques allow to remove intermediate data structures in programs containing primitive recursive functions, mutually recursive functions and functions with multiple recursive arguments.

We also provide an experimental implementation of our techniques which allows their application on user supplied programs.

Keywords: Deforestation, Acid Rain, Fusion, Hylomorphism, Paramorphism, Functional Programming Languages, Generic Programming.

Acknowledgements

This work would not have reached its current form if it had not been for the careful examination of Alberto Pardo. He kindly provided me with the appropriate references many times, when I believed to have invented the wheel; and he asked me to bring to light things that I would have rather swept under the carpet!

Many thansk to reviewers Jose Nuno Oliveira, Javier Baliosian and Gustavo Betarte who provided valuable comments to prepare the final version.

This work also owes to Lucía Lafourcade, my life companion, who believed that I was going to make it when I felt hopelessly hung half-way in a slope.

The research presented herein was partly supported by a one-year scholarship from PEDECIBA (Programa de Desarrollo de Ciencias Básicas – Universidad de la República), and was partly done in the context of the project *Deforestación en presencia de efectos*, funded by PDT – DINACYT (Programa de Desarrollo Tecnológico – Dirección Nacional de Ciencia y Tecnología).

Contents

Chapter 1

Introduction

Most often, programming languages allow to express programs in a modular and succinct way through the composition of subroutines or functions which perform more specific tasks. Functional languages are not an exception in this regard, but for them, compositions can be significantly more expensive to execute, sometimes. Consider the following Haskell[1] function:

$$f\ p\ n = \mathit{sum}\ (\mathit{filter}\ p\ [1\,..\,n])$$

This program sums all the numbers between 1 and n that satisfy a given predicate p. To build it, we have used standard functions that work over lists. Function *sum* sums the numbers in a list, whereas function *filter* p builds a list by selecting from the input list the elements that satisfy predicate p. The expression $[m\,..\,n]$ builds a list with the integers from m to n.

The individual parts are simple enough so that the whole program can be understood from them without much effort. But when composing the parts, we rely on intermediate lists that transfer the data from one stage to the next. We first build a list with the integers from 1 to n and then we build a new list with the numbers satisfying p. These intermediate lists imply more work for allocating, examining and deallocating their nodes, and therefore are not desirable from the runtime standpoint. It could be possible to write a program which computes the same but does not construct the intermediate lists:

$$\begin{array}{l} f\ p\ n = g\ p\ n\ 0\ 1 \\ g\ p\ n\ \mathit{acc}\ m = \\ \quad \textbf{if}\ m > n\ \textbf{then}\ \mathit{acc} \\ \quad\quad \textbf{else if}\ p\ m\ \textbf{then}\ g\ p\ n\ (m + \mathit{acc})\ (m + 1) \\ \quad\quad\quad\quad \textbf{else}\ g\ p\ n\ \mathit{acc}\ (m + 1) \end{array}$$

In this version we certainly avoid the construction of the intermediate lists, but it takes longer to write and it is harder to understand. The intelligibility

[1] We will manipulate throughout our work Haskell programs [Bir98, Jon03].

quickly decreases when attempting to eliminate more intermediate structures in larger programs.

To solve this dilemma between intelligibility and efficiency, automatic program transformation techniques have been formulated, so that the programmer could write and read the modular version of a program while the compiler derives the efficient one (see e.g. [Wad88, Chi92, dMS99, JL98, GLPJ93], and some others with a more algebraic approach [BdM97, SF93, TM95, HIT96b, HIT96a]).

Because other tree-like structures can be used to pass intermediate data between program components, such transformations are known as deforestation [Wad88]. As program components are replaced by monolithic functions performing all of their tasks, the transformation is also called fusion.

In this thesis we present the essentials for fusing programs using one of the algebraic approaches. This is a reformulation and reelaboration of previous work based on describing programs in terms of a recursive scheme called hylomorphism (sometimes referred to simply as hylo), and applying certain laws known as *acid rain* [TM95, OHIT97, Sch00]. As first step we implemented those techniques [Dom04, DP06a], and now we extend the algorithms in order to broaden the application of fusion to more cases.

Our main contributions are:

- A more concise formulation of the essential algorithms presented in [OHIT97, Sch00], which simplifies understanding and extension.
- An extension of the set of techniques available to pre-process definitions before applying the algorithms mentioned above.
- An extension of the previous algorithms to handle definitions which recurse over multiple arguments.
- An extension to handle mutually recursive definitions.
- An extension to handle primitive recursive definitions. This was presented in [DP06b].
- A set of techniques to broaden applicability of fusion laws in the light of the aforementioned extensions. Specifically,
  - we enable partial fusion when it is not possible to completely eliminate an intermediate data structure;
  - we propose a technique for eliminating data structures defined by the so-called *regular functors*; and
  - we identify a practical way to rewrite definitions to enable fusion of compositions like $sum \circ filter\ p$ when *sum* uses an accumulator.

- We provide an experimental implementation of a fusion tool named HFusion, available at

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

- We identify techniques used in our approach which could be put at the service of a somewhat related approach called shortcut deforestation [GLPJ93, Gil96, TM95].

The thesis is organized as follows. In Chapter 2 we present the basics of the theory which justifies correctness of hylo fusion transformations. In Chapter 3 we present the basic algorithms to perform hylo fusion. These are revised versions of the algorithms in [OHIT97, Sch00] together with some complementary algorithms which may be needed prior to fusion. In Chapter 4 extensions of our laws are presented for handling primitive recursive functions. In Chapter 5 we show several rewriting techniques that enable fusion when only part of the intermediate data structures can be eliminated. Chapter 6 is devoted to the extensions needed for handling mutually recursive definitions, and how mutual recursion can be used to deforest intermediate structures that have regular functors. In Chapter 7 we present the extension needed for handling definitions which recurse over multiple arguments, and we show some rewriting techniques that help getting more fusions from our extensions. In Chapter 8 we present the changes in performance in some sample programs after applying our transformations. Finally Chapter 9 relates our work to shortcut fusion, points at future work and draws some conclusions.

In Appendix A we provide counterexamples justifying some restrictions we impose in the input to our algorithms. In Appendix B we present our internal representation of hylomorphisms together with some remarks on how the algorithms are applied in the HFusion system.

Chapter 2

Fundamentals

This chapter presents the essential concepts we will be handling through the rest of the work (see e.g. [BdM97] for more details). They are used for the sole purpose of presenting our techniques; the resulting transformed programs which are returned by HFusion do not reference any of them.

Consider the following program, which sums the squares of all the leaves in a tree.

$$
\begin{aligned}
&\textbf{data } \mathit{BTree} = \mathit{Leaf}\ \mathit{Int} \mid \mathit{Join}\ \mathit{BTree}\ \mathit{BTree}\\
&\mathit{sumsqr} :: \mathit{BTree} \to \mathit{Int}\\
&\mathit{sumsqr} = \mathit{sumBT} \circ \mathit{sqrLeaves}\\
&\mathit{sumBT} :: \mathit{BTree} \to \mathit{Int}\\
&\mathit{sumBT}\ (\mathit{Leaf}\ i) \quad = i\\
&\mathit{sumBT}\ (\mathit{Join}\ l\ r) = \mathit{sumBT}\ l + \mathit{sumBT}\ r\\
&\mathit{sqrLeaves} :: \mathit{BTree} \to \mathit{BTree}\\
&\mathit{sqrLeaves}\ (\mathit{Leaf}\ i) \quad = \mathit{Leaf}\ (\mathit{sqr}\ i)\\
&\mathit{sqrLeaves}\ (\mathit{Join}\ l\ r) = \mathit{Join}\ (\mathit{sqrLeaves}\ l)\ (\mathit{sqrLeaves}\ r)\\
&\mathit{sqr}\ x = x * x
\end{aligned}
$$

Consider now the following term:

$$\mathit{Join}\ (\mathit{Join}\ (\mathit{Leaf}\ 1)\ (\mathit{Leaf}\ 9))\ (\mathit{Leaf}\ 16)$$

If we apply sumBT to it, we get the term

$$(+)\ ((+)\ (\mathit{id}\ 1)\ (\mathit{id}\ 9))\ (\mathit{id}\ 16)$$

which is precisely the result of replacing each occurrence of the constructor *Join* with $(+)$, and each occurrence of constructor *Leaf* with id.

Fusing the composition $\mathit{sumsqr} = \mathit{sumBT} \circ \mathit{sqrLeaves}$ is achieved by performing the previous substitution to the constructors occurring in the right hand side of *sqrLeaves* equations.

$$
\begin{aligned}
&\mathit{sumsqr} :: \mathit{BTree} \to \mathit{Int}\\
&\mathit{sumsqr}\ (\mathit{Leaf}\ i) \quad = \mathit{id}\ (\mathit{sqr}\ i)\\
&\mathit{sumsqr}\ (\mathit{Join}\ l\ r) = (+)\ (\mathit{sumsqr}\ l)\ (\mathit{sumsqr}\ r)
\end{aligned}
$$

In order to formally describe these manipulations, we will represent the definitions of *sumBT* and *sqrLeaves* in terms of a program scheme known as *hylomorphism*. The laws we use to fuse programs are of the form $f \circ g = h$, telling that a composition of hylomorphisms f and g can be eliminated by replacing it with an equivalent hylomorphism h, which is built from the components of f and g. After applying the law, the resulting hylomorphism h can be converted back into a recursive definition in a language like Haskell.

The following sections will introduce the concepts we need to express hylomorphisms as well as the fusion laws.

2.1 Functors

To write our definitions as hylomorphisms we need to structure them. That structure will be described by functors.

Definition 2.1 (Functor) A functor F is an operator which applies to types and functions, and which satisfies the following properties:

- $F\ f : F\ a\ \to\ F\ b$, for all $f : a \to b$
- $F\ \mathit{id} = \mathit{id}$
- $F\ (f \circ g) = F\ f \circ F\ g$

A functor F is specified by its action on types and functions, both written as F. For instance, we could specify the following functor:

$$
\begin{aligned}
&F\ a = \mathit{Either}\ \mathit{Int}\ (a \times a)\\
&F\ f = \mathit{either}\ \mathit{Left}\ (\lambda(l, r) \to \mathit{Right}\ (f\ l, f\ r))
\end{aligned}
$$

where the Haskell type of pairs (a, a) is denoted as $a \times a$, and the type constructor *Either* and the function *either* are given by the following standard Haskell definitions:

$$
\begin{aligned}
&\textbf{data}\ \mathit{Either}\ a\ b = \mathit{Left}\ a \mid \mathit{Right}\ b\\
&\mathit{either} :: (a \to c) \to (b \to c) \to \mathit{Either}\ a\ b \to c\\
&\mathit{either}\ f\ g\ (\mathit{Left}\ a) = f\ a\\
&\mathit{either}\ f\ g\ (\mathit{Right}\ b) = g\ b
\end{aligned}
$$

The functor definition can be generalized to functors with more than one argument.

Definition 2.2 (Bifunctor) A bifunctor F is a binary operator which applies to pairs of types and functions, and which satisfies the following properties:

- $F\ (f, g) : F\ (a, c)\ \rightarrow\ F\ (b, d)$, for all $f : a \rightarrow b$ and $g : c \rightarrow d$
- $F\ (id, id) = id$
- $F\ (f \circ g, h \circ k) = F\ (f, h) \circ F\ (g, k)$

In general, functors are specified in compact notation as compositions of elementary functors. Throughout this text, all types will be interpreted as complete partial orders with minimum (known as pointed CPOs [AJ94]).

- The identity functor: $I\ a = a$, $I\ f = f$
- The constant functor $\overline{c}$: $\overline{c}\ a = c$, $\overline{c}\ f = id$
- The product bifunctor:

$$(a \times b) = \{(x, y) \mid x \in a, y \in b\}$$
$$(f \times g)\ (x, y) = (f\ x, g\ y)$$

- The n-ary product:

$$(a_1 \times \cdots \times a_n) = \{(x_1, \ldots, x_n) \mid x_i \in a_i\}$$
$$(f_1 \times \cdots \times f_n)\ (x_1, \ldots, x_n) = (f_1\ x_1, \ldots, f_n\ x_n)$$

- The disjoint sum bifunctor:

$$a + b = (\{1\} \times a) \cup (\{2\} \times b) \cup \{\bot\}$$
$$(f + g)\ (1, x) = (1, f\ x)$$
$$(f + g)\ (2, y) = (2, g\ y)$$
$$(f + g)\ \bot = \bot$$

- The n-ary sum:

$$(a_1 + \cdots + a_n) = (\{1\} \times a_1) \cup \cdots \cup (\{n\} \times a_n) \cup \{\bot\}$$
$$(f_1 + \cdots + f_n)\ (i, x_i) = (i, f_i\ x_i)$$
$$(f_1 + \cdots + f_n)\ \bot = \bot$$

We assume from now on that application has greater precedence than both product and sum, and product has greater precedence than sum. When defining functors, we will also use the unit type $\mathbf{1} = \{(), \bot\}$.

Our sample functor can then be re-expressed more succinctly as

$$\begin{aligned} F\ a &= Int + a \times a \\ F\ f &= id + f \times f \end{aligned}$$

where we have replaced the *Either* type constructor by the sum functor.

We can still do better to shorten the definition of F.

- Product of functors

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

- Sum of functors

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

Now we can express our sample functor as $F = \overline{Int} + I \times I$. To see this, we evaluate F over a type and a function, and then unfold the product and sum of functors, which yields the same definition we had before:

$$\begin{aligned} F\ a &= (\overline{Int} + I \times I)\ a = \overline{Int}\ a + I\ a \times I\ a = Int + a \times a \\ F\ f &= (\overline{Int} + I \times I)\ f = \overline{Int}\ f + I\ f \times I\ f = id + f \times f \end{aligned}$$

We will call recursive positions of a functor those positions where functor I occurs.

2.2 Hylomorphisms

Now, we are able to write our recursive functions as hylomorphisms.

Definition 2.3 (Hylomorphism) Given a functor F, and functions $\psi :: a \to F\ a$ and $\phi :: F\ b \to b$, a hylomorphism is the least fixed point of the equation

$$h = \phi \circ F\ h \circ \psi$$

which is denoted as $[\![\phi, \psi]\!]_F :: a \to b$.

A hylomorphism has three components:

- a function ψ which takes the initial input and computes the arguments for the recursive calls;
- a functor F, which is used to perform the recursive calls on the output of ψ; and
- a function ϕ which takes the result of the recursive calls and any other value returned by ψ to build the final result.

So, a hylomorphism divides the recursion into the computations that are done before and after the recursive calls.

The representation of *sqrLeaves* and *sumBT* as hylomorphisms follows:

$$
\begin{aligned}
&F = \overline{Int} + I \times I \\
&sqrLeaves = [\![\phi, \psi]\!]_F \\
&\quad \textbf{where } \psi :: BTree \to F\ BTree \\
&\qquad\qquad \psi\ (Leaf\ i) = (1, sqr\ i) \\
&\qquad\qquad \psi\ (Join\ l\ r) = (2, (l, r)) \\
&\qquad\qquad \phi :: F\ BTree \to BTree \\
&\qquad\qquad \phi\ (1, i) = Leaf\ i \\
&\qquad\qquad \phi\ (2, (l, r)) = Join\ l\ r \\
&sumBT = [\![\phi', \psi']\!]_F \\
&\quad \textbf{where } \psi' :: BTree \to F\ BTree \\
&\qquad\qquad \psi'\ (Leaf\ i) = (1, i) \\
&\qquad\qquad \psi'\ (Join\ l\ r) = (2, (l, r)) \\
&\qquad\qquad \phi' :: F\ Int \to Int \\
&\qquad\qquad \phi'\ (1, i) = i \\
&\qquad\qquad \phi'\ (2, (l, r)) = l + r
\end{aligned}
$$

We have functions ψ and ψ' which prepare the arguments for the recursive calls. Then, funtor F specifies over which positions of the tuples returned by ψ and ψ' the recursive calls should be applied. Finally, we have functions ϕ and ϕ' that operate with the results of the calls.

In Section 2.5 we will present a law that tells us that

$$sumBT \circ sqrLeaves = [\![\phi', \psi]\!]_F$$

Note that the components ϕ and ψ, which are the ones which involve the constructors of the intermediate data structure are not part of the resulting hylo. Therein lays the cause of deforestation.

When we inline $[\![\phi', \psi]\!]_F$ (i.e. the inverse process of obtaining the hylomorphism), we get the following recursive definition.

$$
\begin{aligned}
&sumsqr :: BTree \to Int \\
&sumsqr\ (Leaf\ i) = sqr\ i \\
&sumsqr\ (Join\ l\ r) = sumsqr\ l + sumsqr\ r
\end{aligned}
$$

2.3 Algebras and coalgebras

We have shown in the previous section how a functor may be used to factorize a recursive definition into a hylomorphism. The factorization introduces new components ϕ, ϕ', ψ and ψ', which are sometimes substituted one for another when applying a fusion law. In this section we will state the terminology and some notation for working with those components.

The components applied to the result of the recursive calls are called algebras.

Definition 2.4 (F-algebra) Given a functor F and a type a, an F-algebra is any function with type $F\ a \to a$. The type a is said to be the carrier set of the F-algebra.

Note that in the previous section both ϕ and ϕ' are F-algebras with carrier sets *BTree* and *Int* respectively.

The following operator will be useful for expressing algebras.

Definition 2.5 (Case analysis) Case analysis for functions $f_i : a_i \to t$ is defined as

$$\begin{aligned} &f_1 \triangledown \cdots \triangledown f_n && :: a_1 + \cdots + a_n \to t \\ &(f_1 \triangledown \cdots \triangledown f_n)\ (i, x) &&= f_i\ x \end{aligned}$$

Case analysis comes with a useful property we will use later:

$$(f_1 \circ g_1) \triangledown \cdots \triangledown (f_n \circ g_n) = (f_1 \triangledown \cdots \triangledown f_n) \circ (g_1 + \cdots + g_n)$$

For example, the algebra ϕ of *sqrLeaves* can be expressed in terms of the case analysis as $\phi = \mathit{Leaf} \triangledown \mathit{Join}$, and ϕ' of *sumBT* can be expressed as $\phi' = id \triangledown (+)$. There is a slight type mismatch in these case analysis, since $\triangledown$ expects uncurried functions as arguments; however constructor *Join* and function $(+)$ are curried. With the aim to simplify notation as much as possible we will carry on with this kind of mismatch. We will also write algebras like $[\,] \triangledown (:)$, where the list constructor $[\,]$ does not receive any argument at all, understanding this occurrence of $[\,]$ or any other similar constructor as having type $\mathbf{1} \to [a]$, that is, a function taking $()$ and constructing the empty list $[\,]$.

The algebras expressed in terms of case analysis enumerate the operations they perform to produce values in the carrier set. For example, the algebra $\mathit{Leaf} \triangledown \mathit{Join}$ states that it performs a *Leaf* operation on an integer to create a *BTree*, and a *Join* operation on two *BTree*s to produce a new *Btree*. The algebra $id \triangledown (+)$ states that it performs operation id or $(+)$ to create integers. When a fusion law replaces an algebra by another one, it relies on the involved

functor to guarantee that the operations in the original algebra are replaced by operators with the same arity from the other.

Given an F-algebra of the form

$$\begin{array}{l}\phi_1 \triangledown \cdots \triangledown \phi_n \\ \quad \textbf{where } \phi_i \ v_{i1} \ \ldots \ v_{in_i} = t_i\end{array}$$

we will call recursive variables those variables v_i corresponding to recursive positions in functor F. Thus, for example, in the algebra $id \triangledown (+)$ from *sumBT*, the arguments to $(+)$ are recursive. This terminology matches the idea that those arguments hold the result of recursive calls.

We also need a concept for describing the functions that prepare the arguments of the recursive calls (ψ and ψ' in the case of *sqrLeaves* and *sumBT*).

Definition 2.6 (F-coalgebra) Given a functor F and a type a, an F-coalgebra is any function with type $a \rightarrow F \ a$. The type a is said to be the carrier set of the F-coalgebra.

Note that ψ and ψ' are both F-coalgebras with carrier set *BTree*.

The F-coalgebras we will manipulate are of the form

$$\lambda v_0 \rightarrow \textbf{case } t_0 \textbf{ of } p_1 \rightarrow (1, (t_{11}, \ldots, t_{1n_1})); \ldots ; p_m \rightarrow (m, (t_{m1}, \ldots, t_{mn_m}))$$

We will call t_{ij} a recursive term if it corresponds to a recursive position of functor F. This terminology matches the idea that a recursive t_{ij} is the argument of a recursive call.

2.4 Functors derived from types

Functors are related with data types. In fact, a functor may be derived from a data type definition by extracting the arity of its constructors. For instance, the functor $F = \overline{Int} + I \times I$ can be obtained by placing in a sum the signatures of the *BTree* constructors.

$$\textbf{data } \mathit{BTree} = \mathit{Leaf}\ \mathit{Int} \mid \mathit{Join}\ \mathit{BTree}\ \mathit{BTree}$$

Observe that occurrences of the identity functor I correspond to inductive arguments of the data type constructor.

In a similar way, for the type of lists,

$$\textbf{data } [a] = [\,] \mid a : [a]$$

we can derive a functor[1] $F = \overline{\mathbf{1}} + \overline{a} \times I$.

[1]Formally, when the type has a parameter a, the functor should be written as F_a, since the functor is parametric on it. However, we will avoid doing that for the sake of legibility.

Being F the functor derived from a data type T, we also call μF to the data type T. This is because the data type can be understood as the least fixed point of the type equation $x \cong F\ x$.

Having a functor F, we will call in_F the algebra $C_1 \triangledown \cdots \triangledown C_n :: F\ \mu F \to \mu F$, being $C_1, \ldots, C_n$ the constructors of data type μF. We also call $out_F :: \mu F \to F\ \mu F$ the inverse of in_F. Being $F = \overline{Int} + I \times I$, we have that in_F can be expressed as $in_F = Leaf \triangledown Join$ and out_F can be defined as:

$$out_F\ (Leaf\ i) = (1, i)$$
$$out_F\ (Join\ l\ r) = (2, (l, r))$$

The algebra in_F and the coalgebra out_F allow to specialize hylomorphisms to get the well known recursive operators **fold** ($(\!|-|\!)_F$) [BdM97] and **unfold** ($[\!(-)\!]_F$) [GJ98].

$$(\!|\phi|\!)_F = [\![\phi, out_F]\!]_F$$
$$[\!(\psi)\!]_F = [\![in_F, \psi]\!]_F$$

These operators are also known as catamorphisms and anamorphisms respectively [MFP91].

2.5 Acid Rain laws

We have gathered now all the background we need to introduce the fusion laws that HFusion uses [TM95, OHIT97].

Theorem 2.7 (Acid rain)

fold-unfold:
$$[\![\phi, out_F]\!]_F \circ [\![in_F, \psi]\!]_F = [\![\phi, \psi]\!]_F$$

fold-hylo:
$$\frac{(\!|\phi|\!)_F \text{ is strict} \qquad \tau :: \forall a.\ (F\ a \to a) \to (G\ a \to a)}{(\!|\phi|\!)_F \circ [\![\tau(in_F), \psi]\!]_G = [\![\tau(\phi), \psi]\!]_G}$$

hylo-unfold:
$$\frac{\sigma :: \forall a.\ (a \to F\ a) \to (a \to G\ a)}{[\![\phi, \sigma(out_F)]\!]_G \circ [\!(\psi)\!]_F = [\![\phi, \sigma(\psi)]\!]_G}$$

Each of the laws allows to replace a composition by an equivalent hylomorphism, and depends in doing so on the capacity of finding the uses of out_F at the left of the composition, and the uses of in_F at the right of it. The hylomorphism that is used to replace the composition never contains in_F nor out_F, which constitutes the elimination of the intermediate data structure.

The fold-unfold law is the one we applied in the example of *sumsqr*. Fold-hylo fusion requires identifying uses of in_F inside a term described by τ, while

hylo-unfold fusion requires identifying uses of out_F inside a term described by σ.

Functions τ and σ are called **transformers** of algebras and coalgebras, respectively [Fok92].

A drawback of acid rain laws is that they are known to work over Haskell programs without the *seq* operator [Jon03]. If *seq* occurs in the input programs, the laws may need to be weakened [JV04], since their proof depends on free theorems [TM95]. So we assume from now on that none of our input programs contain the *seq* operator.

Example 2.8 (fold-hylo fusion) Consider the following program, which counts the amount of elements in a list satisfying a given predicate.

$$
\begin{aligned}
&\mathit{lf} :: (a \to \mathit{Bool}) \to [\,a\,] \to \mathit{Int}\\
&\mathit{lf}\ p = \mathit{length} \circ \mathit{filter}\ p
\end{aligned}
$$

We will write the definitions used in this program as hylomorphisms.

$$
\begin{aligned}
&\mathit{filter} :: (a \to \mathit{Bool}) \to [\,a\,] \to [\,a\,]\\
&\mathit{filter}\ p = [\![\,[\,]\triangledown\phi_2, \psi]\!]_G\\
&\quad \textbf{where}\ G = \overline{\mathbf{1}} + \overline{a} \times I \times I\\
&\qquad\quad \phi_2\ (a, v_1, v_2) = \textbf{if}\ p\ a\ \textbf{then}\ a : v_1\ \textbf{else}\ v_2\\
&\qquad\quad \psi\ [\,] = (1, ())\\
&\qquad\quad \psi\ (a : ls) = (2, (a, ls, ls))\\
&\mathit{length} :: [\,a\,] \to \mathit{Int}\\
&\mathit{length} = (\!|\,0\triangledown(\lambda(\_, v) \to 1 + v)\,|\!)_F\\
&\quad \textbf{where}\ F = \overline{\mathbf{1}} + \overline{a} \times I
\end{aligned}
$$

Now, we rewrite the algebra of *filter* as $\tau(in_F)$.

$$
\begin{aligned}
&\mathit{filter}\ p = [\![\,\tau(in_F), \psi]\!]_G\\
&\quad \textbf{where}\ \tau :: \forall a.\ (F\ a \to a) \to (G\ a \to a)\\
&\qquad\quad \tau(\alpha) = \tau_1(\alpha)\triangledown\tau_2(\alpha)\\
&\qquad\quad \tau_1(\alpha_1\triangledown\alpha_2) = \alpha_1\\
&\qquad\quad \tau_2(\alpha_1\triangledown\alpha_2)\ (a, v_1, v_2) = \textbf{if}\ p\ a\ \textbf{then}\ \alpha_2\ (a, v_1)\\
&\qquad\qquad\qquad\qquad\qquad\qquad\ \ \textbf{else}\ v_2
\end{aligned}
$$

Applying fold-hylo fusion we get

$$
\begin{aligned}
&\mathit{lf} :: (a \to \mathit{Bool}) \to [\,a\,] \to \mathit{Int}\\
&\mathit{lf}\ p = [\![\,\tau(0\triangledown(\lambda(\_, v) \to 1 + v)), \psi]\!]_G
\end{aligned}
$$

Inlining the above we get:

$$
\begin{aligned}
&\mathit{lf} :: (a \to \mathit{Bool}) \to [\,a\,] \to \mathit{Int}\\
&\mathit{lf}\ p\ [\,] = 0\\
&\mathit{lf}\ p\ (a : ls) = \textbf{if}\ p\ a\ \textbf{then}\ 1 + \mathit{lf}\ ls\ \textbf{else}\ \mathit{lf}\ ls
\end{aligned}
$$

□

Example 2.9 (hylo-unfold fusion) Let us now consider the following definitions.

$$
\begin{aligned}
&\mathit{odds} :: [a] \to [a] \\
&\mathit{odds}\ (a : \_ : as) = a : \mathit{odds}\ as \\
&\mathit{odds}\ (a : [\,]) = [a] \\
&\mathit{odds}\ [\,] = [\,] \\
&\mathit{upto} :: \mathit{Int} \to \mathit{Int} \to \mathit{List}\ \mathit{Int} \\
&\mathit{upto}\ n\ m = \mathbf{if}\ m > n\ \mathbf{then}\ [\,] \\
&\qquad\qquad\qquad \mathbf{else}\ m : \mathit{upto}\ n\ (m+1)
\end{aligned}
$$

and suppose we want to fuse the composition $\mathit{odds}\ (\mathit{upto}\ n\ 1)$ which computes the odd numbers between 1 and n.

First, we factorize the definitions as hylomorphisms:

$$
\begin{aligned}
&\mathit{odds} = [\![[\,] \triangledown (:) \triangledown (:[\,]), \psi]\!]_G \\
&\quad \mathbf{where}\ G = \overline{\mathbf{1}} + \overline{a} \times I + \overline{a} \\
&\qquad\qquad \psi\ (a : \_ : as) = (2, (a, as)) \\
&\qquad\qquad \psi\ (a : [\,]) = (3, a) \\
&\qquad\qquad \psi\ [\,] = (1, ()) \\
&\mathit{upto}\ n = [\![\psi']\!]_F \\
&\quad \mathbf{where}\ F = \overline{\mathbf{1}} + \overline{\mathit{Int}} \times I \\
&\qquad\qquad \psi'\ m = \mathbf{if}\ m > n\ \mathbf{then}\ (1, ()) \\
&\qquad\qquad\qquad\qquad\ \mathbf{else}\ (2, (m, m+1))
\end{aligned}
$$

Now we write ψ as $\sigma(\mathit{out}_F)$.

$$
\begin{aligned}
&\mathit{odds} = [\![[\,] \triangledown (:) \triangledown (:[\,]), \sigma(\mathit{out}_F)]\!]_G \\
&\quad \mathbf{where}\ \sigma :: (a \to F\ a) \to (a \to G\ a) \\
&\qquad\qquad \sigma(\beta)\ l = \\
&\qquad\qquad\quad \mathbf{case}\ \beta\ l\ \mathbf{of} \\
&\qquad\qquad\qquad (1, ()) \to (1, ()) \\
&\qquad\qquad\qquad (2, (i, is)) \to \mathbf{case}\ \beta\ is\ \mathbf{of} \\
&\qquad\qquad\qquad\qquad\qquad\qquad (1, ()) \to (3, i) \\
&\qquad\qquad\qquad\qquad\qquad\qquad (2, (\_, is')) \to (2, (i, is'))
\end{aligned}
$$

Applying hylo-unfold fusion to $\mathit{odds}\ (\mathit{upto}\ n\ 1)$ we get:

$$\mathit{odds\_upto}\ n = [\![[\,] \triangledown (:) \triangledown (:[\,]), \sigma(\psi')]\!]_G$$

Inlining the above hylomorphism yields

$$
\begin{aligned}
&\mathit{odds\_upto} :: \mathit{Int} \to \mathit{Int} \to [\mathit{Int}] \\
&\mathit{odds\_upto}\ n\ m = \\
&\quad \mathbf{if}\ m > n\ \mathbf{then}\ [\,] \\
&\qquad \mathbf{else\ if}\ m + 1 > n\ \mathbf{then}\ [m] \\
&\qquad\qquad \mathbf{else}\ m : \mathit{odds\_upto}\ n\ (m + 1 + 1)
\end{aligned}
$$

□

2.6 Natural transformations

Sometimes, we will need to restructure a hylomorphism before we can apply a fusion law to it. Natural transformations will serve to that purpose.

Definition 2.10 (Natural transformation) A natural transformation η between two functors F and G, denoted as $\eta : F \Rightarrow G$, is a polymorphic function $\eta :: F\ a \to G\ a$.

The concept of natural transformation is more general than the one of polymorphic function, but in our context it is enough to see both concepts as the same.

Every natural transformation comes with a *free theorem* [Wad89] derived from its polymorphic type. For all $f :: a \to b$:

$$\eta \circ F\ f = G\ f \circ \eta$$

The trivial natural transformation is $id : I \Rightarrow I$. Another natural transformation is

$$\begin{aligned} &swap :: a \times Int \to Int \times a \\ &swap\ (a, i) = (i, a) \end{aligned}$$

which can be expressed as $swap : I \times \overline{Int} \Rightarrow \overline{Int} \times I$.

Natural transformations can be composed to form other natural transformations. Given natural transformations $\eta : F \Rightarrow G$ and $\eta' : F' \Rightarrow G'$:

- $\eta \times \eta' : F \times F' \Rightarrow G \times G'$
- $\eta + \eta' : F + F' \Rightarrow G + G'$
- If $G = F'$, $\eta \circ \eta' : F \Rightarrow G'$

The following is an interesting property relating natural transformations and hylomorphisms.

Proposition 2.11 (hylo-shift)

$$\frac{\eta : F \Rightarrow G}{[\![\phi \circ \eta, \psi]\!]_F = [\![\phi, \eta \circ \psi]\!]_G}$$

This property allows to move terms from the algebra to the coalgebra of a hylomorphism.

Example 2.12 (Applying hylo-shift) If we consider again the hylomorphism for *sqrLeaves*,

$$
\begin{aligned}
&\mathit{sqrLeaves} = [\![\psi]\!]_F \\
&\quad \textbf{where } F = \overline{\mathit{Int}} + I \times I \\
&\qquad\quad \psi :: \mathit{BTree} \to F\ \mathit{BTree} \\
&\qquad\quad \psi\ (\mathit{Leaf}\ i) = (1, \mathit{sqr}\ i) \\
&\qquad\quad \psi\ (\mathit{Join}\ l\ r) = (2, (l, r))
\end{aligned}
$$

we can rewrite is as

$$
\begin{aligned}
&\mathit{sqrLeaves} = [\![(\mathit{sqr} + id) \circ \mathit{out}_F]\!]_F \\
&\quad \textbf{where } \mathit{sqr} + id :: F \Rightarrow F
\end{aligned}
$$

where we can see the function $\mathit{sqr} :: \mathit{Int} \to \mathit{Int}$ as a natural transformation $\mathit{sqr} : \overline{\mathit{Int}} \Rightarrow \overline{\mathit{Int}}$ between the constant functor $\overline{\mathit{Int}}$ and itself. In fact, every non-polymorphic function can be seen as a natural transformation by using constant functors.

We can then apply hylo-shift to get $\mathit{sqrLeaves} = (\!|\mathit{in}_F \circ (\mathit{sqr} + id)|\!)_F$. This restructure is useful in case we want to fuse compositions of the form $\mathit{sqrLeaves} \circ [\![\tau(\mathit{in}_F), \psi]\!]_G$ or $\mathit{sqrLeaves} \circ [\![\psi]\!]_F$, because it makes it possible to see *sqrLeaves* as a fold. □

Chapter 3

Basic algorithms

In this chapter we present the most elemental algorithms that are built into the HFusion tool. In the following chapters we will extend both the theory and the algorithms to incorporate more elaborated cases. Many of these algorithms have been already implemented as part of the HYLO system [OHIT97, Sch00], but there are a few that have been developed by us.

From an external viewpoint, the tool takes a list of Haskell definitions, fuses some of them and outputs the result as another list of Haskell definitions. In the following we discuss the different stages of that process.

1. **Build internal representation for recursive definitions.** First, the Haskell definitions are written as hylomorphisms. Here we will use the algorithm presented in Section 3.1.

2. **Classify hylomorphisms according to the cases of the acid rain laws.** It is necessary to determine if the hylomorphisms satisfy any of the hypotheses of the acid rain laws. In this stage we will be answering questions like: Is a hylomorphism a fold? Is it an unfold? Is the algebra in $\tau(in_F)$ form? Is the coalgebra in $\sigma(out_F)$ form? Is it possible to transform the hylomorphism to make it fit any of the hypotheses? The algorithms for answering these questions will be presented in Section 3.2.

3. **Apply fusion laws.** In this stage we take compositions of hylomorphisms and apply the fusion laws to them. The tool expects to be supplied with the compositions that a user wants to have fused. If the tool were to be integrated into a compiler, it would be necessary a procedure that searches compositions in a program and feeds them automatically to this stage. Implementation of such a procedure has been moved to future work.

4. **Inlining.** The fusion laws return hylomorphisms that we want to write in a way a compiler could understand. Therefore, in this stage the

$$
\begin{array}{lll}
\multicolumn{3}{l}{program ::= \; v{=}t} \\
 & \vdots & \\
 & v{=}t & \\
 & & \\
b ::= & v & \text{(variables)} \\
\mid & (b, \ldots, b) & \text{(tuples of variables)} \\
 & & \\
p ::= & v & \text{(variables)} \\
\mid & (p, \ldots, p) & \text{(tuples of patterns)} \\
\mid & c\ p \ldots p & \text{(constructor application)} \\
\mid & l & \text{(literals)}
\end{array}
\qquad
\begin{array}{lll}
t ::= & v & \text{(variables)} \\
\mid & l & \text{(literals)} \\
\mid & (t, \ldots, t) & \text{(tuples)} \\
\mid & \lambda b \rightarrow t & \text{(lambda expressions)} \\
\mid & let\ v{=}t\ in\ t & \text{(let expressions)} \\
\mid & case\ t\ of & \text{(case expressions)} \\
 & \quad p \rightarrow t & \\
 & \quad \vdots & \\
 & \quad p \rightarrow t & \\
\mid & v\ t \ldots t & \text{(function application)} \\
\mid & c\ t \ldots t & \text{(constructor application)} \\
\mid & t\ t & \text{(term application)} \\
\mid & \bot & \text{(undefined term)}
\end{array}
$$

Figure 3.1: Core language grammar

resulting hylomorphisms are inlined obtaining recursive Haskell definitions.

3.1 Hylomorphism derivation

Before manipulating function definitions, we give a description of the grammar for our core language (Figure 3.1). Our core language is a subset of Haskell, but big enough to express all definitions of interest. Types are not specified as they are not needed for our algorithms, though all of them depend on the input programs being well typed. In the grammar, function and constructor applications are separated from general term applications for practical reasons. All applications appear in curried form, but we will treat curried and uncurried forms indistinctly throughout the algorithms and examples.

In Figure 3.2 we present the derivation algorithm for hylomorphisms, which has been proposed by Onoue et al. [OHIT97]. The algorithm accepts definitions in the following form:

$$f = \lambda v_1\ \ldots\ v_m \rightarrow \mathbf{case}\ t_0\ \mathbf{of}\ p_1 \rightarrow t_1; \ldots; p_n \rightarrow t_n$$

It is required that f be well typed and recursive over one argument only. Without loss of generality we will assume that the recursive argument is the last one (v_m); this means that the rest of the arguments $v_1, \ldots, v_{m-1}$ remain constant throughout the recursive calls.

The algorithm will then return a term of the form:

$$f = \lambda v_1\ \ldots\ v_{m-1} \rightarrow [\![\phi_1 \triangledown \cdots \triangledown \phi_n, \psi]\!].$$

$$
\begin{array}{l}
\mathcal{H}(f, \lambda v_1\ \ldots\ v_m \to \textbf{case}\ t_0\ \textbf{of}\ p_1 \to t_1; \ldots; p_n \to t_n) = \\
\quad \lambda v_1\ \ldots\ v_{m-1} \to [\![\phi_1 \triangledown \cdots \triangledown \phi_n, \psi]\!]_F \\
\quad \textbf{where} \\
\qquad \psi\ = \lambda v_m \to \textbf{case}\ t_0\ \textbf{of} \\
\qquad\qquad\qquad p_1 \to (1, (v_{11}, \ldots, v_{1l_1}, t'_{11}, \ldots, t'_{1s_1})) \\
\qquad\qquad\qquad\quad \vdots \\
\qquad\qquad\qquad p_n \to (n, (v_{n1}, \ldots, v_{nl_n}, t'_{n1}, \ldots, t'_{ns_n})) \\
\qquad \phi_i\ = \lambda(v_{i1}, \ldots, v_{il_i}, u_1, \ldots, u_{s_i}) \to t''_i \\
\qquad F\ = F_1 + \cdots + F_n \\
\qquad F_i = \overline{\Gamma(\mathtt{v_{i1}})} \times \cdots \times \overline{\Gamma(\mathtt{v_{i1_i}})} \times I_1 \times \cdots \times I_{s_i} \quad \text{-- } \Gamma(v) \text{ returns the type of } v \\
\qquad (\{v_{i1}, \ldots, v_{il_i}\}, \{(u_1, t_{i1}), \ldots, (u_{s_i}, t_{is_i})\}, t''_i) = \mathcal{D}\ (p_i, t_i) \\
\qquad \mathcal{D}\ (p_i, v) = \quad (\{v\}, \emptyset, v) \text{ if } v \in vars(p_i) \cup \{v_m\} \\
\qquad\qquad\qquad\quad\ (\emptyset, \emptyset, v) \quad \text{otherwise} \\
\qquad \mathcal{D}\ (p_i, (t_1, \ldots, t_n)) = (c_1 \cup \cdots \cup c_n, c'_1 \cup \cdots \cup c'_n, (t'_1, \ldots, t'_n)) \\
\qquad\quad \textbf{where}\ (c_i, c'_i, t'_i) = \mathcal{D}\ (p_i, t_i) \\
\qquad \mathcal{D}\ (p_i, C_j\ (t_1, \ldots, t_n)) = (c_1 \cup \cdots \cup c_n, c'_1 \cup \cdots \cup c'_n, C_j\ (t'_1, \ldots, t_n)) \\
\qquad\quad \textbf{where}\ (c_i, c'_i, t'_i) = \mathcal{D}\ (p_i, t_i) \\
\qquad \mathcal{D}\ (p_i, g\ t_1\ \ldots\ t_m)) = (\emptyset, \{(u, t_m)\}, u) \text{ if } g = f \text{ and } v_i = t_i \text{ for all } i < m \\
\qquad\qquad\qquad\qquad\qquad\quad (c_1 \cup \cdots \cup c_n, c'_1 \cup \cdots \cup c'_n, g\ t'_1\ \ldots\ t'_n) \text{ otherwise} \\
\qquad\quad \textbf{where}\ (c_i, c'_i, t'_i) = \mathcal{D}\ (p_i, t_i) \\
\qquad\qquad\ u \text{ is a fresh variable} \\
\qquad \mathcal{D}\ (p_i, \textbf{let}\ v = t_0\ \textbf{in}\ t_1) = (c_0 \cup c_1, c'_0 \cup c'_1, \textbf{let}\ v = t'_0\ \textbf{in}\ t'_1) \\
\qquad\quad \textbf{where}\ (c_i, c'_i, t'_i) = \mathcal{D}\ (p_i, t_i) \\
\qquad \mathcal{D}\ (p_i, \lambda v \to t) = (c, c', \lambda v \to t') \\
\qquad\quad \textbf{where}\ (c, c', t') = \mathcal{D}\ (p_i, t) \\
\qquad \mathcal{D}\ (p_i, \textbf{case}\ t_0\ \textbf{in}\ p_1 \to t_1; \ldots; p_n \to t_n) = \\
\qquad\quad (c_1 \cup \cdots \cup c_n, c'_1 \cup \cdots \cup c_n, \textbf{case}\ t_0\ \textbf{of}\ p_1 \to t'_1; \ldots; p_n \to t'_n) \\
\qquad\quad \textbf{where}\ (c_i, c'_i, t'_i) = \mathcal{D}\ (p_i, t_i)
\end{array}
$$

Figure 3.2: Hylomorphism derivation algorithm

Each component ϕ_i is constructed from a term t_i. In t_i the recursive calls are replaced by fresh variables representing the result of these recursive calls. Algorithm $\mathcal{D}$ is the one abstracting the recursive calls. The coalgebra contains the **case** over t_0. For each **case** alternative, the coalgebra returns the arguments of the recursive calls and the bounded variables in the corresponding pattern p_i, as long as they are referenced by the term t_i.

There is a technical detail we are not caring of in the algorithm. We are assuming that none of the input variables $v_1 \ldots v_{m-1}$ is captured by the terms t_i. This means that no *case* pattern, *let* declaration and lambda abstraction introduces them again. Otherwise, algorithm $\mathcal{D}$ would need to do more careful checks to verify that the assumed constant arguments effectively remain constant throughout the recursive calls.

3.2 Analyzing data production and consumption schemes

Applying the acid rain laws requires analysis of the algebra and coalgebra of the hylomorphisms. That is, when having a composition

$$[\![\phi, \psi]\!]_F \circ [\![\phi', \psi']\!]_G$$

we want to determine the shape of ψ and ϕ' in order to know which acid rain law can be applied.

In this section we will be answering the following questions:

- $\phi' = in_G$?
- $\psi = out_F$?
- $\phi' = \tau(in_F)$?
- $\psi = \sigma(out_G)$?

The analysis of data production and data consumption schemes may imply restructuring the hylomorphisms so that the algebra or the coalgebra can be shaped in a convenient way. The essence of restructuring is finding a decomposition of an algebra ϕ' into $\phi'' \circ \eta$, where η is a natural transformation $\eta : F \Rightarrow G$. Then, we can apply the *hylo-shift* property (Proposition 2.11) to get that

$$[\![\phi', \psi']\!]_F = [\![\phi'', \eta \circ \psi]\!]_G$$

where ϕ'' is hopefully simpler than ϕ' to continue the analysis.

In a dual way, we could decompose a coalgebra ψ as $\eta \circ \psi''$, where η is again a natural transformation $\eta : G \Rightarrow F$. Then, by *hylo-shift* we get that

$$[\![\phi, \psi]\!]_F = [\![\phi \circ \eta, \psi'']\!]_G$$

where ψ'' is possibly easier to analyze than ψ.

3.2.1 Recognizing in_F

We will say that an algebra $\phi' : G\ a \to a$ is in_G if it is of the form $C_1 \triangledown \cdots \triangledown C_n$ where $C_1, \ldots, C_n$ are all the constructors of type a.

Of course, there are many terms that can be equivalent to an algebra in that form. However, for practical reasons we will stick to this syntactic criterion, and will not consider the other equivalent algebras as being in_G. In essence, this has been the approach taken by Schwartz[Sch00]. Onoue et al.[OHIT97] propose applying ϕ' and in_G to a given set of values in order to verify that the results they deliver match. However, they do not provide any details on how such a method would ensure equivalence of the algebras, or whether it may improve recognition in practice with respect to using the syntactic approach.

When an algebra is not in in_G form, there are some restructures that can be tried in order to get a hylomorphism with a complying algebra. In what

follows, we present a survey of the restructuring techniques that the HFusion tool applies for that purpose. The first one is a synthesis of the restructuring proposed by Onoue et al. [OHIT97] and Schwartz [Sch00], the rest of the restructures were developed by us.

Moving arguments

Given an algebra $\phi = \phi_1 \triangledown \cdots \triangledown \phi_n$ where each ϕ_i is of the form

$$\phi_i = \lambda(v_{i1}, \ldots, v_{ik_i}) \rightarrow C_i\ (t_{i1}, \ldots, t_{il_i})$$

we can restructure each ϕ_i as $C_i \circ \eta_i$ where

$$\eta_i = \lambda(v_{i1}, \ldots, v_{ik_i}) \rightarrow (t_{i1}, \ldots, t_{il_i})$$

Thus, we can restructure algebra ϕ as:

$$\phi = (C_1 \circ \eta_1) \triangledown \cdots \triangledown (C_n \circ \eta_n) = (C_1 \triangledown \cdots \triangledown C_n) \circ (\eta_1 + \cdots + \eta_n)$$

If each η_i is a natural transformation, then so is $(\eta_1 + \cdots + \eta_n)$ and therefore can be moved to the coalgebra. To guarantee that, the restriction is imposed that each term t_{ij} must be a variable or it must be a term that does not make reference to recursive variables.

Example 3.1 (Restructure of *mirror*) Consider the following definition.

$$\begin{aligned} &\mathit{mirror} :: \mathit{BTree} \rightarrow \mathit{BTree} \\ &\mathit{mirror}\ (\mathit{Join}\ l\ r) = \mathit{Join}\ (\mathit{mirror}\ r)\ (\mathit{mirror}\ l) \\ &\mathit{mirror}\ (\mathit{Leaf}\ i) = \mathit{Leaf}\ i \end{aligned}$$

If we derive a hylomorphism from it, we get:

$$\begin{aligned} &\mathit{mirror} = (\!|\phi|\!)_F \\ &\quad \textbf{where}\ F = \overline{\mathit{Int}} + I \times I \\ &\qquad\quad \phi :: F\ \mathit{BTree} \rightarrow \mathit{BTree} \\ &\qquad\quad \phi = \mathit{Leaf} \triangledown (\lambda(v_1, v_2) \rightarrow \mathit{Join}\ v_2\ v_1) \end{aligned}$$

We can decompose the algebra as follows

$$\phi = (\mathit{Leaf} \triangledown \mathit{Join}) \circ (id + \mathit{swap})$$

where $\mathit{swap}\ (v_1, v_2) = (v_2, v_1)$. Since $(id + \mathit{swap}) : F \Rightarrow F$ is a natural transformation, we can restructure the hylomorphism as

$$(\!|\phi|\!)_F = (\!|in_F \circ (id + \mathit{swap})|\!)_F = [\!(id + \mathit{swap}) \circ \mathit{out}_F]\!]_F$$

□

Example 3.2 (Restructure of *map*) Consider now the following definition.

$$
\begin{aligned}
&map : (a \to b) \to [\,a\,] \to [\,b\,] \\
&map\ f\ (a : as) = f\ a : map\ as \\
&map\ f\ [\,] = [\,]
\end{aligned}
$$

as hylomorphism

$$
\begin{aligned}
&map\ f = [\![\phi, out_F]\!]_F \\
&\textbf{where } F = \overline{\mathbf{1}} + \overline{a} \times I \\
&\qquad \phi :: F\ [\,a\,] \to [\,a\,] \\
&\qquad \phi = [\,]\triangledown(\lambda(a, v) \to f\ a : v) \\
&\qquad out_F = \lambda l \to \textbf{case } l \textbf{ of} \\
&\qquad\qquad\qquad [\,] \to (1, ()) \\
&\qquad\qquad\qquad (i : is) \to (2, (i, is))
\end{aligned}
$$

The algebra can be decomposed as

$$\phi = [\,]\triangledown(:) \circ (id + (\lambda(a, v) \to (f\ a, v)))$$

where $(id + (\lambda(a, v) \to (fa, v))) : F \Rightarrow G$ is a natural transformation, with $G = \overline{\mathbf{1}} + \overline{b} \times I$. Restructuring and applying *hylo-shift*:

$$[\![\phi, out_F]\!]_F = [\![(id + (\lambda(a, v) \to (f\ a, v))) \circ out_F]\!]_G$$

□

Adding missing constructors

Let us suppose we have a functor $F = F_1 + \cdots + F_k + \cdots + F_n$, a data type μF with constructors $C_1, \ldots, C_k, \ldots, C_n$, a functor $F' = F_1 + \cdots + F_k$, and the algebra

$$\phi = (C_1\triangledown \cdots \triangledown\, C_k) :: F'\ \mu F \to \mu F$$

where there are constructors of data type μF which do not appear in ϕ. We can restructure the algebra as

$$\phi = (C_1\triangledown \cdots \triangledown\, C_k\triangledown \cdots \triangledown\, C_n) \circ g$$

where g is defined as

$$
\begin{aligned}
&g :: F'\ a \to F\ a \\
&g = (\lambda vs \to (1, vs))\triangledown \cdots \triangledown(\lambda vs \to (k, vs))
\end{aligned}
$$

Here g plays the role of a conversion function from type $F'\ a$ to type $F\ a$ which is a superset of the former. The conversion can be thought of as an application of *constructor subtyping* [BFaF99]. Note that g is a natural transformation $g : F' \Rightarrow F$.

Example 3.3 (Restructure of *repeat*) Consider the following definition.

$$
\begin{aligned}
&\mathit{repeat} : a \to [\,a\,] \\
&\mathit{repeat}\ a = a : \mathit{repeat}\ a
\end{aligned}
$$

It can be represented as the hylomorphism

$$
\begin{aligned}
&\mathit{repeat} = [\![(:), \psi]\!]_F \\
&\quad \textbf{where}\ F = \overline{a} \times I \\
&\qquad\quad \psi = \lambda a \to (a, a)
\end{aligned}
$$

whose algebra we can decompose as

$$\phi = ((:) \triangledown [\,]) \circ (\lambda vs \to (1, vs))$$

being $(\lambda vs \to (1, vs)) : \overline{\mathsf{a}} \times I \Rightarrow \overline{\mathsf{a}} \times I + \overline{\mathbf{1}}$ a natural transformation. Restructuring we get

$$
\begin{aligned}
&[\![\phi, \psi]\!]_F = [\![in_G, (\lambda vs \to (1, vs)) \circ \psi]\!]_G \\
&\quad \textbf{where}\ G = \overline{a} \times I + \overline{\mathbf{1}} \\
&\qquad\quad in_G = (:) \triangledown [\,]
\end{aligned}
$$

□

Moving cases

Given a term ϕ_i as follows

$$
\begin{aligned}
\phi_i = \lambda(v_1, \ldots, v_k) \to \textbf{case}\ t_0\ \textbf{of} \\
p_1 \ &\to t_1 \\
&\vdots \\
p_m &\to t_m
\end{aligned}
$$

we can decompose it as

$$
\begin{aligned}
&\phi_i = (g_1 \triangledown \cdots \triangledown g_m) \circ \eta \\
&\quad \textbf{where}\ \eta = \lambda(v_1, \ldots, v_k) \to \textbf{case}\ t_0\ \textbf{of} \\
&\qquad\qquad\qquad\qquad p_1 \ \to (1, bv(p_1, t_1)) \\
&\qquad\qquad\qquad\qquad\quad \vdots \\
&\qquad\qquad\qquad\qquad p_m \to (m, bv(p_m, t_m)) \\
&\qquad\quad g_i = \lambda bv(p_i, t_i) \to t_i
\end{aligned}
$$

where $bv(p_i, t_i)$ are the variables of t_i bound in pattern p_i or $v_1, \ldots, v_k$.

If ϕ_i is an algebra component, then η is a natural transformation if t_0 does not reference recursive variables.

With this manipulation we can restructure an algebra $\phi_1 \triangledown \cdots \triangledown \phi_n$ as:

$$\phi_1 \triangledown \cdots \triangledown \phi_n = (\phi'_1 \triangledown \cdots \triangledown \phi'_n) \circ (\eta_1 + \cdots + \eta_n)$$

where:

- η_i contains a **case** structure if ϕ_i contains one. The corresponding term ϕ'_i is then of the form $g_{i_1} \triangledown \cdots \triangledown g_{i_{m_i}}$.
- $\eta_i = id$ if ϕ_i does not contain a **case**; then $\phi_i = \phi'_i$.

If ϕ_i has nested **case** structures then ϕ'_i may be factored again.

Example 3.4 (reestructure of *upto*) Consider the following definition

$$\begin{aligned}
&\mathit{upto} : \mathit{Int} \to \mathit{Int} \to [\mathit{Int}] \\
&\mathit{upto}\ n\ m = \mathbf{case}\ m > n\ \mathbf{of} \\
&\qquad\qquad \mathit{True} \to [\,] \\
&\qquad\qquad \mathit{False} \to m : \mathit{upto}\ n\ (m+1)
\end{aligned}$$

Note that we are interpreting an **if** structure as a **case** for the sake of this restructure. We get the following hylomorphism for *upto*:

$$\begin{aligned}
&\mathit{upto}\ n = [\![\phi, \lambda m \to (m, m+1)]\!]_F \\
&\quad \mathbf{where}\ F = \overline{\mathit{Int}} \times I \\
&\qquad\quad \phi :: \mathit{Int} \times [\mathit{Int}] \to [\mathit{Int}] \\
&\qquad\quad \phi = \lambda(m, v) \to \mathbf{case}\ m > n\ \mathbf{of} \\
&\qquad\qquad\qquad\qquad \mathit{True} \to [\,] \\
&\qquad\qquad\qquad\qquad \mathit{False} \to m : v
\end{aligned}$$

The algebra can be decomposed as

$$\begin{aligned}
&\phi = ([\,] \triangledown (:)) \circ \eta \\
&\quad \mathbf{where}\ G = \overline{\mathbf{1}} + \overline{\mathit{Int}} \times I \\
&\qquad\quad \eta :: F\ a \to G\ a \\
&\qquad\quad \eta = \lambda(m, v) \to \mathbf{case}\ m > n\ \mathbf{of} \\
&\qquad\qquad\qquad\qquad \mathit{True} \to (1, ()) \\
&\qquad\qquad\qquad\qquad \mathit{False} \to (2, (m, v))
\end{aligned}$$

Restructuring and applying *hylo-shift* we get

$$\begin{aligned}
[\![\phi, \lambda m \to (m, m+1)]\!]_F &= [\![in_G \circ \eta, \lambda m \to (m, m+1)]\!]_F \\
&= [\![\eta \circ (\lambda m \to (m, m+1))]\!]_G
\end{aligned}$$

□

3.2.2 Recognizing out_F

Having a coalgebra ψ we would like to be able to determine if it is out_F. Consider the following scheme for coalgebra ψ:

$$\begin{aligned}
&\psi = \lambda l \to \mathbf{case}\ t_0\ \mathbf{of} \\
&\qquad\qquad p_1 \to (1, (t_{11}, \ldots, t_{1k_1}))
\end{aligned}$$

$$\begin{array}{l} \vdots \\ p_n \rightarrow (n, (t_{n1}, \ldots, t_{nk_n})) \end{array}$$

Similar to in_F, we define a syntactic form for ψ to consider it out_F.

1. Each pattern p_i must be a constructor application over variables only.
2. The variables in p_i must occur in the corresponding output tuple, in the same order they appear in p_i. No other value shall be returned.
3. There is one and only one pattern for each constructor of the input type.

As in the case of in_F, there are many terms equivalent to out_F not fitting our scheme. We will regard those terms as needing restructures, and if the restructuring does not work then we will give up. This is the same approach adopted in [Sch00]. Next, we present our coalgebra restructures. All of them are implemented in HFusion. The first of them is a synthesis of the restructuring proposed by Onoue et al.[OHIT97]. The second one has been developed by us.

Moving terms

Given a coalgebra

$$\begin{array}{l} \psi = \lambda l \rightarrow \textbf{case } t_0 \textbf{ of} \\ \qquad\qquad p_1 \rightarrow (1, (t_{11}, \ldots, t_{1k_1})) \\ \qquad\qquad \vdots \\ \qquad\qquad p_n \rightarrow (n, (t_{n1}, \ldots, t_{nk_n})) \end{array}$$

we can restructure it as follows

$$\begin{array}{l} \psi = (\eta_1 + \cdots + \eta_n) \circ \psi' \\ \quad \textbf{where } \psi' = \lambda l \rightarrow \textbf{case } t_0 \textbf{ of} \\ \qquad\qquad\qquad\qquad p_1 \rightarrow bv(p_1) \\ \qquad\qquad\qquad\qquad \vdots \\ \qquad\qquad\qquad\qquad p_n \rightarrow bv(p_n) \\ \qquad\qquad \eta_i = \lambda bv(p_i) \rightarrow (i, (t_{i1}, \ldots, t_{ik_i})) \end{array}$$

where $bv(p_i)$ are the variables of p_i. To guarantee that $(\eta_1 + \cdots + \eta_n)$ is a natural transformation, we will ask t_{ij} to be either

- a variable whose occurrences in terms are all recursive; or
- a variable whose occurrences in terms are all non-recursive; or

- a non-recursive term which does not contain variables occurring in recursive terms.

Example 3.5 (Restructure of *filter*) Consider the following definition.

$$
\begin{aligned}
&\mathit{filter} : (a \to \mathit{Bool}) \to [\,a\,] \to [\,a\,] \\
&\mathit{filter}\ p\ (a : as) = \mathbf{if}\ p\ a \\
&\qquad\qquad\qquad\qquad \mathbf{then}\ a : \mathit{filter}\ p\ as \\
&\qquad\qquad\qquad\qquad \mathbf{else}\ \mathit{filter}\ p\ as \\
&\mathit{filter}\ p\ [\,] = [\,]
\end{aligned}
$$

As a hylomorphism

$$
\begin{aligned}
&\mathit{filter}\ p = [\![\, [\,] \triangledown \phi_2, \psi \,]\!]_F \\
&\quad \mathbf{where}\ F = \overline{\mathbf{1}} + \overline{a} \times I \times I \\
&\qquad\quad \phi_2 = \lambda(a, v1, v2) \to \mathbf{if}\ p\ a\ \mathbf{then}\ a : v1 \\
&\qquad\qquad\qquad\qquad\qquad\quad \mathbf{else}\ v2 \\
&\qquad\quad \psi = \lambda l \to \mathbf{case}\ l\ \mathbf{of} \\
&\qquad\qquad\qquad\qquad (a : ls) \to (2, (a, ls, ls)) \\
&\qquad\qquad\qquad\qquad [\,] \to (1, ())
\end{aligned}
$$

The coalgebra can be decomposed like this:

$$
\begin{aligned}
&\psi = (id + \eta_2) \circ \mathit{out}_G \\
&\quad \mathbf{where}\ G = \overline{\mathbf{1}} + \overline{a} \times I \\
&\qquad \eta_2 = \lambda(a, ls) \to (a, ls, ls) \\
&\qquad \mathit{out}_G = \lambda l \to \mathbf{case}\ l\ \mathbf{of} \\
&\qquad\qquad\qquad\qquad [\,] \to (1, ()) \\
&\qquad\qquad\qquad\qquad i : is \to (2, (i, is))
\end{aligned}
$$

Applying *hylo-shift* we get

$$\mathit{filter}\ p = (\!|([\,] \triangledown \phi_2) \circ (id + \eta_2)|\!)_G$$

This is a case where η_2 makes two copies of an input value. The restructure can be done because ls always occurs as a recursive term and a is a variable never appearing in a recursive term. □

Adding constructors

Given a coalgebra

$$
\begin{aligned}
&\psi = \lambda l \to \mathbf{case}\ t_0\ \mathbf{of} \\
&\qquad\qquad C_1\ (p_{11}, \ldots, p_{1r_1}) \to (1, t_1) \\
&\qquad\qquad \vdots \\
&\qquad\qquad C_n\ (p_{n1}, \ldots, p_{nr_n}) \to (n, t_n)
\end{aligned}
$$

where there is a constructor C_{n+1} of the input type which does not occur in the **case** patterns, we can decompose it as follows.

$$\begin{array}{l}\psi = (\eta_1 \triangledown \cdots \triangledown \eta_n \triangledown \bot) \circ \psi' \\ \quad \textbf{where } \psi' = \lambda l \rightarrow \textbf{case } t_0 \textbf{ of} \\ \qquad\qquad\qquad\qquad C_1\ (p_{11}, \ldots, p_{1r_1}) \quad \rightarrow (1, t_1) \\ \qquad\qquad\qquad\qquad \vdots \\ \qquad\qquad\qquad\qquad C_n\ (p_{n1}, \ldots, p_{nr_n}) \quad \rightarrow (n, t_n) \\ \qquad\qquad\qquad\qquad C_{n+1}\ (u_1, \ldots, u_{r_{n+1}}) \rightarrow (n+1, (u_1, \ldots, u_{r_{n+1}})) \\ \qquad \eta_i = \lambda vs \rightarrow (i, vs)\end{array}$$

where $\bot$ is the minimum element of the disjoint n-ary sum. We added to ψ' a new alternative for the constructor C_{n+1}, so that now we could have a better chance to get out_F. We have that $\eta = (\eta_1 \triangledown \cdots \triangledown \eta_n \triangledown \bot)$ is a natural transformation

$$\eta : F_1 + \cdots + F_n + F_{n+1} \Rightarrow F_1 + \cdots + F_n$$

Each functor F_i describes the signature of constructor C_i.

Example 3.6 Consider the following definition

$$\begin{array}{l}mapStream :: (a \rightarrow b) \rightarrow [\,a\,] \rightarrow [\,b\,] \\ mapStream\ f\ (a : as) = f\ a : mapStream\ f\ as\end{array}$$

Now, writing it as a hylomorphism we get

$$\begin{array}{l}mapStream\ f = [\![\phi, \psi]\!]_F \\ \quad \textbf{where } F = \overline{a} \times I \\ \qquad \phi = \lambda(a, v) \rightarrow f\ a : v \\ \qquad \psi = \lambda l \rightarrow \textbf{case } l \textbf{ of} \\ \qquad\qquad\qquad a : as \rightarrow (a, as)\end{array}$$

We can decompose the coalgebra ψ as follows.

$$\begin{array}{l}\psi = (id \triangledown \bot) \circ out_G \\ \quad \textbf{where } G = \overline{a} \times I + \overline{\mathbf{1}} \\ \qquad out_G = \lambda l \rightarrow \textbf{case } l \textbf{ of} \\ \qquad\qquad\qquad\qquad a : as \rightarrow (1, (a, as)) \\ \qquad\qquad\qquad\qquad [\,] \rightarrow (2, ())\end{array}$$

where we have that $id \triangledown \bot : G \Rightarrow F$, and $F\ a$ has to be seen as an unary sum.

Finally, the restructure looks like the following after the application of *hylo-shift*:

$$mapSteam\ f = (\!|\phi \circ (id \triangledown \bot)|\!)_G$$

□

$$
\begin{array}{l}
\mathcal{T}(F, \phi :: G\ a \to a) :: (F\ a \to a) \to (G\ a \to a) \\
\mathcal{T}(F_1 + \cdots + F_m, \phi_1 \triangledown \cdots \triangledown \phi_n) = \\
\qquad\qquad \lambda(\alpha_1 \triangledown \cdots \triangledown \alpha_m) \to \mathcal{T}'(\phi_1) \triangledown \cdots \triangledown \mathcal{T}'(\phi_n) \\
\quad \textbf{where}\ \mathcal{T}'(\lambda bvs \to t) = \lambda bvs \to \mathcal{A}(t) \\
\qquad\qquad \mathcal{A}(v) \qquad\quad = v \quad \text{if } v \text{ is a recursive variable} \\
\qquad\qquad \mathcal{A}(C_j\ t_1\ \ldots\ t_k) = \alpha_j\ (F_j\ \mathcal{A}\ (t_1, \ldots, t_k)) \\
\qquad\qquad \mathcal{A}(t) \qquad\quad = (\!|\alpha_1 \triangledown \cdots \triangledown \alpha_m|\!)_F\ t \quad \text{every other case}
\end{array}
$$

Figure 3.3: Derivation algorithm for τ

3.2.3 Derivation of τ

Given an algebra ϕ we would like to be able to determine if it can be written as $\tau(in_F)$, being τ a term with type $\tau :: \forall a.(F\ a \to a) \to (G\ a \to a)$. In this section we describe an algorithm that derives τ from ϕ.

The algorithm requires that the G-algebra ϕ that is given as input must be of the form $\phi_1 \triangledown \cdots \triangledown \phi_n$, such that each ϕ_i is a term $(\lambda(v_1, \ldots, v_{k_i}) \to t_i)$, where t_i is in the following normal form:

1. it is a recursive variable; or
2. it is a constructor application $C_j\ (t'_1, \ldots, t'_m)$ where each t'_j in a recursive position of constructor C_j is in normal form. Those t'_j that are not in recursive positions can be any term not referencing recursive variables. We say that a term t'_j is in a recursive position if functor F (not G) tells that t'_j is in a recursive position of constructor C_j; or
3. it is any term not referencing recursive variables.

When there are **case** structures embedded in ϕ we still can get the normal form by restructuring the algebra with the techniques used for recognizing in_F.

We present the derivation algorithm for τ in Figure 3.3. The objective of algorithm $\mathcal{A}$ is to abstract constructors, substituting them by the corresponding operations of the F-algebra $\alpha = \alpha_1 \triangledown \cdots \triangledown \alpha_m$. This algorithm is applied recursively only to the recursive arguments, which are indicated by functor F. The functor F is also an input to the derivation algorithm. Its expression can be obtained from a coalgebra in out_F form. Such a coalgebra will be available sooner or later, as we will be deriving τ only if we want to fuse the hylomorphism with algebra ϕ with a fold.

Given a G-algebra ϕ, the output of the algorithm is such that

- $\mathcal{T}(F, \phi) :: \forall a.(F\ a \to a) \to (G\ a \to a)$

- $\mathcal{T}(F, \phi)\ (in_F) = \phi$

This is in essence the algorithm given by Onoue et al.[OHIT97]. We have required the input to be more constrained and removed a case we considered erroneous from the original algorithm. See Section A.1 in the appendices for counterexamples showing the effects of ignoring these additional restrictions we considered.

Example 3.7 (Derivation of τ) Consider the following definition

$$
\begin{aligned}
&\mathit{appendll} :: a \to [[a]] \to [[a]] \\
&\mathit{appendll}\ a\ [\,] = [\,] \\
&\mathit{appendll}\ a\ (xs : xss) = (a : xs) : \mathit{appendll}\ xss
\end{aligned}
$$

As a hylomorphism it looks like

$$
\begin{aligned}
&\mathit{appendll}\ a = (\!|\phi_1 \triangledown \phi_2|\!)_F \\
&\quad \textbf{where}\ F = \overline{\mathbf{1}} + \overline{[a]} \times I \\
&\qquad\quad \phi_1 = \lambda() \to [\,] \\
&\qquad\quad \phi_2 = \lambda(xs, xss) \to (a : xs) : xss
\end{aligned}
$$

Applying algorithm $\mathcal{T}(F, \phi_1 \triangledown \phi_2)$ we get the transformer

$$
\begin{aligned}
&\tau :: (F\ b \to b) \to F\ b \to b \\
&\tau\ (\alpha_1 \triangledown \alpha_2) = \phi_1 \triangledown \phi_2 \\
&\quad \textbf{where}\ \phi_1 = \lambda() \to \mathcal{A}([\,]) \\
&\qquad\quad \phi_2 = \lambda(xs, xss) \to \mathcal{A}((a : xs) : xss)
\end{aligned}
$$

which is the same as

$$
\begin{aligned}
&\tau :: (F\ b \to b) \to F\ b \to b \\
&\tau\ (\alpha_1 \triangledown \alpha_2) = \phi_1 \triangledown \phi_2 \\
&\quad \textbf{where}\ \phi_1 = \lambda() \to \alpha_1\ () \\
&\qquad\quad \phi_2 = \lambda(xs, xss) \to \alpha_2\ (a : xs, \mathcal{A}(xss))
\end{aligned}
$$

Note that we have not abstracted away one of the occurrences of (:) in ϕ_2. This is because it occurs in a non-recursive position according to functor F, and therefore algorithm $\mathcal{A}$ is applied recursively only to the argument xss.

Finally, we get

$$
\begin{aligned}
&\tau :: (F\ b \to b) \to G\ b \to b \\
&\tau\ (\alpha_1 \triangledown \alpha_2) = \phi_1 \triangledown \phi_2 \\
&\quad \textbf{where}\ \phi_1 = \lambda() \to \alpha_1\ () \\
&\qquad\quad \phi_2 = \lambda(xs, xss) \to \alpha_2\ (a : xs, xss)
\end{aligned}
$$

□

3.2.4 Derivation of σ

To conclude with the algebra and coalgebra analysis, we will focus now on deciding whether a coalgebra $\psi = \sigma(out_F)$, where $\sigma :: \forall a.(a \to F\ a) \to (a \to G\ a)$. We will answer that question with an algorithm that takes a coalgebra ψ and returns (probably) its decomposition as $\sigma(out_F)$.

The input coalgebra must be of the form

$$\lambda v_0 \to \textbf{case}\ v_0\ \textbf{of}\ p_1 \to (1, (t_{11}, \ldots, t_{1m})); \ldots; p_n \to (n, (t_{n1}, \ldots, t_{nm}))$$

That is, the case analysis must be evaluated over the input variable. There are also the following restrictions:

- Recursive terms must be variables, and non-recursive terms must not reference those variables.
- The patterns p_i must satisfy the following normal form:
  1. the pattern is a variable; or
  2. the pattern is of the form $C_i\ (p'_1, \ldots, p'_{k_i})$ and pattern p'_j appearing in a recursive position of C_i is in normal form. A pattern p'_j in a non-recursive position can have any shape as far as it does not reference variables appearing in recursive terms. A pattern p'_j is said to appear in a recursive position if the functor F (not G) tells so, being F the functor characterizing the input data type of the coalgebra.

In a dual way to the derivation algorithm for τ, the derivation algorithm for σ abstracts constructors from the coalgebra patterns. For the sake of showing this duality, we will extend patterns in the core language with a convenient notation for our manipulations. The new pattern alternative is:

$$p :: = t \cdot p$$

with the following semantics: To match a value v against a pattern $(t \cdot p)$, evaluate $(t\ v)$ and match the result against p. This idea is known as *view patterns* and has been first proposed by Wadler [Wad87]. It has been added recently as an extension to the GHC compiler (version 6.10.1[1]).

In Figure 3.4 we present our derivation algorithm for σ. Algorithm $\mathcal{B}$, which abstracts constructors inside patterns, here plays the same role as algorithm $\mathcal{A}$ played in the derivation algorithm for τ.

Example 3.8 (Derivation of σ) Consider the following definition

[1] `http://www.haskell.org/ghc/docs/latest/html/users_guide/syntax-extns.html#view-patterns`

$$
\begin{array}{l}
\mathcal{S}(F, \psi :: a \to G\ a) :: \forall a.(a \to F\ a) \to (a \to G\ a) \\
\mathcal{S}(F_1 + \cdots + F_m, \lambda v \to \textbf{case}\ v\ \textbf{of}\ p_1 \to t_1; \ldots; p_n \to t_n) = \\
\quad \lambda\beta \to \lambda v \to \textbf{case}\ v\ \textbf{of}\ \mathcal{B}\ (p_1) \to t_1; \ldots; \mathcal{B}(p_n) \to t_n \\
\quad \textbf{where}\ \mathcal{B}(v) \qquad\qquad = v \ \text{ if } v \text{ is a recursive variable} \\
\qquad\qquad \mathcal{B}(C_j\ p_1\ \ldots\ p_k) = \beta{\cdot}(j, F_j\ \mathcal{B}\ (p_1, \ldots, p_k)) \\
\qquad\qquad \mathcal{B}(p) \qquad\qquad = [\![\beta]\!]_G{\cdot}p \ \text{ all other cases}
\end{array}
$$

Figure 3.4: Derivation algorithm for σ

$$
\begin{array}{l}
odds :: [b] \to [b] \\
odds\ [\,] \qquad = [\,] \\
odds\ (b : [\,]) \quad = [b] \\
odds\ (b : \_ : l) = b : odds\ l
\end{array}
$$

Deriving a hylomorphism from this definition we get

$$
\begin{array}{l}
odds = [\![\phi, \psi]\!]_G \\
\quad \textbf{where}\ G = \overline{\mathbf{1}} + \overline{b} + \overline{b} \times I \\
\qquad \psi = \lambda v \to \textbf{case}\ v\ \textbf{of} \\
\qquad\qquad\qquad [\,] \qquad \to (1, ()) \\
\qquad\qquad\qquad b : [\,] \quad \to (2, b) \\
\qquad\qquad\qquad b : \_ : l \to (3, (b, l)) \\
\qquad \phi\ (1, ()) = [\,] \\
\qquad \phi\ (2, b) = [b] \\
\qquad \phi\ (3, (b, r)) = b : r
\end{array}
$$

Applying the $\mathcal{S}$ algorithm over ψ we get:

$$
\begin{array}{l}
\sigma :: (b \to F\ b) \to b \to G\ b \\
\sigma = \lambda\beta \to \lambda v \to \textbf{case}\ v\ \textbf{of} \\
\qquad\qquad\qquad \beta{\cdot}(1, ()) \qquad\qquad\qquad\ \to (1, ()) \\
\qquad\qquad\qquad \beta{\cdot}(2, (b, \beta{\cdot}(1, ()))) \qquad \to (2, b) \\
\qquad\qquad\qquad \beta{\cdot}(2, (b, \beta{\cdot}(2, (\_, l)))) \to (3, (b, l)) \\
\quad \textbf{where}\ F = \overline{\mathbf{1}} + \overline{b} \times I
\end{array}
$$

such that $\psi = \sigma(out_F)$.

Now, let us fuse *odds* with *map*.

$$
\begin{array}{l}
oddsmap :: (a \to b) \to [a] \to [b] \\
oddsmap\ f = odds \circ map\ f \\
map :: (a \to b) \to [a] \to [b] \\
map\ f = [\![\psi']\!]_F
\end{array}
$$

$$\begin{aligned}&\textbf{where } \psi'\ [\,] &&= (1, ())\\ &\qquad\ \ \psi'\ (x : xs) &&= (2, (f\ x, xs))\end{aligned}$$

The result is $oddsmap\ f = [\![\phi, \sigma(\psi')]\!]_G$. □

We intended to base our implementation in the algorithm provided by Onoue et al. [OHIT97]. It turned out, later, that the algorithm as it is in their paper changes the semantics of the coalgebra in a radical way. In Haskell, a pattern is a tree-like structure whose nodes are matched in preorder against a value. We have been careful to preserve that order in our proposal. In contrast, in [OHIT97] checks are reorganized so they are performed in breath-first order, changing the behavior of the functions in the presence of partially defined arguments.

To guarantee the correctness of the output, we have also introduced a restriction on the form of the recursive terms in the coalgebra, which was not present in Onoue et al.'s proposal. See Section A.2 in the appendices for a counterexample showing the effects of ignoring this restriction.

Inlining $\sigma(\psi')$

If we want to see *oddsmap* as a Haskell definition, we need to go through a rather laborious process. We focus now on getting an almost-Haskell definition for $\sigma(\psi')$. The rest of the inlining process is discussed in the next section.

First, we remove our uses of the $t{\cdot}p$ patterns. For that, we first remove nested patterns from $\sigma(\psi')$:

$$\begin{aligned}&\sigma :: (b \to F\ b) \to b \to G\ b\\ &\sigma(\psi')\ v =\\ &\quad \textbf{case } v \textbf{ of}\\ &\qquad \psi'{\cdot}(1, ()) \to (1, ())\\ &\qquad \_ \to \textbf{case } v \textbf{ of}\\ &\qquad\quad \psi'{\cdot}(2, (b, u_0)) \to\\ &\qquad\qquad \textbf{case } u_0 \textbf{ of}\\ &\qquad\qquad\quad \psi'{\cdot}(1, ()) \to (2, b)\\ &\qquad\qquad\quad \_ \qquad\quad \to \textbf{case } v \textbf{ of}\\ &\qquad\qquad\qquad\qquad\quad \psi'{\cdot}(2, (b, u_0)) \to \textbf{case } u_0 \textbf{ of}\\ &\qquad\qquad\qquad\qquad\qquad\qquad\quad \psi'{\cdot}(2, (\_, l)) \to (3, (b, l))\\ &\qquad\quad \_ \to \textbf{case } v \textbf{ of}\\ &\qquad\qquad\quad \psi'{\cdot}(2, (b, u_0)) \to \textbf{case } u_0 \textbf{ of}\\ &\qquad\qquad\qquad\qquad\qquad\quad \psi'{\cdot}(2, (\_, l)) \to (3, (b, l))\end{aligned}$$

Now, we remove the uses of the $t{\cdot}p$ patterns:

$$\begin{aligned}&\sigma :: (b \to F\ b) \to b \to G\ b\\ &\sigma(\psi')\ v =\end{aligned}$$

$$
\begin{array}{l}
\textbf{case } \psi'\ v \textbf{ of} \\
\quad (1,()) \to (1,()) \\
\quad \_ \to \textbf{case } \psi'\ v \textbf{ of} \\
\qquad (2,(b,u_0)) \to \\
\qquad\quad \textbf{case } \psi'\ u_0 \textbf{ of} \\
\qquad\qquad (1,()) \to (2,b) \\
\qquad\qquad \_ \qquad \to \textbf{case } \psi'\ v \textbf{ of} \\
\qquad\qquad\qquad\qquad (2,(b,u_0)) \to \textbf{case } \psi'\ u_0 \textbf{ of} \\
\qquad\qquad\qquad\qquad\qquad\qquad (2,(\_,l)) \to (3,(b,l)) \\
\qquad \_ \to \textbf{case } \psi'\ v \textbf{ of} \\
\qquad\qquad (2,(b,u_0)) \to \textbf{case } \psi'\ u_0 \textbf{ of} \\
\qquad\qquad\qquad\qquad (2,(\_,l)) \to (3,(b,l))
\end{array}
$$

We could then somewhat shrink the term to this:

$$
\begin{array}{l}
\sigma :: (b \to F\ b) \to b \to G\ b \\
\sigma(\psi')\ v = \\
\quad \textbf{case } \psi'\ v \textbf{ of} \\
\qquad (1,()) \qquad \to (1,()) \\
\qquad (2,(b,u_0)) \to \textbf{case } \psi'\ u_0 \textbf{ of} \\
\qquad\qquad\qquad (1,()) \qquad \to (2,b) \\
\qquad\qquad\qquad (2,(\_,l)) \to (3,(b,l))
\end{array}
$$

To get the above term we had to apply some simplifications listed below. To express the simplifications we will write as $t[\,v_1,\ldots,v_n \,/\, v'_1,\ldots,v'_n\,]$ the simultaneous substitution of variables v'_i by v_i in term t. We will also use term contexts C with a placeholder, where $C[\,t\,]$ denotes the term resulting from filling the placeholder with term t.

1. $$
\begin{array}{l}
\textbf{case } \beta\ u \textbf{ of} \\
\quad \vdots \\
\quad (i,(v_1,\ldots,v_n)) \to C[\textbf{case } \beta\ u \textbf{ of} \\
\qquad\qquad\qquad\qquad \ldots;(i,(v'_1,\ldots,v'_n)) \to e;\ldots] \\
\quad \vdots \\
= \textbf{case } \beta\ u \textbf{ of} \\
\qquad \ldots;(i,(v_i,\ldots,v_n)) \to C[\,e[\,v_i,\ldots,v_n \,/\, v'_1,\ldots,v'_n\,]];\ldots
\end{array}
$$

2. $$
\begin{array}{l}
\textbf{case } \beta\ u \textbf{ of} \\
\quad \vdots \\
\quad (i,(v_1,\ldots,v_n)) \to C[\textbf{case } \beta\ u \textbf{ of} \\
\qquad\qquad\qquad\qquad \vdots \quad \text{-- alternatives do not match i}
\end{array}
$$

$$\_ \to e\,]$$

$$\vdots = \textbf{case}\ \beta\ u\ \textbf{of}$$
$$\ldots; (i, (v_i, \ldots, v_n)) \to C[\,e\,]; \ldots$$

3. $\textbf{case}\ \beta\ u\ \textbf{of}$

$$\begin{array}{l}
\vdots \quad \text{-- at least one alternative forces beta evaluation of} \\
\quad\ \text{-- the case term alternatives do not match i} \\
_ \to C[\textbf{case}\ \beta\ u\ \textbf{of} \\
\qquad\qquad \vdots \quad \text{-- all these alternatives do not match i} \\
\qquad\qquad (i, (v_1, \ldots, v_n)) \to e \\
\qquad\qquad \vdots \qquad\qquad\qquad\qquad]
\end{array}$$

$$\begin{array}{l}
\vdots \\
= \\
\quad \textbf{case}\ \beta\ u\ \textbf{of} \\
\qquad \vdots \\
\qquad (i, (v_i, \ldots, v_n)) \to C[\,e\,] \\
\qquad _ \to C\ [\textbf{case}\ \beta\ u\ \textbf{of} \\
\qquad\qquad\qquad \ldots; (i, (v_1, \ldots, v_n)) \to e; \ldots\,]
\end{array}$$

4. $\textbf{case}\ \beta\ u\ \textbf{of}$

$$\begin{array}{l}
\vdots \quad \text{-- all of the alternatives of the inner case appear} \\
\quad\ \text{-- among these alternatives} \\
_ \to C[\textbf{case}\ \beta\ u\ \textbf{of}\ \ldots; _ \to e\,] \\
= \\
\quad \textbf{case}\ \beta\ u\ \textbf{of}\ \ldots; _ \to C[\,e\,]
\end{array}$$

Now, we will β-reduce the coalgebra applications we have in σ.

$$\begin{array}{l}
\sigma :: (b \to F\ b) \to b \to G\ b \\
\sigma(\psi')\ v = \\
\quad \textbf{case}\ (\textbf{case}\ v\ \textbf{of} \\
\qquad\qquad [\,] \quad\ \to (1, ()) \\
\qquad\qquad x : xs \to (2, (f\ x, xs)) \\
\qquad)\ \textbf{of} \\
\qquad (1, ()) \to (1, ()) \\
\qquad (2, (b, u_0)) \to \textbf{case}\ (\textbf{case}\ u_0\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad [\,] \quad\ \to (1, ()) \\
\qquad\qquad\qquad\qquad\qquad x : xs \to (2, (f\ x, xs))
\end{array}$$

```
                  ) of
                (1, ()) → (2, b)
                (2, (_, l)) → (3, (b, l))
```

If we apply the case-of-case transformation we can remove the nested **case**s.

```
σ :: (b → F b) → b → G b
σ(ψ′) v =
  case v of
    [ ]       → case (1, ()) of
                  (1, ())      → (1, ())
                  (2, (b, u0)) → case u0 of
                                   [ ]       → case (1, ()) of
                                                 (1, ())      → (2, b)
                                                 (2, (_, l)) → (3, (b, l))
                                   x′ : xs′ → case (2, (f x′, xs′)) of
                                                 (1, ())      → (2, b)
                                                 (2, (_, l)) → (3, (b, l))
    (x : xs) → case (2, (f x, xs)) of
                  (1, ())      → (1, ())
                  (2, (b, u0)) → case u0 of
                                   [ ]       → case (1, ()) of
                                                 (1, ())      → (2, b)
                                                 (2, (_, l)) → (3, (b, l))
                                   x′ : xs′ → case (2, (f x′, xs′)) of
                                                 (1, ())      → (2, b)
                                                 (2, (_, l)) → (3, (b, l))
```

Reducing the cases over terms of the form $(i, (\ldots))$, we finally obtain:

```
σ :: (b → F b) → b → G b
σ(ψ′) v =
  case v of
    [ ]       → (1, ())
    (x : xs) → case xs of
                  [ ]       → (2, f x)
                  x′ : xs′ → (3, (f x, xs′))
```

All of the steps performed above can be reasonably automated. From here the rest of the inlining process is more straightforward as we will see in the next section.

3.3 Inlining hylomorphisms

Inlining is the inverse process of deriving a hylomorphism. We have as input a hylomorphism in the form

$$
\begin{array}{l}
f\ v_1\ \ldots\ v_{m-1} = [\![\phi_1 \triangledown \cdots \triangledown \phi_n, \psi]\!]_{F_1+\cdots+F_n} \\
\quad \textbf{where}\ \psi\ v_m = \textbf{case}\ t_0\ \textbf{of} \\
\qquad\qquad\qquad\qquad p_1 \rightarrow (0, t_1) \\
\qquad\qquad\qquad\qquad\quad \vdots \\
\qquad\qquad\qquad\qquad p_n \rightarrow (n, t_n)
\end{array}
$$

and we want to get the equivalent recursive definition. For the sake of presenting the algorithm, we assume that there are no nested cases in the coalgebra. You can also find inlining discussed in [Sch00].

From the definition of hylomorphism, we can obtain the following recursive function:

$$f\ v_1\ \ldots\ v_{m-1} = (\phi_1 \triangledown \cdots \triangledown \phi_n) \circ (F_1 + \cdots + F_n)\ (f\ v_1\ \ldots\ v_{m-1}) \circ \psi$$

where we can move the algebra and the functor components into the alternatives of the coalgebra ψ:

$$
\begin{array}{l}
f\ v_1\ \ldots\ v_m = \textbf{case}\ t_0\ \textbf{of} \\
\qquad\qquad\qquad p_1 \rightarrow \phi_1\ (F_1\ (f\ v_1\ \ldots\ v_{m-1})\ t_1) \\
\qquad\qquad\qquad\quad \vdots \\
\qquad\qquad\qquad p_n \rightarrow \phi_n\ (F_n\ (f\ v_1\ \ldots\ v_{m-1})\ t_n)
\end{array}
$$

Simplifying through β-reductions the **case** alternatives, we obtain the recursive definition we want, which may be again in the input form required by the derivation algorithm for hylomorphisms.

Example 3.9 (Inlining of *oddsmap*) Consider the definition of *oddsmap* from the previous section, where we have replace the nested **case**s in the coalgebra by a **case** with nested constructors in the patterns.

$$
\begin{array}{l}
oddsmap :: (a \rightarrow b) \rightarrow [a] \rightarrow [b] \\
oddsmap\ f = [\![const\ [\,]\triangledown(:[\,])\triangledown uncurry\ (:), \sigma(\psi')]\!]_{\overline{\mathbf{1}}+\overline{b}+\overline{b}\times I} \\
\quad \textbf{where}\ \sigma(\psi')\ v = \textbf{case}\ v\ \textbf{of} \\
\qquad\qquad\qquad\qquad [\,] \rightarrow (1, ()) \\
\qquad\qquad\qquad\qquad (x : [\,]) \rightarrow (2, f\ x) \\
\qquad\qquad\qquad\qquad (x : x' : xs') \rightarrow (3, (f\ x, xs')) \\
\qquad\quad const\ c\ \_ = c \\
\qquad\quad uncurry\ f\ (a, b) = f\ a\ b
\end{array}
$$

After inlining we obtain:

$$
\begin{array}{l}
oddsmap\ f\ v = \\
\quad \textbf{case}\ v\ \textbf{of} \\
\qquad [\,] \rightarrow const\ [\,]\ ((\overline{\mathbf{1}}\ (oddsmap\ f))\ ())
\end{array}
$$

$$
\begin{array}{ll}
(x : [\,]) & \rightarrow (:[\,])\ ((\overline{b}\ (oddsmap\ f))\ (f\ x)) \\
(x : x' : xs') & \rightarrow uncurry\ (:)\ (((\overline{b} \times I)\ (oddsmap\ f))\ (f\ x, xs'))
\end{array}
$$

Which can be simplified to obtain the following Haskell definition:

$$
\begin{array}{l}
oddsmap :: (a \rightarrow b) \rightarrow [a] \rightarrow [b] \\
oddsmap\ f\ [\,] = [\,] \\
oddsmap\ f\ (x : [\,]) = [f\ x] \\
oddsmap\ f\ (x : \_ : xs') = f\ x : oddsmap\ f\ xs'
\end{array}
$$

□

3.4 Summary

In this chapter we presented the stages of HFusion processing and the algorithms used in them. Those stages are hylomorphism derivation, classification and restructuring, application of fusion laws and inlining.

We are using the algorithm of Onoue et al. [OHIT97] for deriving hylomorphisms. The hylomorphism restructuring algorithms proposed in [Sch00] and [OHIT97] were reformulated, and a few more restructurings were added.

We presented the algorithm needed for deriving an algebra transformer τ from a given algebra. The algorithm was slightly reformulated with respect to the version in [OHIT97] by changing a bit the restrictions on the input and by eliminating one of its cases. Counterexamples justifying these modifications can be found in Section A.1 in the appendices.

We also presented the algorithm for deriving a coalgebra transformer σ from a given coalgebra. The algorithm proposed in [OHIT97] was completely rewritten, being view patterns the fundamental feature driving the rewriting. In our version, the algorithm is more compact, simpler, and it really looks like the dual version of the derivation algorithm for τ. This will make extending the algorithms easier to describe in the next chapters.

Finally, we have shown how hylomorphism are inlined as an inverse process of their derivation. It is an algorithm that can also be found in [Sch00].

Chapter 4

Paramorphism fusion

Paramorphisms [Mee92] correspond to primitive recursive functions. Therefore, like folds, they capture functions that are defined by structural recursion. Fusion laws for paramorphisms are important for deforesting programs because in general the acid rain laws based on fold, unfolds and hylos can not handle this kind of functions appropriately. As an example, consider the following program.

$$
\begin{aligned}
&\mathit{repf}\ x\ y\ p = \mathit{replace}\ x\ y \circ \mathit{filter}\ p \\
&\mathit{replace} \qquad\qquad :: \mathit{Eq}\ a \Rightarrow a \to a \to [a] \to [a] \\
&\mathit{replace}\ x\ y\ [\,] \qquad = [\,] \\
&\mathit{replace}\ x\ y\ (a : as) = \mathbf{if}\ a == x\ \mathbf{then}\ y : as\ \mathbf{else}\ a : \mathit{replace}\ x\ y\ as
\end{aligned}
$$

The fused program should be:

$$
\begin{aligned}
&\mathit{repf}\ x\ y\ p\ [\,] \qquad = [\,] \\
&\mathit{repf}\ x\ y\ (a : as) = \mathbf{if}\ p\ a \\
&\qquad\qquad\qquad\qquad \mathbf{then\ if}\ a == x \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{then}\ y : \mathit{filter}\ p\ as \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ a : \mathit{repf}\ x\ y\ p\ as \\
&\qquad\qquad\qquad\qquad \mathbf{else}\ \mathit{repf}\ x\ y\ p\ as
\end{aligned}
$$

However, we can not get this function with the fusion laws presented so far, since *replace* and *filter* are of the form:

$$
\begin{aligned}
&\mathit{replace}\ x\ y = [\![[\,] \triangledown \phi_2, \psi]\!]_F \\
&\quad \mathbf{where}\ F = \overline{\mathbf{1}} + \overline{a} \times \overline{[a]} \times I \\
&\qquad\qquad \psi\ [\,] = (1, ()) \\
&\qquad\qquad \psi\ (a : as) = (2, (a, as, as)) \\
&\qquad\qquad \phi_2\ (a, as, vs) = \mathbf{if}\ a == x\ \mathbf{then}\ y : as \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{else}\ x : vs \\
&\mathit{filter}\ p = (\!| [\,] \triangledown \phi_2' |\!)_{F'}
\end{aligned}
$$

$$\begin{aligned}\textbf{where } F' &= \overline{\mathbf{1}} + \overline{a} \times I \\ \phi_2'\ (a, vs) &= \textbf{if } p\ a \textbf{ then } a : vs \\ &\qquad\qquad \textbf{else } vs\end{aligned}$$

In fact, as *replace* can not be restructured as a fold, and *filter* can not be restructured as an unfold, we can not apply any of the acid rain laws (Theorem 2.7).

In this chapter we review the definition of paramorphisms, and we introduce a new program scheme called *generalized paramorphism*, which generalizes paramorphism much in the same way hylomorphism generalizes fold. We present corresponding acid rain laws for paramorphisms and generalized paramorphisms, which enable us to handle cases like that of *replace* $x\ y \circ$ *filter* p.

We also show examples where applying our paramorphism laws may worsen the performance of the original programs. This is a first red flag indicating that fused programs may not always be better. Fortunately, these cases can be ruled out with a relatively simple syntactical analysis over the involved definitions prior to fusion.

This chapter is a revised version of [DP06b].

4.1 Paramorphisms

In this section we review the definition of paramorphism and some of its standard laws. We also introduce new acid rain laws that relate paramorphisms with folds.

Given a function $\phi :: F(a \times \mu F) \to a$, the *paramorphism* $(\!|\phi|\!)_F :: \mu F \to a$ is the least function that satisfies the equation $f \circ in_F = \phi \circ F\langle f, id\rangle$, where $\langle f, g\rangle\ x\ =\ (f\ x, g\ x)$ is known as the *split* operator.

The following diagram makes the types explicit:

$$\begin{array}{ccc} \mu F & \xrightarrow{\quad (\!|\phi|\!)_F \quad} & a \\ {\scriptstyle in_F}\uparrow & & \uparrow{\scriptstyle \phi} \\ F\mu F & \xrightarrow[F\langle (\!|\phi|\!)_F,\, id\rangle]{} & F(a \times \mu F) \end{array}$$

The difference between paramorphisms and folds is in the amount of information available in each recursive step. In addition to the values returned by the recursive calls (as in fold), function ϕ has also available their arguments. As we will see later on in this section, this subtle difference with folds makes paramorphisms inappropriate for fusion in some cases.

Example 4.1 Consider the following definition.

$$
\begin{array}{l}
\mathit{dropWhile} :: (a \to \mathit{Bool}) \to [\,a\,] \to [\,a\,] \\
\mathit{dropWhile}\ p\ [\,] = [\,] \\
\mathit{dropWhile}\ p\ (a : as) = \mathbf{if}\ p\ a\ \mathbf{then}\ \mathit{dropWhile}\ p\ as\ \mathbf{else}\ a : as
\end{array}
$$

The function *dropWhile* can be defined as:

$$
\begin{array}{l}
\mathit{dropWhile} = (\!|\, [\,] \triangledown \phi_2 \,|\!)_F \\
\quad \mathbf{where}\ F = \overline{\mathbf{1}} + \overline{a} \times I \\
\qquad\quad \phi_2 = \lambda(a, (ys, as)) \to \mathbf{if}\ p\ a\ \mathbf{then}\ ys\ \mathbf{else}\ a : as
\end{array}
$$

□

The following equations express the well-known relationship between paramorphisms and folds.

$$(\!|\phi|\!)_F = \pi_1 \circ (\!|\langle \phi, in_F \circ F\pi_2 \rangle|\!)_F \tag{4.1}$$

$$(\!|\phi|\!)_F = (\!|\phi \circ F\ \pi_1|\!)_F \tag{4.2}$$

Equation (4.1) is usually taken as the definition of paramorphism. It states that a paramorphism can be implemented as a fold that produces a pair, whose second component contains a (recursively generated) copy of the input. Equation (4.2) shows that a fold is a paramorphism that ignores the copy of the arguments to the recursive calls.

The following is the fusion law for paramorphism [Mee92].

$$f \text{ strict} \ \wedge\ f \circ \phi = \phi' \circ F(f \times id) \ \Rightarrow\ f \circ (\!|\phi|\!)_F = (\!|\phi'|\!)_F \tag{4.3}$$

The rest of this section is devoted to the analysis of acid rain laws for paramorphisms. We are not aware that they have been presented before. These laws will serve us as basis for designing the acid rain laws for generalized paramorphisms in Section 4.2.

The first law we consider refers to the composition of a fold with a paramorphism.

Proposition 4.2 (fold-para fusion) *For strict ϕ,*

$$\tau :: \forall a.\ (F\ a \to a) \to (G\ (a \times \mu G) \to a) \ \Rightarrow\ (\!|\phi|\!)_F \circ (\!|\tau(in_F)|\!)_G = (\!|\tau(\phi)|\!)_G$$

Proof Since $(\!|\phi|\!)_F$ is an homomorphism between the algebras in_F and ϕ, by the free theorem associated with the polymorphic type of τ it follows that

$$(\!|\phi|\!)_F \circ \tau(in_F) = \tau(\phi) \circ G\ ((\!|\phi|\!)_F \times id)$$

Therefore, by applying (4.3) we obtain the desired result. The strictness condition required to $(\!|\phi|\!)_F$ in (4.3) follows from the assumption that ϕ is strict. □

The next fusion law refers to the composition between a paramorphism and a fold. It is particularly interesting and important as it exhibits a case in which the paramorphism internalizes the generation of values of the intemediate data structure that wants to be eliminated. The following lemma will be used in the proof of the law.

Lemma 4.3 *For F-algebras $\phi :: Fa \to a$ and $\psi :: F(b \times a) \to (b \times a)$, and strict $f :: a \to b$,*

$$\langle f, id\rangle \circ \phi = \psi \circ F\langle f, id\rangle \quad \Rightarrow \quad f \circ (\!|\phi|\!)_F = (\!|\pi_1 \circ \psi \circ F(id \times (\!|\phi|\!)_F)|\!)_F$$

Proof Let us call p the paramorphism and c the fold. By definition of paramorphism and fold we have that

$$p \circ in_F = \pi_1 \circ \psi \circ F\langle p, c\rangle \qquad \text{and} \qquad c \circ in_F = \phi \circ F\, c$$

The two functions are defined simultaneously in an asymmetric way. That is, p depends on c while c does not depend on p. Definitions following this pattern are called a *zygomorphism* [Mal90]. From the definition of p and c, it can be derived that [Fok92]:

$$\langle p, c\rangle = (\!|\langle \pi_1 \circ \psi, \phi \circ F\pi_2\rangle|\!)_F$$

In the context of **Cpo** this equation is proved by fixed point induction. If we call pc the split $\langle p, c\rangle$, then the statement of the lemma can be rewritten as:

$$\langle f, id\rangle \circ \phi = \psi \circ F\langle f, id\rangle \quad \Rightarrow \quad f \circ \pi_2 \circ pc = \pi_1 \circ pc$$

which can then be proved by fixed point induction. □

Proposition 4.4 (para-fold fusion) *For strict ϕ,*

$$\begin{array}{l} \quad \tau :: \forall a.\, (F\ a \to a) \to (G\ a \to a) \\ \Rightarrow \\ \quad (\!|\phi|\!)_F \circ (\!|\tau(in_F)|\!)_G = (\!|\pi_1 \circ \tau(\langle \phi, in_F \circ F\pi_2\rangle) \circ G(id \times (\!|\tau(in_F)|\!)_G)|\!)_G \end{array}$$

Proof From the definition of paramorphism we can derive that $\langle (\!|\phi|\!)_F, id\rangle$ is an homomorphism between the F-algebras in_F and $\langle \phi, in_F \circ F\pi_2\rangle$:

$$\langle (\!|\phi|\!)_F, id\rangle \circ in_F = \langle \phi, in_F \circ F\pi_2\rangle \circ F\langle (\!|\phi|\!)_F, id\rangle$$

Then, by the free theorem associated with the polymorphic type of τ it follows that $\langle (\!|\phi|\!)_F, id\rangle$ is also an homomorphism between the G-algebras $\tau(in_F)$ and $\tau(\langle \phi, in_F \circ F\pi_2\rangle)$:

$$\langle (\!|\phi|\!)_F, id\rangle \circ \tau(in_F) = \tau(\langle \phi, in_F \circ F\pi_2\rangle) \circ G\langle (\!|\phi|\!)_F, id\rangle$$

Finally, by Lemma 4.3 the desired result follows. Strictness of $(\!|\phi|\!)_F$, necessary for the application of Lemma 4.3, is a consequence of the assumption that ϕ is strict. □

Example 4.5 This example shows a simple case in which the fold is copied into the body of the resulting paramorphism, producing multiple generations of data structures.

$$
\begin{array}{l}
tf\ p = tails \circ filter\ p \\
tails \quad\quad :: [a] \to [[a]] \\
tails\ [\,] \quad\quad = [\,] \\
tails\ (a : as) = as : tails\ as
\end{array}
$$

Function *tails* is a paramorphism while *filter* is a fold:

$$
\begin{array}{l}
tails = (\!|[\,] \triangledown \phi_2|\!)_F \\
\quad \textbf{where}\ F = \overline{\mathbf{1}} + \overline{a} \times I \\
\qquad\quad \phi_2 = \lambda(a, (ys, as)) \to as : ys \\
filter\ p = (\!|[\,] \triangledown \phi_2'|\!)_F \\
\quad \textbf{where}\ \phi_2'\ (a, vs) = \textbf{if}\ p\ a\ \textbf{then}\ a : vs\ \textbf{else}\ vs
\end{array}
$$

The algebra of *filter* can be expressed as $\tau(in_F)$, where τ is a polymorphic function given by:

$$
\begin{array}{ll}
\tau & :: (F\ b \to b) \to F\ b \to b \\
\tau(\alpha) & = \tau_1(\alpha) \triangledown \tau_2(\alpha) \\
\tau_1(\alpha_1 \triangledown \alpha_2)\ () & = \alpha_1 \\
\tau_2(\alpha_1 \triangledown \alpha_2)\ (a, b) & = \textbf{if}\ p\ a\ \textbf{then}\ \alpha_2\ (a, b)\ \textbf{else}\ b
\end{array}
$$

Therefore, if we apply para-fold fusion we obtain the following:

$$tf\ p = (\!|\pi_1 \circ \tau(\langle [\,] \triangledown \phi_2, in_F \circ F\ \pi_2 \rangle) \circ F\ (id \times filter\ p)|\!)_F$$

Inlining,

$$
\begin{array}{l}
tf\ p\ [\,] = [\,] \\
tf\ p\ (a : as) = \textbf{if}\ p\ a\ \textbf{then}\ filter\ p\ as : tf\ p\ as\ \textbf{else}\ tf\ p\ as
\end{array}
$$

We applied fusion with the aim to eliminate the intermediate list that was generated by *filter*, but as result we obtained a function that filters the successive tails of the input list separately. This means that fusion transformed the composition of two functions with linear time behaviour to a function which is quadratic! In other words, in this case the effect of the medicine was worse than the illness itself. □

Example 4.6 This example shows another case of the situation presented in the previous example (we simply show the result of applying fusion and skip the details). Consider the function that counts the number of words of a text after having filtered it with a predicate p.

$$
\begin{aligned}
&wcf\ p = wc \circ filter\ p\\
&wc\ \ \ \ \ \ \ \ :: String \to Int\\
&wc\ [\,]\ \ \ \ \ \ = 0\\
&wc\ (c : cs) = \textbf{case}\ cs\ \textbf{of}\\
&\qquad\qquad [\,] \to \textbf{if}\ isSpace\ c\ \textbf{then}\ 0\ \textbf{else}\ 1\\
&\qquad\qquad d : ys \to \textbf{if}\ \neg\ (isSpace\ c) \wedge (isSpace\ d)\\
&\qquad\qquad\qquad\qquad \textbf{then}\ 1 + wc\ cs\\
&\qquad\qquad\qquad\qquad \textbf{else}\ wc\ cs
\end{aligned}
$$

Function wc is a paramorphism. It is inspired in one of the word counting algorithms described in [Gib06]. This function adds one each time the end of a word is detected, and for this it uses the current character c and the next one d (except at the end). By para-fold fusion we obtain as result a paramorphism with the following recursive definition:

$$
\begin{aligned}
&wcf\ p\ [\,]\ \ \ \ \ \ = 0\\
&wcf\ p\ (c : cs) = \textbf{if}\ p\ c\\
&\qquad\qquad \textbf{then case}\ filter\ p\ cs\ \textbf{of}\\
&\qquad\qquad\qquad\qquad [\,] \to \textbf{if}\ isSpace\ c\ \textbf{then}\ 0\ \textbf{else}\ 1\\
&\qquad\qquad\qquad\qquad d : ys \to \textbf{if}\ (\neg\ (isSpace\ c)) \wedge (isSpace\ d)\\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{then}\ 1 + wcf\ p\ cs\\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ wcf\ p\ cs\\
&\qquad\qquad \textbf{else}\ wcf\ p\ cs
\end{aligned}
$$

In the original definition of *wcf*, the inspection of the tail was performed on a text that was already filtered. Now, on the contrary, an on-line filtering of the tail is necessary each time before inspection. In this case the time behaviour of the resulting program is linear as the original ones. However, it may happen that the predicate p is applied twice to some of the elements of the input string: once in the context of *filter* and another one in the condition of the *if-then-else*. Also, note that the list nodes originally produced by *filter* are still produced when evaluating the *case* on *filter p cs*. So, in spite of our efforts, we could not eliminate the intermediate list. □

There exist of course applications of para-fold fusion that yield satisfactory results. This is illustrated by the following example.

Example 4.7 Consider the function *replace* given at the beginning of this chapter. This function is a paramorphism because it returns the tail of the input list as part of the result when the sought value is met.

$$
\begin{aligned}
&\mathit{replace}\ x\ y = \langle\!|[\,]\triangledown\phi_2|\!\rangle_F \\
&\quad \textbf{where}\ F = \overline{\mathbf{1}} + a \times I \\
&\qquad\quad \phi_2\ (a,(zs,as)) = \textbf{if}\ a == x\ \textbf{then}\ y : as\ \textbf{else}\ a : zs
\end{aligned}
$$

We can obtain the fused version of *repf x y p* presented at the beginning of this chapter by applying para-fold fusion.

In this case, *filter* needs to be applied to the sublist that remains after the replaced element (in case that element was found), as that sublist is returned as part of the result. □

Example 4.8 Consider the composition of the function that inserts a value in a binary search tree with the map function for binary trees.

$$
\begin{aligned}
&\textbf{data}\ \mathit{Tree}\ a = \mathit{Empty} \mid \mathit{Node}\ a\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a) \\
&\mathit{insmap}\ x\ f = \mathit{insert}\ x \circ \mathit{mapT}\ f \\
&\mathit{insert}\ x\ \mathit{Empty} = \mathit{Node}\ x\ \mathit{Empty}\ \mathit{Empty} \\
&\mathit{insert}\ x\ (\mathit{Node}\ a\ t1\ t2) = \textbf{if}\ x < a\ \textbf{then}\ \mathit{Node}\ a\ (\mathit{insert}\ x\ t1)\ t2 \\
&\qquad\qquad\qquad\qquad\qquad\qquad \textbf{else}\ \mathit{Node}\ a\ t1\ (\mathit{insert}\ x\ t2) \\
&\mathit{mapT}\ f\ \mathit{Empty} = \mathit{Empty} \\
&\mathit{mapT}\ f\ (\mathit{Node}\ a\ t1\ t2) = \mathit{Node}\ (f\ a)\ (\mathit{mapT}\ f\ t1)\ (\mathit{mapT}\ f\ t2)
\end{aligned}
$$

The application of para-fold fusion yields a satisfactory result in this case:

$$
\begin{aligned}
&\mathit{insmap}\ x\ f\ \mathit{Empty} = \mathit{Node}\ x\ \mathit{Empty}\ \mathit{Empty} \\
&\mathit{insmap}\ x\ f\ (\mathit{Node}\ a\ t1\ t2) = \\
&\qquad \textbf{if}\ x < f\ a\ \textbf{then}\ \mathit{Node}\ (f\ a)\ (\mathit{insmap}\ x\ f\ t1)\ (\mathit{mapT}\ f\ t2) \\
&\qquad\quad \textbf{else}\ \mathit{Node}\ (f\ a)\ (\mathit{mapT}\ f\ t1)\ (\mathit{insmap}\ x\ f\ t2)
\end{aligned}
$$

□

Note 4.9 The previous examples have shown the existence of some cases where para-fold fusion may worsen performance. These are fusions of the form $\langle\!|\phi|\!\rangle_F \circ f$ in which occurrences of f in the result produce the generation of duplicated computations. This means that, in the presence of paramorphisms, fusion cannot be applied without restrictions. It is necessary thus to include some code analysis that helps us to avoid the application of fusion in those cases we know performance will decrease. At the moment HFusion does not perform this kind of analysis, but we plan to do so in the future.

We give an intuitive characterization of the different cases of $\langle\!|\phi|\!\rangle_F \circ f$ in terms of the notion of "computation". The analysis focuses on function ϕ of the paramorphism:

- If during the computation of ϕ both the values returned by the recursive calls and their arguments are necessary, then fusion should be avoided. This is the case of *tails* $\circ$ *filter p* and *wc* $\circ$ *filter p*.

- If the values returned by the recursive calls or their arguments (but not both) appear during the computation of ϕ, then fusion can be safely performed. This is the case of *replace* x $y \circ$ *filter* p and *insert* $x \circ$ *mapT* f.

  For instance,

$$\begin{aligned} &\mathit{insert}\ x = (\!|\phi_1 \triangledown \phi_2|\!)_F \\ &\quad \textbf{where}\ F = \overline{1} + \overline{a} \times I \times I \\ &\qquad\quad \phi_1\ () = \mathit{Node}\ x\ \mathit{Empty}\ \mathit{Empty} \\ &\qquad\quad \phi_2\ (a, (r1, t1), (r2, t2)) = \textbf{if}\ x < a\ \textbf{then}\ \mathit{Node}\ a\ r1\ t2 \\ &\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ \mathit{Node}\ a\ t1\ r2 \end{aligned}$$

  If a computation uses *t1*, then it does not use *r1*, and vice-versa. The same holds for *t2* and *r2*. This is the reason that makes fusion in Example 4.8 adequate. □

4.2 Generalized paramorphisms

This section presents a new program scheme that generalizes paramorphisms in the same sense hylomorphisms generalize folds. This generalization of paramorphism will permit us to capture a wider class of recursive functions that use the arguments of the recursive calls to compute the final result. We will state fusion laws associated with generalized paramorphisms, but now in combination with folds, unfolds and hylomorphisms.

To see how this generalization is obtained, let us recall the diagram that a paramorphism satisfies, writing out_F instead of in_F:

$$\begin{array}{ccc} \mu F & \xrightarrow{\quad f \quad} & a \\ {\scriptstyle out_F}\downarrow & & \uparrow{\scriptstyle \phi} \\ F\mu F & \xrightarrow[F\langle f, id\rangle]{} & F(a \times \mu F) \end{array}$$

The arguments to the recursive calls are obtained by applying the coalgebra corresponding to the destructors of the data type. The generalization we introduce is obtained by considering an arbitrary coalgebra instead.

Given $\phi :: F(b \times a) \to b$ and a coalgebra $\psi :: a \to F\ a$, the *generalized paramorphism* $\{\!|\phi, \psi|\!\}_F :: a \to b$ is the least function that makes the following diagram commute:

$$\begin{array}{ccc} a & \xrightarrow{\quad f \quad} & b \\ {\scriptstyle \psi}\downarrow & & \uparrow{\scriptstyle \phi} \\ F\ a & \xrightarrow[F\langle f, id\rangle]{} & F(b \times a) \end{array}$$

The notion of generalized paramorphism is in some sense related with that of parametrically recursive coalgebra [CUV06].

Example 4.10 Consider the functor $F = \overline{\mathbf{1}} + \overline{a} \times I$ that captures the signature of lists. For $\phi_1 :: \mathbf{1} \to b$ and $\phi_2 :: a \times (c \times b) \to c$, the paramorphism $f = \{\!|\phi_1 \triangledown \phi_2, \psi|\!\}_F :: b \to c$ is the least function such that

$$\begin{array}{l} f\ b = case\ \psi\ b\ of \\ \qquad\qquad (1,()) \quad\ \to \phi_1 \\ \qquad\qquad (2,(a,b')) \to \phi_2(a,(f\ b',b')) \end{array}$$

□

The following equation expresses the fact that paramorphisms are a particular instance of generalized paramorphisms:

$$\langle\!|\phi|\!\rangle_F = \{\!|\phi, out_F|\!\}_F \tag{4.4}$$

Generalized paramorphisms are as expressive as hylomorphisms. The following equation shows that every hylomorphism can be written as a generalized paramorphism. It states a relationship similar to the one between folds and paramorphisms (equation 4.2).

$$[\![\phi,\psi]\!]_F = \{\!|\phi \circ F\,\pi_1, \psi|\!\}_F \tag{4.5}$$

The relationship in the other direction is the following. For each $\psi :: a \to Fa$, let us define the functor $G\,x = F(x \times a)$. Then,

$$\{\!|\phi,\psi|\!\}_F = [\![\phi, F\,\Delta \circ \psi]\!]_G \tag{4.6}$$

where $\Delta = \langle id, id\rangle$.

The following two fusion laws resemble laws for hylomorphisms. Observe that in (4.8) the colagebra homomorphism is internalized as part of the code of the resulting generalized paramorphism.

Proposition 4.11 (gpara fusion)

$$f\ strict\ \wedge\ f \circ \phi = \phi' \circ F(f \times id)\ \Rightarrow\ f \circ \{\!|\phi,\psi|\!\}_F = \{\!|\phi',\psi|\!\}_F \tag{4.7}$$

$$\psi \circ f = F\,f \circ \psi'\ \Rightarrow\ \{\!|\phi,\psi|\!\}_F \circ f = \{\!|\phi \circ F(id \times f), \psi'|\!\}_F \tag{4.8}$$

Proof Both laws can be proved by fixed point induction. We show the proof of (4.8) as it illustrates how f becomes part of the result. Let us define $\gamma(g) = \phi \circ F\langle g, id\rangle \circ \psi$ and $\gamma'(g) = \phi \circ F(id \times f) \circ F\langle g, id\rangle \circ \psi'$. The proof proceeds with predicate $g \circ f = g'$. The base case $\bot \circ f = \bot$ is immediate. Now, assume that $g \circ f = g'$. Then, $\gamma(g) \circ f = \phi \circ F\langle g, id\rangle \circ \psi \circ f =$

$\phi \circ F\langle g, id\rangle \circ Ff \circ \psi' = \phi \circ F\langle g \circ f, f\rangle \circ \psi' = \phi \circ F(id \times f) \circ \langle g', id\rangle \circ \psi' = \gamma'(g')$. Therefore, by fixed point induction it follows that $\mathsf{fix}(\gamma) \circ f = \mathsf{fix}(\gamma')$. □

Taking into account the close similarity between generalized paramorphisms and hylomorphisms, one may think of the existence of a factorization property similar to that of hylomorphism, which states that every generalized paramorphism can be split up into the composition of a paramorphism with an unfold, i.e. $\{\!|\phi, \psi|\!\}_F = (\!|\phi|\!)_F \circ [\!|\psi|\!]_F$. However, this law does not hold. The reason for the failure is originated in the fact that paramorphisms, in contrast to folds, use the arguments to the recursive calls to compute their results. The following law shows that the result of fusing the composition of a paramorphism with an unfold is a generalized paramorphism which internalizes the computation of the unfold as part of its code.

Proposition 4.12 (para-unfold fusion)

$$(\!|\phi|\!)_F \circ [\!|\psi|\!]_F = \{\!|\phi \circ F(id \times [\!|\psi|\!]_F), \psi|\!\}_F$$

Proof

$$\begin{array}{ll} & (\!|\phi|\!)_F \circ [\!|\psi|\!]_F \\ = & \quad \{\ (4.4)\ \} \\ & \{\!|\phi, out_F|\!\}_F \circ [\!|\psi|\!]_F \\ = & \quad \{\ (out_F \circ [\!|\psi|\!]_F = F\ [\!|\psi|\!]_F \circ \psi) \text{ and } (4.8)\ \} \\ & \{\!|\phi \circ F(id \times [\!|\psi|\!]_F), \psi|\!\}_F \end{array}$$

□

Example 4.13 Consider the following composition:

$$\begin{array}{l} tdown = tails \circ down \\ down \quad :: Int \to [Int] \\ down\ 0 = [\,] \\ down\ n = n : down\ (n-1) \end{array}$$

Function *tails* is a paramorphism while *down* is an unfold. By applying para-unfold fusion we obtain:

$$\begin{array}{l} tdown\ 0 = [\,] \\ tdown\ n = down\ (n-1) : tdown\ (n-1) \end{array}$$

This is again a situation in which the composition of two linear time functions gives a quadratic function as result. This is due to *tails*. □

The following law is a direct consequence of para-fold fusion (Proposition 4.4).

Proposition 4.14 (para-hylo fusion) *For strict ϕ,*

$$\tau :: \forall a.\, (F\ a \to a) \to (G\ a \to a) \Rightarrow$$
$$(\!|\phi|\!)_F \circ [\!|\tau(in_F), \psi|\!]_G = \{\!|\pi_1 \circ \tau(\langle \phi, in_F \circ F\pi_2\rangle) \circ G(id \times [\!|\tau(in_F), \psi|\!]_G), \psi|\!\}_G$$

Proof

$$\begin{aligned}
& (\!|\phi|\!)_F \circ [\!|\tau(in_F), \psi|\!]_G \\
= & \quad \{\ \text{hylo factorization}\ \} \\
& (\!|\phi|\!)_F \circ (\!|\tau(in_F)|\!)_G \circ [\!|\psi|\!]_G \\
= & \quad \{\ \text{para-fold fusion (Prop. 4.4)}\ \} \\
& (\!|\pi_1 \circ \tau(\langle \phi, in_F \circ F\pi_2\rangle) \circ G(id \times (\!|\tau(in_F)|\!)_G)|\!)_G \circ [\!|\psi|\!]_G \\
= & \quad \{\ \text{para-unfold fusion (Prop. 4.12)}\ \} \\
& \{\!|\pi_1 \circ \tau(\langle \phi, in_F \circ F\pi_2\rangle) \circ G(id \times (\!|\tau(in_F)|\!)_G) \circ G(id \times [\!|\psi|\!]_G), \psi|\!\}_G \\
= & \quad \{\ \text{functor } G \text{ and hylo factorization}\ \} \\
& \{\!|\pi_1 \circ \tau(\langle \phi, in_F \circ F\pi_2\rangle) \circ G(id \times [\!|\tau(in_F), \psi|\!]_G), \psi|\!\}_G
\end{aligned}$$

□

The two previous fusion laws show compositions that yield generalized paramorphisms as result. The laws that follow are acid rain laws with generalized paramorphism as arguments.

Proposition 4.15 (fold-gpara fusion) *Let $\psi : b \to G\ b$. For strict ϕ,*

$$\begin{aligned}
& \tau :: \forall a.\, (F\ a \to a) \to (G\ (a \times b) \to a)\ \Rightarrow \\
& (\!|\phi|\!)_F \circ \{\!|\tau(in_F), \psi|\!\}_G = \{\!|\tau(\phi), \psi|\!\}_G
\end{aligned}$$

Proof Similar proof to fold-para fusion (Prop. 4.2), but using (4.7) instead.□

The generalization of paramorphism opens the possibility of an acid rain law with unfold.

Proposition 4.16 (gpara-unfold fusion)

$$\begin{aligned}
& \sigma :: (a \to F\ a) \to (a \to G\ a)\ \Rightarrow \\
& \{\!|\phi, \sigma(out_F)|\!\}_G \circ [\!|\psi|\!]_F = \{\!|\phi \circ G(id \times [\!|\psi|\!]_F), \sigma(\psi)|\!\}_G
\end{aligned}$$

Proof Same proof to fold-para fusion (Prop. 4.2), but using (4.8) and the free theorem for σ. □

Example 4.17 Consider the following composition:

$$
\begin{aligned}
&dm\ p\ f = drop2While\ p \circ map\ f\\
&drop2While :: (a \to Bool) \to [a] \to [a]\\
&drop2While\ p\ [\,] \ \ = [\,]\\
&drop2While\ p\ [a] = \textbf{if}\ p\ a\ \textbf{then}\ [\,]\ \textbf{else}\ [a]\\
&drop2While\ p\ (a : a' : as) = \textbf{if}\ p\ a\ \textbf{then}\ drop2While\ p\ as\ \textbf{else}\ a : a' : as
\end{aligned}
$$

Function *drop2While* can be defined as a generalized paramorphism.

$$
\begin{aligned}
&drop2While\ p = \{\!|[\,]\triangledown\phi_2\triangledown\phi_3, \psi|\!\}_G\\
&\quad \textbf{where}\ G = \overline{\mathbf{1}} + \overline{a} + \overline{a} \times \overline{a} \times I\\
&\qquad\quad \phi_2\ a = \textbf{if}\ p\ a\ \textbf{then}\ [\,]\ \textbf{else}\ [a]\\
&\qquad\quad \phi_3\ (a, a', (ys, as)) = \textbf{if}\ p\ a\ \textbf{then}\ ys\ \textbf{else}\ a : a' : as\\
&\qquad\quad \psi\ [\,] = (1, ())\\
&\qquad\quad \psi\ [a] = (2, a)\\
&\qquad\quad \psi\ (a : a' : as) = (3, (a, a', as))
\end{aligned}
$$

The coalgebra ψ does not correspond to out_F, for $F = \overline{\mathbf{1}} + \overline{a} \times I$ the base functor of lists. It can, however, be written as $\psi = \sigma(out_F)$, where σ is given by:

$$
\begin{aligned}
&\sigma :: (b \to F\ b) \to (b \to G\ b)\\
&\sigma(\beta)\ b = \textbf{case}\ \beta\ b\ \textbf{of}\\
&\qquad\qquad (1, ()) \qquad \to (1, ())\\
&\qquad\qquad (2, (a, b')) \to \textbf{case}\ \beta\ b'\ \textbf{of}\\
&\qquad\qquad\qquad\qquad\qquad (1, ()) \qquad \to (2, a)\\
&\qquad\qquad\qquad\qquad\qquad (2, (a', b'')) \to (3, (a, a', b''))
\end{aligned}
$$

On the other hand, *map*, which is usually presentd as a fold over lists, can be expressed as an unfold as well (see Example 3.2):

$$
\begin{aligned}
&map\ f = [\!|\psi|\!]_F\\
&\quad \textbf{where}\ \psi\ [\,] = (1, ())\\
&\qquad\quad \psi\ (x : xs) = (2, (f\ x, xs))
\end{aligned}
$$

Therefore, we can apply gpara-unfold fusion, obtaining

$$dm\ p\ f = \{\!|[\,]\triangledown\phi_2\triangledown(\phi_3 \circ (id \times id \times (id \times map\ f))), \sigma(\psi)|\!\}_G$$

Inlining,

$$
\begin{aligned}
&dm\ p\ f\ [\,] \qquad = [\,]\\
&dm\ p\ f\ (a : as) = \textbf{let}\ fa = f\ a
\end{aligned}
$$

$$
\begin{array}{l}
\textbf{in case } as \textbf{ of} \\
\quad [\,] \qquad \rightarrow \textbf{if } p\ fa \textbf{ then } [\,] \textbf{ else } [fa] \\
\quad (a', xs) \rightarrow \textbf{if } p\ fa \textbf{ then } dm\ p\ f\ xs \\
\qquad\qquad\qquad\quad \textbf{else } fa : f\ a' : map\ f\ xs
\end{array}
$$

Fusion in this case is completely satisfactory. □

And now we show a law that relates paramorphisms with generalized paramorphisms.

Proposition 4.18 (para-gpara fusion) *Let $\psi : b \rightarrow G\ b$. For strict ϕ,*

$$\tau :: \forall a.\ (F\ a \rightarrow a) \rightarrow (G\ (a \times b) \rightarrow a) \Rightarrow$$
$$(\!|\phi|\!)_F \circ \{\!|\tau(in_F), \psi|\!\}_G = \{\!|\pi_1 \circ \tau(\langle \phi, in_F \circ F\pi_2 \rangle) \circ G\langle id \times \{\!|\tau(in_F), \psi|\!\}_G, \pi_2 \rangle, \psi|\!\}_G$$

Proof

$$
\begin{array}{ll}
 & (\!|\phi|\!)_F \circ \{\!|\tau(in_F), \psi|\!\}_G \\
= & \qquad \{\ (4.6),\ H\ x = G(x \times b)\ \} \\
 & (\!|\phi|\!)_F \circ [\!|\tau(in_F), G\Delta \circ \psi|\!]_H \\
= & \qquad \{\ \text{Prop. 4.14 and def. of } H\ \} \\
 & \{\!|\pi_1 \circ \tau(\langle \phi, in_F \circ F\pi_2 \rangle) \circ G((id \times \{\!|\tau(in_F), \psi|\!\}_G) \times id), G\Delta \circ \psi|\!\}_H \\
= & \qquad \{\ \text{product manipulation}\ \} \\
 & \{\!|\pi_1 \circ \tau(\langle \phi, in_F \circ F\pi_2 \rangle) \circ G\langle id \times \{\!|\tau(in_F), \psi|\!\}_G, \pi_2 \rangle, \psi|\!\}_G
\end{array}
$$

□

Corollary 4.19 (para-para fusion) *For strict ϕ,*

$$\tau :: \forall a.\ (F\ a \rightarrow a) \rightarrow (G\ (a \times \mu G) \rightarrow a) \Rightarrow$$
$$(\!|\phi|\!)_F \circ (\!|\tau(in_F)|\!)_G = (\!|\pi_1 \circ \tau(\langle \phi, in_F \circ F\pi_2 \rangle) \circ G\langle id \times (\!|\tau(in_F)|\!)_G, \pi_2 \rangle|\!)_G$$

Example 4.20 Using para-para fusion we can transform

$$
\begin{array}{l}
dWt\ p = dropWhile\ p \circ tails \\
dropWhile :: (a \rightarrow Bool) \rightarrow [a] \rightarrow [a] \\
dropWhile\ p\ [\,] = [\,] \\
dropWhile\ p\ (a : as) = \textbf{if } p\ a \textbf{ then } dropWhile\ p\ as \textbf{ else } a : as
\end{array}
$$

into

$$
\begin{array}{l}
dWt\ p\ [\,] = [\,] \\
dWt\ p\ (a : as) = \textbf{if } p\ as \textbf{ then } dWt\ p\ as \textbf{ else } as : tails\ as
\end{array}
$$

□

Note 4.21 The characterization of good and bad cases of fusion that we can add with the introduction of generalized paramorphism is very the same as the one presented in Note 4.9. Now we must analyze function ϕ in compositions of the form $\{\!|\phi, \psi|\!\}_F \circ f$ and $\langle\!|\phi|\!\rangle_F \circ f$ in order to conclude whether fusion is desirable or not. Performing such an analysis we can conclude, for instance, that *tails* $\circ$ *down* is a bad case while *drop2While* $p \circ$ *map* f and *dropWhile* $p \circ$ *tails* are good ones. □

4.3 Fusion in practice

Our interest in studying generalized paramorphisms has arisen in the context of the development of HFusion. During the implementation of the kernel of the tool we started experimenting with some examples that were fusable by our implementation (modulo some simple modifications to the internal representation of hylomorphisms), but were impossible to be fused with the original representation and laws. We wanted then to give an explanation of these modifications at the abstract level, and it was during that process that the notion of generalized paramorphism came up as the appropriate abstraction that reflects the class of special cases we were playing with. With these modifications, the tool essentially interprets every recursive function as a generalized paramorphism. The laws presented in this chapter allow reusing the different derivation algorithms we have seen in Chapter 3.

The equivalence in the expressive power between hylomorphism and generalized paramorphism (witnessed by equations (4.5) and (4.6)) permits us to assure that we are not losing fusion cases with the introduction of generalized paramorphism. On the contrary, we gain new cases captured by para-hylo fusion, like the one shown in Example 4.7.

Note that, in general, paramorphisms deforest very little when compared with the traditional hylo deforestation. For instance, in the case of *insert* $x \circ$ *mapT* f, fusion deforests just a single path from the root to the leaves. This is due to the fact that a paramorphism not only traverses its input, but also keeps it for computing the outcome. So in that case only a small amount of the intermediate data structure was eliminated. Nonetheless, fusion with paramorphisms may be useful for bringing other functions together. For example, after fusing *map* $g \circ$ *replace* x $y \circ$ *filter* q the composition *map* $g \circ$ *filter* q will appear in the body of the resulting function, representing a residual case where fusion can be applied again. See Section 8.4 for a practical test of the fact that paramorphism fusion may improve programs.

4.4 Summary

In this chapter we have shown how fusion of primitive recursive functions can be achieved by extending the acid rain laws.

We have started by presenting the laws for paramorphisms, which enable fusion of some cases we could not handle without them. Later, we proposed generalized paramorphisms, a convenient representation to handle paramorphism-like hylos, which allow to reuse the algorithms for deriving transformers τ and σ. Then, we proceeded to show our extensions of acid rain laws for generalized paramorphisms.

We have discussed how fusion can worsen a program like $tails \circ map\ f$, and we have seen how we can reasonably rule out those cases with a simple and decidable criterion.

Finally, we have made a case for paramorphisms fusion as a means for bringing function calls together, which constitutes an opportunity for applying fusion again.

Chapter 5

Partial deforestation

When one looks at fusion with paramorphism, it's outstanding that only part of the intermediate data structure is deforested. This partial deforestation is not exclusive of paramorphisms but also of some other definitions.

In the first part of this chapter we will show how some fusion cases with paramorphisms can be fused with the normal fusion laws by rewriting the involved definitions. In the second part, we will present some examples of fusions that result in partial deforestation and that cannot be obtained using any of the laws we have seen so far unless we rewrite the definitions with new techniques to be described.

5.1 A disguised paramorphism

Consider the following definitions

$$
\begin{aligned}
&\textbf{data}\ \mathit{Tree}\ a = \mathit{Empty} \mid \mathit{Node}\ a\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a)\\
&\mathit{mirror} :: \mathit{Tree}\ a \to \mathit{Tree}\ a\\
&\mathit{mirror}\ \mathit{Empty} = \mathit{Empty}\\
&\mathit{mirror}\ (\mathit{Node}\ a\ l\ r) = \mathit{Node}\ a\ (\mathit{mirror}\ r)\ (\mathit{mirror}\ l)\\
&\mathit{mapl} :: (a \to a) \to \mathit{Tree}\ a \to \mathit{Tree}\ a\\
&\mathit{mapl}\ f\ \mathit{Empty} = \mathit{Empty}\\
&\mathit{mapl}\ f\ (\mathit{Node}\ a\ l\ r) = \mathit{Node}\ (f\ a)\ (\mathit{mapl}\ f\ l)\ r
\end{aligned}
$$

Let's suppose that we want to fuse $\mathit{mapl}\ f \circ \mathit{mirror}$. If we write the above definitions as hylomorphisms we get:

$$
\begin{aligned}
&\mathit{mirror} = [\![\psi]\!]_F\\
&\quad \textbf{where}\ F = \overline{\mathbf{1}} + \overline{a} \times I \times I\\
&\qquad\quad \psi\ \mathit{Empty} = (1, ())\\
&\qquad\quad \psi\ (\mathit{Node}\ a\ l\ r) = (2, (a, r, l))\\
&\mathit{mapl}\ f = (\!|\phi|\!)_{F'}
\end{aligned}
$$

$$\begin{aligned}&\textbf{where } F' = \overline{\mathbf{1}} + \overline{a} \times I \times \overline{\mathit{Tree}\ a}\\ &\qquad \phi = \mathit{Empty}\triangledown(\lambda(a, \mathit{v1}, r) \to \mathit{Node}\ (f\ a)\ \mathit{v1}\ r)\end{aligned}$$

Note that both hylomorphisms traverse a structure of type *Tree a*, nonetheless, their functors do not match. But they do if we rewrite *mirror* like this:

$$\begin{aligned}&\mathit{mirror} = [\![\psi]\!]_F\\ &\quad \textbf{where } F = \overline{\mathbf{1}} + \overline{a} \times I \times \overline{\mathit{Tree}\ a}\\ &\qquad \psi\ \mathit{Empty} = (1, ())\\ &\qquad \psi\ (\mathit{Node}\ a\ l\ r) = (2, (a, r, \mathit{mirror}\ l))\end{aligned}$$

Note that we have disguised one of the recursive calls. By doing so, we are making explicit that we won't deforest the produced intermediate structure over that branch. Now $F = F'$, and the fold-unfold law (Theorem 2.7) yields $[\![\phi, \psi]\!]_F$, which expands to:

$$\begin{aligned}&\mathit{maplmirror} :: (a \to a) \to \mathit{Tree}\ a \to \mathit{Tree}\ a\\ &\mathit{maplmirror}\ f\ \mathit{Empty} = \mathit{Empty}\\ &\mathit{maplmirror}\ f\ (\mathit{Node}\ a\ l\ r) = \mathit{Node}\ (f\ a)\ (\mathit{maplmirror}\ f\ r)\ (\mathit{mirror}\ l)\end{aligned}$$

Alternatively, we could reach fusion with Proposition 4.12 (para-unfold law) if we rewrite *mapl* as a paramorphism.

$$\begin{aligned}&\mathit{mapl}\ f = (\!|\phi|\!)_{F'}\\ &\quad \textbf{where } F' = \overline{\mathbf{1}} + \overline{a} \times I \times I\\ &\qquad \phi = \mathit{Empty}\triangledown(\lambda(a, (\mathit{v1}, \_), (\_, r)) \to \mathit{Node}\ (f\ a)\ \mathit{v1}\ r)\end{aligned}$$

Then the result of the fusion would be $\{\!|\phi \circ F'\ (\mathit{mirror} \times \mathit{id}), \psi|\!\}_{F'}$, which expands to exactly the same definition of *maplmirror*.

Looking at the outcome of fusion, it is easy to see why the call *mirror l* is not eliminated. It is because that branch is not traversed by *mapl f*, it is copied from the input as is. Therefore, a partial deforestation is unavoidable. In the following section we will turn to an example which does not allow fusion through paramorphisms.

5.2 Partial deforestation without paras

Consider the following definition

$$\begin{aligned}&\mathit{leaves} :: \mathit{Tree}\ a \to [a]\\ &\mathit{leaves}\ \mathit{Empty} = [\,]\\ &\mathit{leaves}\ (\mathit{Node}\ a\ \mathit{Empty}\ \mathit{Empty}) = [a]\\ &\mathit{leaves}\ (\mathit{Node}\ a\ l\ r) = \mathit{leaves}\ l +\!\!+ \mathit{leaves}\ r\end{aligned}$$

Let's say we want to fuse $leaves \circ mapl\ f$. We will use the following hylomorphism definitions.

$$
\begin{array}{l}
leaves = [\![[\,] \triangledown (:[\,]) \triangledown (+\!\!+), \psi']\!]_G \\
\quad \textbf{where}\ G = \overline{\mathbf{1}} + \overline{a} + I \times I \\
\qquad\quad \psi'\ Empty = (1, ()) \\
\qquad\quad \psi'\ (Node\ a\ Empty\ Empty) = (2, a) \\
\qquad\quad \psi'\ (Node\ a\ l\ r) = (3, (l, r)) \\
mapl\ f = [\!(\psi)\!]_{F'} \\
\quad \textbf{where}\ F' = \overline{\mathbf{1}} + \overline{a} \times I \times \overline{Tree\ a} \\
\qquad\quad \psi\ Empty = (1, ()) \\
\qquad\quad \psi\ (Node\ a\ l\ r) = (2, (f\ a, l, r))
\end{array}
$$

To apply fusion we have to derive σ from coalgebra ψ' because of the nested pattern *Node a Empty Empty*, obtaining

$$
\begin{array}{l}
leaves = [\![[\,] \triangledown (:[\,]) \triangledown (+\!\!+), \sigma(out_F)]\!]_G \\
\quad \textbf{where}\ G = \overline{\mathbf{1}} + \overline{a} + I \times I \\
\qquad\quad F = \overline{\mathbf{1}} + \overline{a} \times I \times I \\
\qquad\quad \sigma(\beta) = \lambda t \rightarrow \textbf{case}\ t\ \textbf{of} \\
\qquad\qquad\qquad\quad \beta{\cdot}(1, ()) \rightarrow (1, ()) \\
\qquad\qquad\qquad\quad \beta{\cdot}(2, (a, \beta{\cdot}(1, ()), \beta{\cdot}(1, ()))) \rightarrow (2, a) \\
\qquad\qquad\qquad\quad \beta{\cdot}(2, (a, t1, t2)) \rightarrow (3, (t1, t2))
\end{array}
$$

However, we cannot fuse the composition using the hylo-unfold law because the functor F of *leaves* is different from the functor F' of $mapl\ f$. And using the paramorphism representation of $mapl\ f$ does not help reaching fusion either.

The only alternative we have is disguising recursive calls in *leaves* to get

$$
\begin{array}{l}
leaves = [\![[\,] \triangledown (:[\,]) \triangledown \phi_3, \sigma(out_F)]\!]_G \\
\quad \textbf{where}\ G = \overline{\mathbf{1}} + \overline{a} + I \times \overline{a} \\
\qquad\quad F = \overline{\mathbf{1}} + \overline{a} \times I \times \overline{a} \\
\qquad\quad \sigma(\beta) = \lambda t \rightarrow \textbf{case}\ t\ \textbf{of} \\
\qquad\qquad\qquad\quad \beta{\cdot}(1, ()) \rightarrow (1, ()) \\
\qquad\qquad\qquad\quad \beta{\cdot}(2, (a, \beta{\cdot}(1, ()), Empty)) \rightarrow (2, a) \\
\qquad\qquad\qquad\quad \beta{\cdot}(2, (a, t1, t2)) \rightarrow (3, (t1, t2)) \\
\qquad\quad \phi_3 = \lambda(v1, t2) \rightarrow v1 +\!\!+ leaves\ t2
\end{array}
$$

Now, the hylo-unfold fusion yields $[\![[\,] \triangledown (:[\,]) \triangledown \phi_3, \sigma(\psi)]\!]_G$ which inlines to

$$
\begin{array}{l}
leavesmapl :: (a \rightarrow a) \rightarrow Tree\ a \rightarrow [a] \\
leavesmapl\ f\ Empty = [\,] \\
leavesmapl\ f\ (Node\ a\ Empty\ Empty) = [f\ a] \\
leavesmapl\ f\ (Node\ a\ l\ r) = leavesmapl\ f\ l +\!\!+ leaves\ r
\end{array}
$$

This example showed that partial deforestation is not constrained to the realm of paramorphisms, and that an extra technique to disguise the recursive calls is necessary. It is not difficult to come up with an algorithm that deduces which calls need to be disguised by comparing the functors of *leaves* and *mapl f*. In essence, the functors will differ only in some positions: where one functor shows I the other may show a constant functor ($\overline{Tree\ a}$ in the case of *mapl f*). A difference like that identifies one or more recursive calls that need to be disguised. HFusion is employing such an algorithm.

5.3 Multiple occurrences of a constructor

In this section we will see another example of partial deforestation which cannot be fused with the laws for paramorphisms presented in Chapter 4.

Consider the definitions of *drop2While* and *replace* presented in Chapter 4, and suppose that we want to fuse *drop2While* $p \circ$ *replace* e. The following are the hylo representations:

$$
\begin{aligned}
&\mathit{drop2While}\ p = \{\!|\, [\,] \triangledown \phi_2 \triangledown \phi_3, \psi \,|\!\}_G \\
&\quad \textbf{where}\ G = \overline{\mathbf{1}} + \overline{a} + \overline{a} \times \overline{a} \times I \\
&\qquad\quad \phi_2 = \lambda a \rightarrow \textbf{if}\ p\ a\ \textbf{then}\ [\,]\ \textbf{else}\ [a] \\
&\qquad\quad \phi_3 = \lambda(a, a', (ys, as)) \rightarrow \textbf{if}\ p\ a\ \textbf{then}\ ys\ \textbf{else}\ a : a' : as \\
&\qquad\quad \psi' = \lambda as \rightarrow \textbf{case}\ as\ \textbf{of} \\
&\qquad\qquad\qquad\qquad [\,] \qquad\quad \rightarrow (1, ()) \\
&\qquad\qquad\qquad\qquad a : [\,] \qquad \rightarrow (2, a) \\
&\qquad\qquad\qquad\qquad a : a' : as \rightarrow (3, (a, a', as)) \\
&\mathit{replace}\ e = [\![\, [\,] \triangledown (:) \triangledown (:), \psi \,]\!]_{F'} \\
&\quad \textbf{where}\ F' = \overline{\mathbf{1}} + \overline{a} \times \overline{[a]} + \overline{a} \times I \\
&\qquad\quad \psi\ [\,] \qquad = (1, ()) \\
&\qquad\quad \psi\ (a, as) = \textbf{if}\ a == e\ \textbf{then}\ (2, (e, as)) \\
&\qquad\qquad\qquad\qquad\ \ \textbf{else}\ (3, (a, as))
\end{aligned}
$$

Note that *replace e* is not an unfold because of the duplicated occurrence of constructor (:). On the other hand *drop2While p* is not a fold because of the nested constructors in the coalgebra patterns. Therefore, we can not apply any of the acid rain laws. However, let's imagine for a minute that we extend the type of lists with an extra constructor $(\triangleright) :: a \rightarrow [a] \rightarrow [a]$ and rewrite the hylos as follows:

$$
\begin{aligned}
&\mathit{drop2While}\ p = \{\!|\, [\,] \triangledown \phi_2 \triangledown \phi_3 \triangledown \phi_4 \triangledown \phi_5 \triangledown \phi_6 \triangledown \phi_7, \psi \,|\!\}_G \\
&\quad \textbf{where}\ G = \overline{\mathbf{1}} + \overline{a} + \overline{a} + \overline{a} \times \overline{a} \times I + \overline{a} \times \overline{a} \times I + \overline{a} \times \overline{a} \times I + \overline{a} \times \overline{a} \times I \\
&\qquad\quad \phi_2 = \lambda a \rightarrow \textbf{if}\ p\ a\ \textbf{then}\ [\,]\ \textbf{else}\ [a] \\
&\qquad\quad \phi_3 = \lambda a \rightarrow \textbf{if}\ p\ a\ \textbf{then}\ [\,]\ \textbf{else}\ [a] \\
&\qquad\quad \phi_4 = \lambda(a, a', (ys, as)) \rightarrow \textbf{if}\ p\ a\ \textbf{then}\ ys\ \textbf{else}\ a : a' : as \\
&\qquad\quad \phi_5 = \lambda(a, a', (ys, as)) \rightarrow \textbf{if}\ p\ a\ \textbf{then}\ ys\ \textbf{else}\ a : a' : as
\end{aligned}
$$

$$
\begin{aligned}
&\phi_6 = \lambda(a, a', (ys, as)) \to \textbf{if } p\ a \textbf{ then } ys \textbf{ else } a : a' : as\\
&\phi_7 = \lambda(a, a', (ys, as)) \to \textbf{if } p\ a \textbf{ then } ys \textbf{ else } a : a' : as\\
&\psi' = \lambda as \to \textbf{case } as \textbf{ of}\\
&\qquad\qquad [\,] \qquad\quad \to (1, ())\\
&\qquad\qquad a \triangleright [\,] \qquad \to (2, a)\\
&\qquad\qquad a : [\,] \qquad \to (3, a)\\
&\qquad\qquad a : a' \triangleright as \ \to (4, (a, a', as))\\
&\qquad\qquad a : a' : as \ \to (5, (a, a', as))\\
&\qquad\qquad a \triangleright a' : as \ \to (6, (a, a', as))\\
&\qquad\qquad a \triangleright a' \triangleright as \to (7, (a, a', as))\\
&\mathit{replace}\ e = [\![[\,] \triangledown (\triangleright) \triangledown (:), \psi]\!]_{F'}
\end{aligned}
$$

By introducing a new constructor ($\triangleright$) we can see *replace e* as an unfold. We also extended the definition of *drop2While* to handle ($\triangleright$) in the same way as (:) by duplicating cases. Now, if we derive σ from ψ we get:

$$
\begin{aligned}
&\mathit{drop2While}\ p = \{\!| [\,] \triangledown \phi_2 \triangledown \phi_3 \triangledown \phi_4 \triangledown \phi_5 \triangledown \phi_6 \triangledown \phi_7, \sigma(\mathit{out}_F) |\!\}_G\\
&\quad \textbf{where } F = \overline{\mathbf{1}} + \overline{a} \times I + \overline{a} \times I\\
&\qquad \sigma(\beta) = \lambda as \to \textbf{case } as \textbf{ of}\\
&\qquad\qquad \beta{\cdot}(1, ()) \qquad\qquad\quad \to (1, ())\\
&\qquad\qquad \beta{\cdot}(2, (a, \beta{\cdot}(1, ()))) \qquad \to (2, a)\\
&\qquad\qquad \beta{\cdot}(3, (a, \beta{\cdot}(1, ()))) \qquad \to (3, a)\\
&\qquad\qquad \beta{\cdot}(2, (a, \beta{\cdot}(2, (a', as)))) \to (4, (a, a', as))\\
&\qquad\qquad \beta{\cdot}(2, (a, \beta{\cdot}(3, (a', as)))) \to (5, (a, a', as))\\
&\qquad\qquad \beta{\cdot}(3, (a, \beta{\cdot}(2, (a', as)))) \to (6, (a, a', as))\\
&\qquad\qquad \beta{\cdot}(3, (a, \beta{\cdot}(3, (a', as)))) \to (7, (a, a', as))
\end{aligned}
$$

Unfortunately this σ is of no use because the functor F is different from the functor F' of *replace e*. However, now we can resort to the disguising calls trick:

$$
\begin{aligned}
&\mathit{drop2While}\ p = \{\!| [\,] \triangledown \phi_2 \triangledown \phi_3 \triangledown \phi_4 \triangledown \phi_5 \triangledown \phi_6 \triangledown \phi_7, \sigma(\mathit{out}_F) |\!\}_G\\
&\quad \textbf{where } F = \overline{\mathbf{1}} + \overline{a} \times \overline{[a]} + \overline{a} \times I\\
&\qquad G = \overline{\mathbf{1}} + \overline{a} + \overline{a} + \overline{a} \times \overline{a} \times \overline{[a]} + \overline{a} \times \overline{a} \times \overline{[a]}\\
&\qquad\qquad\qquad + \overline{a} \times \overline{a} \times \overline{[a]} + \overline{a} \times \overline{a} \times I\\
&\qquad \phi_4 = \lambda(a, a', as) \to \textbf{if } p\ a \textbf{ then } \mathit{drop2While}\ p\ as \textbf{ else } a : a' : as\\
&\qquad \phi_5 = \lambda(a, a', as) \to \textbf{if } p\ a \textbf{ then } \mathit{drop2While}\ p\ as \textbf{ else } a : a' : as\\
&\qquad \phi_6 = \lambda(a, a', as) \to \textbf{if } p\ a \textbf{ then } \mathit{drop2While}\ p\ as \textbf{ else } a : a' : as\\
&\qquad \phi_7 = \lambda(a, a', (ys, as)) \to \textbf{if } p\ a \textbf{ then } ys \textbf{ else } a : a' : as\\
&\qquad \sigma(\beta) = \lambda as \to \textbf{case } as \textbf{ of}\\
&\qquad\qquad \beta{\cdot}(1, ()) \qquad\qquad\quad \to (1, ())\\
&\qquad\qquad \beta{\cdot}(2, (a, [\,])) \qquad\quad \to (2, a)\\
&\qquad\qquad \beta{\cdot}(3, (a, \beta{\cdot}(1, ()))) \qquad \to (3, a)\\
&\qquad\qquad \beta{\cdot}(2, (a, a' \triangleright as)) \qquad \to (4, (a, a', as))\\
&\qquad\qquad \beta{\cdot}(2, (a, a' : as))) \qquad\quad \to (5, (a, a', as))\\
&\qquad\qquad \beta{\cdot}(3, (a, \beta{\cdot}(2, (a', as)))) \to (6, (a, a', as))\\
&\qquad\qquad \beta{\cdot}(3, (a, \beta{\cdot}(3, (a', as)))) \to (7, (a, a', as))
\end{aligned}
$$

Doing so, now we have that $F = F'$, and therefore we can proceed with the fusion of *drop2While p* $\circ$ *replace e*, yielding

$$\{\!|\, [\,] \triangledown \phi_2 \triangledown \phi_3 \triangledown \phi_4 \triangledown \phi_5 \triangledown \phi_6 \triangledown \phi_7, \sigma(\psi) \,|\!\}_G$$

We could still cut some cases from the last version of *drop2While p* in order to make it more readable.

$$\begin{array}{l}
\mathit{drop2While}\ p = \{\!|\, [\,] \triangledown \phi_2 \triangledown \phi_3 \triangledown \phi_4, \sigma(\mathit{out}_F) \,|\!\}_G \\
\quad \textbf{where}\ G = \overline{\mathbf{1}} + \overline{a} + \overline{a} + \overline{a} \times \overline{a} \times \overline{[a]} + \overline{a} \times \overline{a} \times I \\
\qquad \phi_2 = \lambda a \to \textbf{if}\ p\ a\ \textbf{then}\ [\,]\ \textbf{else}\ [a] \\
\qquad \phi_3 = \lambda(a, a', as) \to \textbf{if}\ p\ a\ \textbf{then}\ \mathit{drop2While}\ p\ as\ \textbf{else}\ a : a' : as \\
\qquad \phi_4 = \lambda(a, a', (ys, as)) \to \textbf{if}\ p\ a\ \textbf{then}\ ys\ \textbf{else}\ a : a' : as \\
\qquad \sigma(\beta) = \lambda as \to \textbf{case}\ as\ \textbf{of} \\
\qquad\qquad \begin{array}{ll}
\beta{\cdot}(1, ()) & \to (1, ()) \\
\beta{\cdot}(2, (a, [\,])) & \to (2, a) \\
\beta{\cdot}(3, (a, \beta{\cdot}(1, ()))) & \to (2, a) \\
\beta{\cdot}(2, (a, a' : as)) & \to (3, (a, a', as)) \\
\beta{\cdot}(2, (a, a' \triangleright as))) & \to (3, (a, a', as)) \\
\beta{\cdot}(3, (a, \beta{\cdot}(2, (a', as)))) & \to (3, (a, a', as)) \\
\beta{\cdot}(3, (a, \beta{\cdot}(3, (a', as)))) & \to (4, (a, a', as))
\end{array}
\end{array}$$

We could dispense also of the coalgebra alternative that uses our artificial constructor ($\triangleright$). That is because only in_F produces that constructor, and the positions where the constructor is matched will never hold the result of calling in_F.

$$\begin{array}{l}
\sigma(\beta) = \lambda as \to \textbf{case}\ as\ \textbf{of} \\
\qquad \begin{array}{ll}
\beta{\cdot}(1, ()) & \to (1, ()) \\
\beta{\cdot}(2, (a, [\,])) & \to (2, a) \\
\beta{\cdot}(3, (a, \beta{\cdot}(1, ()))) & \to (2, a) \\
\beta{\cdot}(2, (a, a' : as)) & \to (3, (a, a', as)) \\
\beta{\cdot}(3, (a, \beta{\cdot}(2, (a', as)))) & \to (3, (a, a', as)) \\
\beta{\cdot}(3, (a, \beta{\cdot}(3, (a', as)))) & \to (4, (a, a', as))
\end{array}
\end{array}$$

Inlining $\{\!|\, [\,] \triangledown \phi_2 \triangledown \phi_3 \triangledown \phi_4, \sigma(\psi) \,|\!\}_G$ we get

$$\begin{array}{l}
\mathit{d2WRep}\ p\ e\ [\,] = [\,] \\
\mathit{d2WRep}\ p\ e\ (a : as) = \textbf{if}\ a == e\ \textbf{then} \\
\qquad\qquad \textbf{case}\ as\ \textbf{of} \\
\qquad\qquad\quad [\,] \qquad \to \textbf{if}\ p\ e\ \textbf{then}\ [\,] \\
\qquad\qquad\qquad\qquad\quad \textbf{else}\ [e] \\
\qquad\qquad\quad a' : ass \to \textbf{if}\ p\ e\ \textbf{then}\ \mathit{drop2While}\ p\ ass \\
\qquad\qquad\qquad\qquad\quad \textbf{else}\ e : a' : ass \\
\qquad\qquad \textbf{else case}\ as\ \textbf{of} \\
\qquad\qquad\qquad [\,] \qquad \to \textbf{if}\ p\ a\ \textbf{then}\ [\,] \\
\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ [a]
\end{array}$$

$$
\begin{aligned}
a' : ass \rightarrow \textbf{if } a' == e \textbf{ then}& \\
\textbf{if } p\ a \textbf{ then } &drop2While\ p\ ass \\
\textbf{else } a : e : &ass \\
\textbf{else if } p\ a \textbf{ then } &d2WRep\ p\ e\ ass \\
\textbf{else } a : a' : &replace\ e\ ass
\end{aligned}
$$

Note that the auxiliary constructor ($\triangleright$) is no longer used in the final result. As it was abstracted from the coalgebra, when we dispense of in_F it disapears completely.

These steps can be automated, first by searching in the almost in_F whose constructors are duplicated, and then by introducing a new constructor for each repetition. If the functors for all the repetitions of a constructor were the same, it may be possible to restructure the hylomorphism to get an unfold without adding artificial constructors. Note, however, that this is not our situation, because the functors are $\overline{a} \times \overline{[a]}$ and $\overline{a} \times I$ for the respective occurences of constructor (:) in algebra $[\,]\triangledown(:)\triangledown(:)$. Therefore, the other hylomorphism needs to be extended to handle the new constructors. This is obtained by duplicating the cases of the coalgebra together with the algebra components (ϕ_i). Not all the cases may be duplicated, but only those where the copied constructor occurs in a recursive position of a pattern. For the sake of clarity, in our example we have duplicated cases for every occurrence of (:) in a pattern, and later removed the occurrences of ($\triangleright$) in non-recursive positions.

Chapter 6

Mutually recursive functions

So far, we have been dealing with recursive definitions that call themselves to define the recursion. We will show in this chapter a broader class of recursive functions, the theory and the algorithms to fuse them. A substantial part of the theoretic treatment of those functions is based on previous work by Iwasaki et al. [IHT98].

Consider the following definitions:

$$
\begin{aligned}
&\textbf{data } \mathit{Rose}\ a = \mathit{Branch}\ a\ (\mathit{Forest}\ a)\\
&\textbf{data } \mathit{Forest}\ a = \mathit{NilF} \mid \mathit{ConsF}\ (\mathit{Rose}\ a)\ (\mathit{Forest}\ a)\\
&\mathit{sumRose} :: \mathit{Rose}\ \mathit{Int} \to \mathit{Int}\\
&\mathit{sumRose}\ (\mathit{Branch}\ a\ \mathit{fr}) \ = a + \mathit{sumForest}\ \mathit{fr}\\
&\mathit{sumForest} :: \mathit{Forest}\ \mathit{Int} \to \mathit{Int}\\
&\mathit{sumForest}\ \mathit{NilF} \qquad\quad = 0\\
&\mathit{sumForest}\ (\mathit{ConsF}\ r\ \mathit{fr}) = \mathit{sumRose}\ r + \mathit{sumForest}\ \mathit{fr}\\
&\mathit{mapRose} :: (a \to b) \to \mathit{Rose}\ a \to \mathit{Rose}\ b\\
&\mathit{mapRose}\ f\ (\mathit{Branch}\ a\ \mathit{fr}) \ = \mathit{Branch}\ (f\ a)\ (\mathit{mapForest}\ f\ \mathit{fr})\\
&\mathit{mapForest} :: (a \to b) \to \mathit{Forest}\ a \to \mathit{Forest}\ b\\
&\mathit{mapForest}\ f\ \mathit{NilF} \qquad\quad = \mathit{NilF}\\
&\mathit{mapForest}\ f\ (\mathit{ConsF}\ r\ \mathit{fr}) = \mathit{ConsF}\ (\mathit{mapRose}\ f\ r)\ (\mathit{mapForest}\ f\ \mathit{fr})
\end{aligned}
$$

Normal hylomorphisms cannot express in their functors the recursive calls to these mutually recursive definitions. For instance, function *sumRose* does not call itself, but calls function *sumForest* which calls *sumRose*. This kind of somewhat more indirect recursion motivates a generalization.

We will work also with functors which take pairs and may return pairs.

- Projection functors:

$$
\begin{aligned}
&\Pi_1\ (X, Y) = X\\
&\Pi_1\ (f, g) = f\\
&\Pi_2\ (X, Y) = Y\\
&\Pi_2\ (f, g) = g
\end{aligned}
$$

- Our original elemental functors $\times$, $+$ and $\bar{}$ overloaded to work over pairs of types and functions. For instance, $(A, B) \times (C, D) = (A \times C, B \times D)$
- An split operator overloaded for functors taking pairs:

$$\langle F, G \rangle\ (X, Y) = (F\ (X, Y), G\ (X, Y))$$
$$\langle F, G \rangle\ (f, g) = (F\ (f, g), G\ (f, g))$$

It follows the generalization of hylomorphisms to express mutual recursion.

Definition 6.1 (Mutual hylomorphism) Let F be a functor from pairs to pairs of types and functions. Let $\phi : F\ (C, D) \to (C, D)$ be an F-algebra and $\psi : (A, B) \to F\ (A, B)$ an F-coalgebra. A mutual hylomorphism $[\![\phi, \psi]\!]_F$ is a pair of functions $(f : A \to C, g : B \to D)$ which is the least fixpoint of

$$(f, g) = \phi \circ F\ (f, g) \circ \psi$$

Unlike hylomorphisms, mutual hylomorphisms do not use to appear directly applied to input values in functional programs. Instead, it is far more common to see their components f and g independently applied.

Now we can write *sumRose* as a mutual hylo:

$$\begin{array}{l}(\mathit{sumRose}, \mathit{sumForest}) = [\![(\phi_1, \phi_2), \mathit{out}_F]\!]_F \\ \quad \textbf{where}\ F = \langle F_1, F_2 \rangle \\ \qquad F_1 = \overline{\mathit{Int}} \times \Pi_2 \qquad\qquad F_2 = \overline{\mathbf{1}} + \Pi_1 \times \Pi_2 \\ \qquad \phi_1 :: F_1\ (\mathit{Int}, \mathit{Int}) \to \mathit{Int} \\ \qquad \phi_1\ (a, \mathit{fr}) = a + \mathit{fr} \\ \qquad \phi_2 :: F_2\ (\mathit{Int}, \mathit{Int}) \to \mathit{Int} \\ \qquad \phi_2\ (1, ()) \quad\ = 0 \\ \qquad \phi_2\ (2, (r, \mathit{fr})) = r + \mathit{fr}\end{array}$$

Note that the hylo functor is expressed as a split of functors. Each component describes the structure for each of the recursive definitions: *sumRose* and *sumForest*. The coalgebra out_F corresponds to a pair of coalgebras, each of them belonging to the respective recursive definition.

The Acid Rain laws for mutual hylos are expressed in the same way as for normal hylos, now using functors from pairs to pairs.

Theorem 6.2 (Acid rain for mutual hylos)

fold-unfold:
$$[\![\phi, \mathit{out}_F]\!]_F \circ [\![\mathit{in}_F, \psi]\!]_F = [\![\phi, \psi]\!]_F$$

fold-hylo:
$$\frac{(\!|\phi|\!)_F \text{ is strict} \qquad \tau :: (F\ (a, b) \to (a, b)) \to (G\ (a, b) \to (a, b))}{(\!|\phi|\!)_F \circ [\![\tau(\mathit{in}_F), \psi]\!]_G = [\![\tau(\phi), \psi]\!]_G}$$

hylo-unfold:
$$\frac{\sigma :: ((a, b) \to F\ (a, b)) \to ((a, b) \to G\ (a, b))}{[\![\phi, \sigma(\mathit{out}_F)]\!]_G \circ [\!(\psi)\!]_F = [\![\phi, \sigma(\psi)]\!]_G}$$

The proof is a generalization to the mutual hylo context of the proof of the normal acid rain laws [TM95, Wad89].

To illustrate application of acid rain in this context, suppose we want to fuse $sumMapRoseForest = (sumRose, sumForest) \circ mapRoseForest\ f$. If we write *mapRoseForest* as a mutual hylomorphism, we obtain:

$$
\begin{aligned}
&mapRoseForest\ f = [\![(\psi_1, \psi_2)]\!]_F \\
&\quad \textbf{where}\ \psi_1 :: Rose\ a \to F_1\ (Rose\ a, Forest\ a) \\
&\qquad\quad \psi_1\ (Branch\ a\ (Forest\ fr)) = (f\ a, fr) \\
&\qquad\quad \psi_2 :: Forest\ a \to F_2\ (Rose\ a, Forest\ a) \\
&\qquad\quad \psi_2\ NilF\ \ \ \ \ \ \ \ \ \ = (1, ()) \\
&\qquad\quad \psi_2\ (ConsF\ r\ fr) = (2, (r, fr))
\end{aligned}
$$

Now, we can apply the fold-unfold law to get $[\![(\phi_1, \phi_2), (\psi_1, \psi_2)]\!]_F$. Inlining we get:

$$
\begin{aligned}
&sumMapRose :: (a \to Int) \to Rose\ a \to Int \\
&sumMapRose\ f\ (Branch\ a\ fr)\ \ = f\ a + sumMapForest\ f\ fr \\
&sumMapForest :: (a \to Int) \to Forest\ Int \to Int \\
&sumMapForest\ f\ NilF\ \ \ \ \ \ \ \ \ \ = 0 \\
&sumMapForest\ f\ (ConsF\ r\ fr) = sumMapRose\ f\ r + sumMapForest\ f\ fr
\end{aligned}
$$

6.1 Derivation of mutual hylomorphisms

Let f and g be mutualy recursive functions with definitions in the following form.

$$
\begin{aligned}
f &= \lambda v_1\ \ldots\ v_m \to b_f \\
g &= \lambda v_1\ \ldots\ v_m \to b_g
\end{aligned}
$$

We will assume that both f and g make recursion over a single argument, and that they have the same amount of constant arguments. Without loss of generality, it can be assumed that both f and g are recursive over the last argument and that their constant arguments appear in the same order.

The mutual hylomorphism is derived as follows:

$$
\begin{aligned}
&(sumRose, sumForest) = \lambda v_1\ \ldots\ v_m \to [\![(\phi_f, \phi_g), (\psi_f, \psi_g)]\!]_{\langle F_f, F_g \rangle} \\
&\quad \textbf{where}\ (\phi_f, \psi_f, F_f) = \mathcal{H}(f, g, \lambda v_1\ \ldots\ v_m \to b_f) \\
&\qquad\quad (\phi_g, \psi_g, F_g) = \mathcal{H}(f, g, \lambda v_1\ \ldots\ v_m \to b_g)
\end{aligned}
$$

We present in Figure 6.1 the auxiliary algorithm $\mathcal{H}$ which does the hard work.

$$
\begin{array}{l}
\mathcal{H}(f, g, \lambda v_1\ \ldots\ v_m \to \textbf{case}\ t_0\ \textbf{of}\ p_1 \to t_1; \ldots; p_n \to t_n) = (\phi_1 \triangledown \cdots \triangledown \phi_n, \psi, F) \\
\quad \textbf{where} \\
\quad\quad \psi\ = \lambda v_m \to \textbf{case}\ t_0\ \textbf{of} \\
\quad\quad\quad\quad\quad\quad p_1 \to (1, (v_{11}, \ldots, v_{1l_1}, t'_{11}, \ldots, t'_{1s_1})) \\
\quad\quad\quad\quad\quad\quad\quad\quad \vdots \\
\quad\quad\quad\quad\quad\quad p_n \to (n, (v_{n1}, \ldots, v_{nl_n}, t'_{n1}, \ldots, t'_{ns_n})) \\
\quad\quad \phi_i\ = \lambda(v_{i1}, \ldots, v_{il_i}, u_1, \ldots, u_{s_i}) \to t''_i \\
\quad\quad F\ = F_1 + \cdots + F_n \\
\quad\quad F_i = \overline{\Gamma(\mathtt{v_{i1}})} \times \cdots \times \overline{\Gamma(\mathtt{v_{i1_i}})} \times P_1 \times \cdots \times P_{s_i} \quad \text{-- } \Gamma(v) \text{ returns the type of } v \\
\quad\quad (\{v_{i1}, \ldots, v_{il_i}\}, \{(P_1, u_1, t_{i1}), \ldots, (P_{s_i}, u_{s_i}, t_{is_i})\}, t''_i) = \mathcal{D}(p_i, t_i) \\
\quad\quad \mathcal{D}(p_i, v) = (\{v\}, \emptyset, v) \text{ if } v \in vars(p_i) \cup \{v_m\} \\
\quad\quad\quad\quad\quad\quad (\emptyset, \emptyset, v) \quad \text{otherwise} \\
\quad\quad \mathcal{D}(p_i, (t_1, \ldots, t_n)) = (c_1 \cup \cdots \cup c_n, c'_1 \cup \cdots \cup c'_n, (t'_1, \ldots, t'_n)) \\
\quad\quad\quad \textbf{where}\ (c_i, c'_i, t'_i) = \mathcal{D}(p_i, t_i) \\
\quad\quad \mathcal{D}(p_i, C_j\ (t_1, \ldots, t_n)) = (c_1 \cup \cdots \cup c_n, c'_1 \cup \cdots \cup c'_n, C_j\ (t'_1, \ldots, t_n)) \\
\quad\quad\quad \textbf{where}\ (c_i, c'_i, t'_i) = \mathcal{D}(p_i, t_i) \\
\quad\quad \mathcal{D}(p_i, h\ t_1\ \ldots\ t_m)) = (\emptyset, \{(\Pi_1, u, t_m)\}, u) \text{ if } h = f \text{ and } v_i = t_i \text{ for all } i < m \\
\quad\quad\quad\quad\quad\quad\quad\quad (\emptyset, \{(\Pi_2, u, t_m)\}, u) \text{ if } h = g \text{ and } v_i = t_i \text{ for all } i < m \\
\quad\quad\quad\quad\quad\quad\quad\quad (c_1 \cup \cdots \cup c_n, c'_1 \cup \cdots \cup c'_n, h\ t'_1\ \ldots\ t'_n) \text{ otherwise} \\
\quad\quad\quad \textbf{where}\ (c_i, c'_i, t'_i) = \mathcal{D}(p_i, t_i) \\
\quad\quad\quad\quad\ u \text{ is a fresh variable} \\
\quad\quad \mathcal{D}(p_i, \textbf{let}\ v = t_0\ \textbf{in}\ t_1) = (c_0 \cup c_1, c'_0 \cup c'_1, \textbf{let}\ v = t'_0\ \textbf{in}\ t'_1) \\
\quad\quad\quad \textbf{where}\ (c_i, c'_i, t'_i) = \mathcal{D}(p_i, t_i) \\
\quad\quad \mathcal{D}(p_i, \lambda v \to t) = (c, c', \lambda v \to t') \\
\quad\quad\quad \textbf{where}\ (c, c', t') = \mathcal{D}(p_i, t) \\
\quad\quad \mathcal{D}(p_i, \textbf{case}\ t_0\ \textbf{in}\ p_1 \to t_1; \ldots; p_n \to t_n) = \\
\quad\quad\quad\quad (c_1 \cup \cdots \cup c_n, c'_1 \cup \cdots \cup c_n, \textbf{case}\ t_0\ \textbf{of}\ p_1 \to t'_1; \ldots; p_n \to t'_n) \\
\quad\quad\quad \textbf{where}\ (c_i, c'_i, t'_i) = \mathcal{D}(p_i, t_i)
\end{array}
$$

Figure 6.1: Mutual hylomorphism derivation algorithm

The algorithm is similar to that for hylomorphism derivation. The major changes are in the auxiliary algorithm $\mathcal{D}$. Algorithm $\mathcal{D}$ tells where the recursive calls are, and which are their arguments. As we are working now with mutual hylomorphisms we must tell for each recursive call whether f or g was called. That is said through the projections Π_1 and Π_2, respectively. The same provisions regarding variable capture we made for the derivation algorithm for normal hylomorphisms apply now for mutual hylos.

The result of applying the algorithm over *sumRose* is:

$$
\begin{array}{l}
(sumRose, sumForest) = [\![(\phi_1, \phi_2), (\psi_1, \psi_2)]\!]_F \\
\quad \textbf{where}\ F = \langle \overline{Int} \times \Pi_2, \overline{\mathbf{1}} + \Pi_1 \times \Pi_2 \rangle \\
\quad\quad\quad \phi_1\ (a, u_0) = a + u_0 \\
\quad\quad\quad \phi_2\ (1, ()) = 0 \\
\quad\quad\quad \phi_2\ (2, (u_1, u_2)) = u_1 + u_2 \\
\quad\quad\quad \psi_1\ v = \textbf{case}\ v\ \textbf{of}\ \mathit{Branch}\ a\ f \to (a, f) \\
\quad\quad\quad \psi_2\ v = \textbf{case}\ v\ \textbf{of} \\
\quad\quad\quad\quad\quad\quad\quad \mathit{NilF} \to (1, ()) \\
\quad\quad\quad\quad\quad\quad\quad \mathit{ConsF}\ r\ fr \to (2, (r, fr)
\end{array}
$$

$$
\begin{array}{l}
\mathcal{T}(h, F, \phi :: G_h\ a_h \to a_h) : (F\ (a_1, a_2) \to (a_1, a_2)) \to G_h\ a_h \to a_h \\
\mathcal{T}(h, \langle F_1, F_2\rangle, (\phi_1 \triangledown \cdots \triangledown \phi_n) :: G_h\ a_h \to a_h) = \\
\qquad \lambda(\alpha_1 \triangledown \cdots \triangledown \alpha_{m_1}, \beta_1 \triangledown \cdots \triangledown \beta_{m_2}) \to \mathcal{T}'(\phi_1) \triangledown \cdots \triangledown \mathcal{T}'(\phi_n) \\
\quad \textbf{where}\ F_{i1} + \cdots + F_{im_i} = F_i \\
\qquad \mathcal{T}'(\lambda bvs \to t) = \lambda bvs \to \mathcal{A}_h\ t \\
\qquad \mathcal{A}_h(v) \qquad\quad = v \text{ (if } v \text{ is a recursive variable according to } G) \\
\qquad \mathcal{A}_1(C_j\ t_1\ \ldots\ t_k) = \alpha_j\ (F_{1j}\ (\mathcal{A}_1, \mathcal{A}_2)\ (t_1, \ldots, t_k)) \\
\qquad \mathcal{A}_2(C_j\ t_1\ \ldots\ t_k) = \beta_j\ (F_{2j}\ (\mathcal{A}_1, \mathcal{A}_2)\ (t_1, \ldots, t_k)) \\
\qquad \mathcal{A}_h(t) \qquad\quad = \Pi_h\ (\!|(\alpha_1 \triangledown \cdots \triangledown \alpha_{m_1}, \beta_1 \triangledown \cdots \triangledown \beta_{m_2})|\!)_{\langle F_1, F_2\rangle}\ t \text{ (all other cases)}
\end{array}
$$

Figure 6.2: Algorithm for deriving τ

6.2 Derivation of τ

Having a mutual hylomorphism $[\![(\phi_1, \phi_2), \psi]\!]_{\langle G_1, G_2\rangle}$ we might want to derive an equivalent one of the form $[\![\tau(in_F), \psi]\!]_{\langle G_1, G_2\rangle}$. Indeed, we will be deriving a hylo of the form $[\![(\lambda\alpha \to (\tau_1\ \alpha, \tau_2\ \alpha))\ in_F, \psi]\!]_{\langle G_1, G_2\rangle}$ that we will often write as $[\![(\tau_1(in_F), \tau_2(in_F)), \psi]\!]_{\langle G_1, G_2\rangle}$.

This is how we obtain τ:

$$
\begin{array}{l}
\tau :: (F\ (a, b) \to (a, b)) \to \langle G_1, G_2\rangle\ (a, b) \to (a, b) \\
\tau(\alpha) = (\mathcal{T}(1, F, \phi_1 :: G_1\ (a, b) \to a)(\alpha), \mathcal{T}(2, F, \phi_2 :: G_2\ (a, b) \to b)(\alpha))
\end{array}
$$

Algorithm $\mathcal{T}$ is presented in Figure 6.2. The functor F must be known a priori. It will come most likely from the mutual fold in a composition of the form $(\!|\phi'|\!)_F \circ [\![(\phi_1, \phi_2), \psi]\!]_{\langle G_1, G_2\rangle}$.

As with the mutual hylomorphism derivation algorithm, we made small tweaks to the derivation algorithm of τ for the normal case. The auxiliary algorithm $\mathcal{A}$ must keep track of which type contains the constructor whose arguments are traversed. In a mutual hylomorphism, the algebra of either f and g may contain nested applications of constructors of mixed types.

Example 6.3 (Deriving τ) Consider the following definitions

$$
\begin{array}{l}
\mathit{addChildRose} :: a \to \mathit{Rose}\ a \to \mathit{Rose}\ a \\
\mathit{addChildRose}\ a\ (\mathit{Branch}\ b\ \mathit{fr}) = \\
\qquad\qquad \mathit{Branch}\ b\ (\mathit{ConsF}\ (\mathit{Branch}\ a\ \mathit{NilF}) \\
\qquad\qquad\qquad\qquad (\mathit{addChildForest}\ a\ \mathit{fr})) \\
\mathit{addChildForest} :: a \to \mathit{Rose}\ a \to \mathit{Rose}\ a \\
\mathit{addChildForest}\ a\ \mathit{NilF} = \mathit{NilF} \\
\mathit{addChildForest}\ a\ (\mathit{ConsF}\ r\ \mathit{fr}) = \\
\qquad\qquad \mathit{ConsF}\ (\mathit{addChildRose}\ a\ r)\ (\mathit{addChildForest}\ a\ \mathit{fr})
\end{array}
$$

We express it as a mutual hylomorphism:

$$
\begin{array}{l}
\mathit{addChild}\ a = [\![(\phi_1, \phi_2), \mathit{out}_F]\!]_F \\
\quad \textbf{where}\ F = \langle F_1, F_2\rangle
\end{array}
$$

$$
\begin{aligned}
&F_1 = \overline{a} \times \Pi_2 \\
&F_2 = \overline{\mathbf{1}} + \Pi_1 \times \Pi_2 \\
&\phi_1\ (b, u_0) &&= \mathit{Branch}\ b\ (\mathit{ConsF}\ (\mathit{Branch}\ a\ \mathit{NilF})\ u_0) \\
&\phi_2\ (1, ()) &&= \mathit{NilF} \\
&\phi_2\ (2, (u_1, u_2)) &&= \mathit{ConsF}\ u_1\ u_2
\end{aligned}
$$

If we apply the derivation algorithm for τ to *addChild* we get

$$
\begin{aligned}
&\mathit{addChild}\ a = [\![(\tau_1(\mathit{in}_F), \tau_2(\mathit{in}_F)), \mathit{out}_F]\!]_F \\
&\quad \textbf{where}\ \tau_1 :: (F\ (a, b) \to (a, b)) \to F_1\ (a, b) \to a \\
&\qquad\qquad \tau_1(\alpha_1, \beta_1 \triangledown \beta_2)\ (b, u_0) &&= \alpha_1\ b\ (\beta_2\ (\alpha_1\ a\ \beta_1)\ u_0) \\
&\qquad\qquad \tau_2 :: (F\ (a, b) \to (a, b)) \to F_2\ (a, b) \to b \\
&\qquad\qquad \tau_2(\alpha_1, \beta_1 \triangledown \beta_2)\ (1, ()) &&= \beta_1\ () \\
&\qquad\qquad \tau_2(\alpha_1, \beta_1 \triangledown \beta_2)\ (2, (u_1, u_2)) &&= \beta_2\ u_1\ u_2
\end{aligned}
$$

If we wanted to fuse $\mathit{sumRose} \circ \mathit{addChild}\ a$, we would get the mutual hylo $[\![(\tau_1(\phi_1), \tau_2(\phi_2)), \mathit{out}_F]\!]_F$, being ϕ_1 and ϕ_2 the respective algebra components of $(\mathit{sumRose}, \mathit{sumForest})$. The inlined result is

$$
\begin{aligned}
&\mathit{sumAddChildRose}\ a\ (\mathit{Branch}\ b\ \mathit{fr}) = \\
&\qquad\qquad\qquad b + (a + 0) + \mathit{sumAddChildForest}\ a\ \mathit{fr} \\
&\mathit{sumAddChildForest}\ a\ \mathit{NilF} = 0 \\
&\mathit{sumAddChildForest}\ \ a\ (\mathit{ConsF}\ r\ \mathit{fr}) = \\
&\qquad\qquad\qquad \mathit{sumAddChildRose}\ a\ r + \mathit{sumAddChildForest}\ a\ \mathit{fr}
\end{aligned}
$$

□

6.3 Derivation of σ

Having a mutual hylomorphism $[\![\phi, (\psi_1, \psi_2)]\!]_{\langle G_1, G_2 \rangle}$ we can derive an equivalent one of the form $[\![\phi, \sigma(\mathit{out}_F)]\!]_{\langle G_1, G_2 \rangle}$. The transformer σ is calculated as follows

$$
\begin{aligned}
&\sigma : ((a, b) \to F\ (a, b)) \to (a, b) \to \langle G_1, G_2 \rangle\ (a, b) \\
&\sigma(\beta) = (\mathcal{S}(1, F, \psi_1 :: a \to G_1\ (a, b))(\beta), \mathcal{S}(2, F, \psi_2 :: b \to G_2\ (a, b))(\beta))
\end{aligned}
$$

where algorithm $\mathcal{S}$ is presented in Figure 6.3. Again, the functor F must be known a priori, and it is expected to come from the mutual unfold in a composition of the form $[\![\phi, (\psi_1, \psi_2)]\!]_{\langle G_1, G_2 \rangle} \circ [\![\phi']\!]_F$.

Algorithm $\mathcal{S}$ is the dual of the τ derivation algorithm. Therefore, the kind of generalization we did with respect to the σ derivation algorithm for the non-mutual case is very similar. Essentially, $\mathcal{B}_h$ keeps track through its subindex of which are the types of the constructors being abstracted in the pattern.

$$
\begin{array}{l}
\mathcal{S}(h, \langle F_1, F_2\rangle, \lambda v \to \textbf{case } v \textbf{ of } p_1 \to t_1; \ldots; p_n \to t_n) = \\
\quad \lambda(\beta_1, \beta_2) \to \lambda v \to \textbf{case } v \textbf{ of } \mathcal{B}_h\ (p_1) \to t_1; \ldots; \mathcal{B}_h(p_n) \to t_n \\
\quad \textbf{where } F_{i1} + \cdots + F_{im_i} = F_i \\
\qquad\quad \begin{array}{ll}
\mathcal{B}_h(v) & = v \text{ if } v \text{ is a recursive variable} \\
\mathcal{B}_h(C_j\ p_1\ \ldots\ p_k) & = \beta_h{\cdot}(j, F_{hj}\ (\mathcal{B}_1, \mathcal{B}_2)\ (p_1, \ldots, p_k)) \\
\mathcal{B}_h(p) & = \Pi_h\ [\![\beta]\!]_{\langle F_1, F_2\rangle}{\cdot}p \text{ all other cases}
\end{array}
\end{array}
$$

Figure 6.3: Algorithm for deriving σ

Example 6.4 (Derivation of σ) Consider the following definitions:

$$
\begin{array}{l}
\mathit{removeLeavesRose} :: \mathit{Rose}\ a \to \mathit{Rose}\ a \\
\mathit{removeLeavesRose}\ (\mathit{Branch}\ a\ \mathit{fr}) = \mathit{Branch}\ a\ (\mathit{removeLeavesForest}\ \mathit{fr}) \\
\mathit{removeLeavesForest} :: \mathit{Forest}\ a \to \mathit{Forest}\ a \\
\begin{array}{ll}
\mathit{removeLeavesForest}\ \mathit{NilF} & = \mathit{NilF} \\
\mathit{removeLeavesForest}\ (\mathit{ConsF}\ (\mathit{Branch}\ \_\ \mathit{NilF})\ \mathit{fr}) = & \\
\qquad \mathit{removeLeavesForest}\ \mathit{fr} & \\
\mathit{removeLeavesForest}\ (\mathit{ConsF}\ r\ \mathit{fr}) & = \\
\qquad \mathit{ConsF}\ (\mathit{removeLeavesRose}\ r) & \\
\qquad\quad (\mathit{removeLeavesForest}\ \mathit{fr}) &
\end{array}
\end{array}
$$

As a mutual hylomorphism they look like:

$$
\begin{array}{l}
\mathit{removeLeaves} = [\![\phi, (\psi_1, \psi_2)]\!]_{\langle G_1, G_2\rangle} \\
\quad \textbf{where } G_1 = \overline{a} \times \Pi_2 \\
\qquad\quad G_2 = \overline{\mathbf{1}} + \Pi_2 + \Pi_1 \times \Pi_2 \\
\qquad\quad \phi\ \ = (\mathit{Branch}, \mathit{NilF} \triangledown \mathit{id} \triangledown \mathit{ConsF}) \\
\qquad\quad \psi_1\ (\mathit{Branch}\ a\ \mathit{fr}) = (a, \mathit{fr}) \\
\qquad\quad \begin{array}{ll}
\psi_2\ \mathit{NilF} & = (1, ()) \\
\psi_2\ (\mathit{ConsF}\ (\mathit{Branch}\ \_\ \mathit{NilF})\ \mathit{fr}) & = (2, \mathit{fr}) \\
\psi_2\ (\mathit{ConsF}\ r\ \mathit{fr}) & = (3, (r, \mathit{fr}))
\end{array}
\end{array}
$$

If we derive σ using the algorithm above we get

$$
\begin{array}{l}
\mathit{removeLeaves} = [\![\phi, (\sigma_1(\mathit{out}_F), \sigma_2(\mathit{out}_F))]\!]_{\langle G_1, G_2\rangle} \\
\quad \textbf{where } F = \langle \overline{a} \times \Pi_2, \overline{\mathbf{1}} + \Pi_1 \times \Pi_2\rangle \\
\qquad\quad \sigma_1 :: ((a, b) \to F\ (a, b)) \to (a, b) \to G_1\ (a, b) \\
\qquad\quad \sigma_1(\beta_1, \beta_2)\ v = \textbf{case } v \textbf{ of } \beta_1{\cdot}(1, (a, \mathit{fr})) \to (1, (a, \mathit{fr})) \\
\qquad\quad \sigma_2 :: ((a, b) \to F\ (a, b)) \to (a, b) \to G_2\ (a, b) \\
\qquad\quad \sigma_2(\beta_1, \beta_2)\ v = \textbf{case } v \textbf{ of} \\
\qquad\qquad \begin{array}{ll}
\beta_2{\cdot}(1, ()) & \to (1, ()) \\
\beta_2{\cdot}(2, (\beta_1{\cdot}(1, (\_, \beta_2{\cdot}(1, ()))), \mathit{fr}))) & \to (2, \mathit{fr}) \\
\beta_2{\cdot}(2, (r, \mathit{fr})) & \to (3, (r, \mathit{fr}))
\end{array}
\end{array}
$$

□

6.4 Regular Functors

We will show in this section that mutual hylomorphims can be used to express functions for which it is not possible to derive a hylomorphism with the standard algorithms. In fact, the standard algorithms are able to derive only hylomorphisms with a polynomial functor. Some functions, however, employ a functor of a broader class.

We need the preliminary definition of type functor. A common example of type functor is *List*, which maps a type a into the type $[a]$ of lists of elements of type a.

Definition 6.5 Given a bifunctor F, we define the functor $\tau(F)$, called a **type functor**, as follows:

$$\begin{aligned} \tau(F)(x) &= \mu(F_x) \text{ \{- } F_x \text{ is the partial application of } F \text{ -\}} \\ \tau(F)(f) &= (\!|in_F \circ F\ (f, id)|\!)_F \end{aligned}$$

for every function $f : a \to b$.

Now we can define $List = \tau(\overline{\mathbf{1}} + \Pi_1 \times \Pi_2)$, whose action on functions is just expressed by the function *map*:

$$map\ f = (\!|in_{\overline{\mathbf{1}}+\overline{b}\times I} \circ (id + f \times id)|\!)_{\overline{\mathbf{1}}+\overline{a}\times I}$$

Consider the following definitions:

$$\begin{aligned} &\mathbf{data}\ Rose\ a = Fork\ a\ (List\ (Rose\ a)) \\ &mapRose :: (a \to b) \to Rose\ a \to Rose\ b \\ &mapRose\ g\ (Fork\ a\ ls) = Fork\ (g\ a)\ (map\ (mapRose\ g)\ ls) \end{aligned}$$

Now that we have the definition of *List* as a functor we can write this function as a hylomorphism

$$\begin{aligned} &mapRose\ g = [\![in_F \circ (id + g \times id), out_F]\!]_F \\ &\quad \mathbf{where}\ F = \overline{a} \times List \end{aligned}$$

Unfortunately, F is not a polynomial functor since it contains the functor *List* which is not polynomial. This implies that we can not derive this hylomorphism with the standard algorithms. Functors of this kind are called regular functors.

Definition 6.6 A **regular functor** is a functor built from polynomial and type functors.

We have found, however, that the fusion laws for mutual hylomorphisms can be used to fuse hylomorphisms involving regular (yet non-polynomial)

functors. The trick is writing the problematic functions as a set of mutually recursive functions. For example,

$$
\begin{aligned}
&\mathit{mapRose} :: (a \to b) \to \mathit{Rose}\ a \to \mathit{Rose}\ b\\
&\mathit{mapRose}\ g\ (\mathit{Fork}\ a\ ls) = \mathit{Fork}\ (g\ a)\ (\mathit{mapmapRose}\ g\ ls)\\
&\mathit{mapmapRose} :: (a \to b) \to \mathit{List}\ (\mathit{Rose}\ a) \to \mathit{List}\ (\mathit{Rose}\ b)\\
&\mathit{mapmapRose}\ g\ \mathit{Nil} = \mathit{Nil}\\
&\mathit{mapmapRose}\ g\ (\mathit{Cons}\ a\ as) = \mathit{Cons}\ (\mathit{mapRose}\ g\ a)\\
&\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{mapmapRose}\ g\ as)
\end{aligned}
$$

Now, we can derive a mutual hylomorphism:

$$
\begin{aligned}
&\mathit{mapRose}\ g = [\![\mathit{in}_F \circ (\mathit{id} + g \times \mathit{id}, \mathit{id}), \mathit{out}_F]\!]_F\\
&\quad \textbf{where}\ F = \langle \overline{a} \times \Pi_2, \overline{\mathbf{1}} + \Pi_1 \times \Pi_2 \rangle
\end{aligned}
$$

Doing so we have a better chance to apply the fusion laws for mutual hylos when a function like *mapRose* appears in a composition.

We have introduced a new definition *mapmapRose* satisfying the equality $\mathit{mapmapRose}\ g = \mathit{map}\ (\mathit{mapRose}\ g)$. It was obtained by fixing the first argument of *map* to *mapRose* g.

The procedure can be generalized for a function f of the form:

$$
\begin{aligned}
&f\ l_1\ \ldots\ l_m\ r = C_1[h\ (f\ l_1\ \ldots\ l_m)]\\
&h\ g\ r = C_2[g][h\ g]
\end{aligned}
$$

where $l_1\ \ldots\ l_m$ are the constant parameters of f, C_1 and C_2 are term contexts, r is the recursive argument, and h is some arbitrary function to be processed by fixing its first argument to $f\ l_1\ \ldots\ l_m$ and adding f constant parameters as constant parameters of the resulting function. In our previous example *map* played the role of h. Note that f is not applied to the recursive argument r in the right hand side of its definition, though r may be referenced in the contexts C_1 and C_2.

For each function h appearing in a call of the form $h\ (f\ l_1\ \ldots\ l_m)$ a new definition hf is derived.

$$
\begin{aligned}
&f\ l_1\ \ldots\ l_m\ r = C_1[hf\ l_1\ \ldots\ l_m]\\
&hf\ l_1\ \ldots\ l_m\ r = C_2[f\ l_1\ \ldots\ l_m][hf\ l_1\ \ldots\ l_m]
\end{aligned}
$$

The procedure is applied iteratively to generated definitions until no more definitions are produced.

6.5 Varying the amount of components

Acid Rain laws for mutual hylos generalize very naturally to recursive definitions that involve more than two definitions. Some mismatches arise, however, when one wants to fuse two mutual hylos that have a different amount of components.

Consider, for example, the following definitions:

$$
\begin{aligned}
&mapF\ [\,] &&= [\,]\\
&mapF\ (a : as) &&= f\ a : mapG\ as\\
&mapG\ [\,] &&= [\,]\\
&mapG\ (a : as) &&= g\ a : mapH\ as\\
&mapH\ [\,] &&= [\,]\\
&mapH\ (a : as) &&= h\ a : mapF\ as\\
&even\ [\,] &&= [\,]\\
&even\ (a : as) &&= odd\ as\\
&odd\ [\,] &&= [\,]\\
&odd\ (a : as) &&= a : even\ as
\end{aligned}
$$

The functions above can be written as mutual hylomorphisms with three and two components respectively.

$$
\begin{aligned}
&(mapF, mapG, mapH) = [\![in_F, (\psi_1, \psi_2, \psi_3)]\!]_F\\
&\quad \textbf{where } F = \langle \overline{\mathbf{1}} + \overline{b} \times \Pi_2, \overline{\mathbf{1}} + \overline{b} \times \Pi_3, \overline{\mathbf{1}} + \overline{b} \times \Pi_1 \rangle\\
&\qquad\quad \psi_1\ [\,] &&= (1, ())\\
&\qquad\quad \psi_1\ (a : as) &&= (2, (f\ a, as))\\
&\qquad\quad \psi_2\ [\,] &&= (1, ())\\
&\qquad\quad \psi_2\ (a : as) &&= (2, (g\ a, as))\\
&\qquad\quad \psi_3\ [\,] &&= (1, ())\\
&\qquad\quad \psi_3\ (a : as) &&= (2, (h\ a, as))\\
&(even, odd) = [\![(\phi_1, \phi_2), out_G]\!]_G\\
&\quad \textbf{where } G = \langle \overline{\mathbf{1}} + \overline{b} \times \Pi_2, \overline{\mathbf{1}} + \overline{b} \times \Pi_1 \rangle\\
&\qquad\quad \phi_1\ (1, ()) &&= [\,]\\
&\qquad\quad \phi_1\ (2, (a, as)) &&= as\\
&\qquad\quad \phi_2\ (1, ()) &&= [\,]\\
&\qquad\quad \phi_2\ (2, (a, as)) &&= a : as
\end{aligned}
$$

We are prevented from fusing $mapF \circ even$ because they belong to mutual hylos with different amount of components, and no generalization of Acid Rain will help us here.

Nevertheless, the difference in the amount of components can resolved by copying some hylo components as follows:

$$
\begin{aligned}
&F' = \langle \overline{\mathbf{1}} + \overline{b} \times \Pi_2, \overline{\mathbf{1}} + \overline{b} \times \Pi_3, \overline{\mathbf{1}} + \overline{b} \times \Pi_4\\
&\qquad , \overline{\mathbf{1}} + \overline{b} \times \Pi_5, \overline{\mathbf{1}} + \overline{b} \times \Pi_6, \overline{\mathbf{1}} + \overline{b} \times \Pi_1 \rangle\\
&(mapF, mapG, mapH, mapF, mapG, mapH) =\\
&\qquad\qquad\qquad\qquad [\![in_{F'}, (\psi_1, \psi_2, \psi_3, \psi_1, \psi_2, \psi_3)]\!]_{F'}\\
&(even, odd, even, odd, even, odd) = [\![(\phi_1, \phi_2, \phi_1, \phi_2, \phi_1, \phi_2), out_{F'}]\!]_{F'}
\end{aligned}
$$

```
pairComponents i j ⟨F_1, ,... F_m⟩ ⟨G_1, ,... G_n⟩ = unzip3 ((i, j, F') : fs)
  where (F', fs, _) = match F_i G_j [(i, j, 1)]
        match Π_i Π_j p =
          let (F', fs, p') = match F_i G_j (p ++ [(i, j, 1 + length p)])
          in  if ∃k.(i, j, k) ∈ p then (Π_k, { }, p)
                else (Π_{1+length p}, fs ∪ {(i, j, F')}, p')
        match Ā Ā p = (Ā, { }, p)
        match (F_i1 ⊕ F_i2) (G_j1 ⊕ G_j2) P | ⊕ ∈ {+, ×} =
          let (F', fs, p') = match F_i1 G_j1 p
              (F'', fs', p'') = match F_i2 G_j2 p'
          in  (F' ⊕ F'', fs ++ fs', p'')
```

Figure 6.4: Matching algorithm for mutual hylos

Now, we could proceed to apply an Acid Rain law for two mutual hylos of six components.

The above transformation changing the amount of components can be performed based on the functors of those hylomorphisms. The algorithm *pairComponents* presented in Figure 6.4 can be used to obtain the following lists:

$$\begin{aligned} \mathit{pairComponents}\ 1\ 1\ F\ G = ([&1,2,3,1,2,3] \\ ,&[1,2,1,2,1,2] \\ ,&[\overline{\mathbf{1}} + \overline{b} \times \Pi_2, \overline{\mathbf{1}} + \overline{b} \times \Pi_3 \\ &,\overline{\mathbf{1}} + \overline{b} \times \Pi_4, \overline{\mathbf{1}} + \overline{b} \times \Pi_5 \\ &,\overline{\mathbf{1}} + \overline{b} \times \Pi_6, \overline{\mathbf{1}} + \overline{b} \times \Pi_1]) \end{aligned}$$

These lists describe which components of the original hylos must be copied to produce the resulting hylos with six components. The first list contains the indexes identifying the *mapF* coalgebra components, and the numbers in the second list contains the indexes identifiying the *even* algebra components. The functors in the third list are the six components of the common functor.

In the algorithm *pairComponents*, the function *unzip3* converts lists of triplets into a triplet of lists. Note that the algorithm expects functors to match in non-recursive positions, and in the amount of sums and products. If they don't, then other techniques will be needed to complement this one.

Whe have shown here a technique that enables application of Acid Rain for the fold-unfold case. However, very similar strategies can be used for the other cases. For instance, a composition involving hylos like the following

$$\begin{aligned} &[\![(\phi_1, \phi_2, \phi_3), (in_{F_1}, in_{F_2}, in_{F_3})]\!]_{\langle F_1, F_2, F_3 \rangle} \\ &[\![(\tau_1(in_{F'_1}, in_{F'_2}), \tau_2(in_{F'_1}, in_{F'_2})), (\psi_1, \psi_2)]\!]_{\langle G_1, G_2 \rangle} \end{aligned}$$

could be smoothed calling *pairComponents* over functors $\langle F_1, F_2, F_3 \rangle$ and $\langle F'_1, F'_2 \rangle$, and copying the hylo components accordingly.

As normal hylos can be considered mutual hylos with only one component, this technique can be also used to fuse normal hylos with mutual hylos with an arbitrary amount of components.

Chapter 7

Recursion over multiple arguments

Consider now that a composition involves a function which is recursive over more than one argument. Such is the case of *zip*, *zipWith* or equality for any recursive data type. If a function like those takes as input the result of another recursive function, then we cannot eliminate the intermediate data structure with the current laws. It is not a problem of their representation as hylomorphisms. For example, consider the function *zip*:

$$
\begin{aligned}
&zip :: [\,a\,] \to [\,b\,] \to [\,a \times b\,] \\
&zip\ (x : xs)\ (y : ys) = (x, y) : zip\ xs\ ys \\
&zip\ \_\quad\ \_\quad\ = [\,]
\end{aligned}
$$

We can derive the following hylomorphism for the uncurried version of it:

$$
\begin{aligned}
&zip :: [\,a\,] \times [\,b\,] \to [\,a \times b\,] \\
&zip = [\![in_F, \psi]\!]_F \\
&\quad \textbf{where}\ F = \overline{\mathbf{1}} + \overline{a \times b} \times I \\
&\qquad\quad \psi :: [\,a\,] \times [\,b\,] \to F\ ([\,a\,] \times [\,b\,]) \\
&\qquad\quad \psi\ (x : xs, y : ys) = (2, ((x, y), (xs, ys))) \\
&\qquad\quad \psi\ (\_\quad , \_\quad\ ) = (1, ())
\end{aligned}
$$

The problem is that the acid rain laws expect the hylomorphism to take as input the recursive structure to eliminate, whereas in a composition like

$$zip \circ (map\ f \times id)$$

the intermediate structure comes as a component of the input pair.

This problem has been studied in [HIT97], where a special operator is proposed to express coalgebras that take pairs of values. Then, an extension of Acid Rain is stated to cope with the new operator. We won't adopt that approach here, since we find unnecessary to introduce such operator. Instead, we suggest the following law:

Lemma 7.1 *Acid rain for recursion over two arguments*

$$\frac{\sigma :: \forall a.(a \to F\ a) \to a \times b \to G\ (a \times b)}{[\![\phi, \sigma(out_F)]\!]_G \circ ([\![\psi]\!]_F \times id) = [\![\phi, \sigma(\psi)]\!]_G}$$

As you can see, the coalgebra transformer σ is not much different from the one in the usual hylo-unfold law. The only small and crucial tweak is that the out_F coalgebra is abstracted for the type of the first component of the pair taken by $\sigma(out_F)$.

It turns out that the law can be generalized to:

Lemma 7.2

$$\frac{\sigma :: \forall a.\ (a \to F\ a) \to H\ a \to G\ (H\ a)}{[\![\phi, \sigma(out_F)]\!]_G \circ H\ [\![\psi]\!]_F = [\![\phi, \sigma(\psi)]\!]_G}$$

This means that we can apply the law whatever the amount of recursive arguments we have, and whichever is the argument where the composition appears.

In the proof of Lemma 7.2 we make use of the following property known as **hylo fusion** [FM91, Mal90].

$$\psi \circ f = F\ f \circ \psi' \quad \Rightarrow \quad [\![\phi, \psi]\!]_F \circ f = [\![\phi, \psi']\!]_F$$

We use, also, the characterization of unfolds as the unique homomorphism from a given coalgebra ψ to coalgebra out_F [GJ98]:

$$out_F \circ f = F\ f \circ \psi \quad \Leftrightarrow \quad f = [\![\psi]\!]_F$$

Proof From the type of σ we can obtain the following free theorem [Wad89]:

$$\psi' \circ f = F\ f \circ \psi \Rightarrow \sigma(\psi') \circ H\ f = G\ (H\ f) \circ \sigma(\psi)$$

Now, by taking $\psi' = out_F$ and $f = [\![\psi]\!]_F$ we get:

$$\begin{aligned} out_F \circ [\![\psi]\!]_F = F\ [\![\psi]\!]_F \circ \psi \Rightarrow \\ \sigma(out_F) \circ H\ [\![\psi]\!]_F = G\ (H\ [\![\psi]\!]_F) \circ \sigma(\psi) \end{aligned}$$

The premise of this implication holds by the characterization of unfold, and therefore we have that:

$$\sigma(out_F) \circ H\ [\![\psi]\!]_F = G\ (H\ [\![\psi]\!]_F) \circ \sigma(\psi)$$

Finally, by applying **hylo fusion**, we obtain:

$$[\![\phi, \sigma(out_F)]\!]_G \circ H\ [\![\psi]\!]_F = [\![\phi, \sigma(\psi)]\!]_G$$

as desired. □

Example 7.3 To see this law in action we rewrite *zip* in terms of a coalgebra transformer which only abstracts the patterns corresponding to the first input list:

$$
\begin{array}{l}
zip = [\![\sigma(out_F)]\!]_G \\
\quad \textbf{where } G = \overline{\mathbf{1}} + (\overline{a} \times \overline{b}) \times I \\
\qquad\quad F = \overline{\mathbf{1}} + \overline{a} \times I \\
\qquad\quad \sigma :: \forall a.(a \to F\ a) \to a \times [\,b\,] \to G\ (a \times [\,b\,]) \\
\qquad\quad \sigma(\beta)\ (x, y) = \textbf{case } (\beta\ x, y) \textbf{ of} \\
\qquad\qquad\qquad\qquad ((2, (x, xs)), y : ys) \to (2, ((x, y), (xs, ys))) \\
\qquad\qquad\qquad\qquad (\_\quad\ \ , \_\quad\ ) \to (1, ())
\end{array}
$$

and consider the composition $zipmap\ f = zip \circ (map\ f \times id)$.

Being function *map* written as:

$$
\begin{array}{l}
map :: (a \to b) \to [\,a\,] \to [\,b\,] \\
map\ f = [\![\psi]\!]_F \\
\quad \textbf{where } \psi\ [\,] \quad\ \ = (1, ()) \\
\qquad\quad \psi\ (a : as) = (2, (f\ a, as))
\end{array}
$$

the result of applying Law 7.2 is:

$$zipmap\ f = [\![\sigma(\psi)]\!]_G$$

which has the equivalent inlined definition:

$$
\begin{array}{l}
zipmap :: (a \to c) \to [\,a\,] \times [\,b\,] \to [\,c \times b\,] \\
zipmap\ f\ (x : xs)\ (y : ys) = (f\ x, y) : zipmap\ f\ xs\ ys \\
zipmap\ f\ \_\quad\ \ \_\quad\ \ = [\,]
\end{array}
$$

□

In [HIT97], the focus was on how to apply fusion on all arguments of a function simultaneously. In contrast, our solution is selective in the argument we want to fuse. Nevertheless, in case the consumer is composed with a product of several producers, like e.g. $zip \circ (map\ f \times map\ g)$, we can simply proceed in various steps by splitting the product: $zip \circ (map\ f \times id) \circ (id \times map\ g)$. Then, in our case, we first fuse $zip \circ (map\ f \times id)$ as in Example 2, and then we fuse the result with $(id \times map\ g)$.

Certain functions that use additional parameters where they hold intermediate results (usually called accumulators), can be represented also as hylomorphisms over multiple arguments. That's the case of functions like *take*, *drop* and *foldl* [Bir98].

Example 7.4 The function *foldl*:

$$
\begin{aligned}
&\mathit{foldl} :: (b \to a \to b) \to b \to [a] \to [b] \\
&\mathit{foldl}\ f\ e\ [\,] = e \\
&\mathit{foldl}\ f\ e\ (x : xs) = \mathit{foldl}\ f\ (f\ e\ x)\ xs
\end{aligned}
$$

can be written in uncurried form as hylomorphism:

$$
\begin{aligned}
&\mathit{foldl}\ f = [\![\mathit{id}, \psi]\!]_G \\
&\quad \textbf{where}\ F = \overline{\mathbf{1}} + \overline{a} \times I \\
&\qquad\quad G = \overline{b} + I \\
&\qquad\quad \psi :: (b \times [a]) \to G\ (b \times [a]) \\
&\qquad\quad \psi\ (e, l) = \textbf{case}\ l\ \textbf{of} \\
&\qquad\qquad\qquad\qquad [\,] \quad\ \to (1, e) \\
&\qquad\qquad\qquad\qquad x : xs \to (2, (f\ e\ x, xs))
\end{aligned}
$$

and we can rewrite ψ as $\sigma(\mathit{out}_F)$:

$$
\begin{aligned}
&\psi = \sigma(\mathit{out}_F) \\
&\quad \textbf{where}\ \sigma :: \forall a.\ (a \to F\ a) \to (b \times a) \to G\ (b \times a) \\
&\qquad\quad \sigma(\beta)\ (e, l) = \textbf{case}\ \beta\ l\ \textbf{of} \\
&\qquad\qquad\qquad\qquad (1, ()) \quad\ \to (1, e) \\
&\qquad\qquad\qquad\qquad (2, (x, xs)) \to (2, (f\ e\ x, xs))
\end{aligned}
$$

where σ abstracts the constructors appearing in the patterns of the second argument of ψ.

Note that we can not derive a σ of type

$$\sigma :: \forall b.\ (b \to F\ b) \to (b \times a) \to G\ (b \times a)$$

which would be the one needed to fuse compositions on the parameter e. The parameter e would be required to have a polymorphic type, but is used in ψ as the first argument of function f which may not be polymorphic. This means that in this case we cannot perform fusion on the accumulator position. □

Because for functions like this we can fuse compositions in some of the recursive arguments only, it is important to have a law allowing us to do so without further ado about the other arguments. This is the subject of the next section.

7.1 Derivation of σ

As part of HFusion we have implemented the treatment of fusion cases involving definitions that make recursion over multiple arguments. The main

modification was in the derivation algorithm for σ. Though we can manipulate functions that have any amount of recursive arguments, and perform fusion on any of them, we will present a simplified version of the algorithm for the sake of clarity. We will show how to derive a transformer σ which enables fusion on the first argument for a definition having two recursive arguments.

The input coalgebra is now expected to be in the form:

$$\begin{aligned}
&\psi :: (a \times b) \to G\ (a \times b)\\
&\psi = \lambda(v_1, v_2) \to \mathbf{case}\ (v_1, v_2)\ \mathbf{of}\\
&\qquad\qquad (p_{11}, p_{12}) \to (1, (t_{11}, ..., t_{1k_1}))\\
&\qquad\qquad ...\\
&\qquad\qquad (p_{m1}, p_{m2}) \to (m, (t_{m1}, ..., t_{mk_m}))
\end{aligned}$$

Every term t_{ij} in a recursive position must be of the form (v, t), where v is a variable being bound in a recursive position of a constructor in pattern p_{i1}, and the second component of those pairs might not reference any such variable. Then, the patterns p_{i1} must conform to the same normal form that we required for patterns in our original derivation algorithm for σ. The precedent examples in this chapter satisfy the restrictions.

Note that the input variables of the coalgebra must be matched in the same order they appear. That's a condition that simplifies presentation of the algorithm, but it certainly could be leveraged by implementing a slightly more complex variant.

Here is the extended algorithm:

$$\begin{aligned}
&\mathcal{S}'(F, \lambda v \to \mathbf{case}\ v\ \mathbf{of}\ (p_{11}, p_{12}) \to t_1; \ldots; (p_{m1}, p_{m2}) \to t_n) =\\
&\qquad \lambda\beta \to \lambda v \to \mathbf{case}\ v\ \mathbf{of}\ (\mathcal{B}(p_{11}), p_{12}) \to t_1; \ldots; (\mathcal{B}(p_{m1}), p_{m2}) \to t_n
\end{aligned}$$

where algorithm $\mathcal{B}$ is the same presented in Section 3.2.4.

The extended algorithm will return a transformer σ of type $\sigma : \forall a.(a \to F\ a) \to (a \times b \to G\ (a \times b))$, such that $\psi = \sigma(out_F)$. Generalizations to a larger amount of arguments can be directly obtained from this scheme.

To illustrate the restriction over the recursive terms, consider the following definition:

$$\begin{aligned}
&zip' :: [a] \to [a] \to [a \times a]\\
&zip'\ (x : xs)\ (y : ys) = (x, y) : zip'\ ys\ xs\\
&zip'\ _\qquad\ _\qquad\ = [\,]
\end{aligned}$$

Note that it differs from *zip* in that it swaps the list tails in the recursive call. If we write its uncurried version as a hylomorphism we have:

$$\begin{aligned}
&zip' = [\![\psi]\!]_G\\
&\quad \mathbf{where}\ G = \overline{\mathbf{1}} + \overline{a \times a} \times I
\end{aligned}$$

$$\begin{aligned}
&\psi :: [a] \times [a] \to G\ ([a] \times [a]) \\
&\psi\ (x : xs, y : ys) = (2, ((x, y), (ys, xs))) \\
&\psi\ (_\quad\ , _\quad\ ) = (1, ())
\end{aligned}$$

If we wanted now to derive $\sigma :: \forall b.\ (b \to F\ b) \to (b \times [a]) \to G\ (b \times [a])$, being $F = \overline{\mathbf{1}} + \overline{a} \times I$ the functor of lists, it couldn't be possible with our algorithm. That is because in the recursive position of the coalgebra the arguments are swapped. If we insist in applying the algorithm we obtain:

$$\begin{aligned}
&zip' = [\![\sigma(out_F)]\!]_F \\
&\quad \textbf{where}\ G = \overline{\mathbf{1}} + \overline{a \times a} \times I \\
&\qquad\quad \sigma :: (b \to F\ b) \to (b \times [a]) \to G\ (b \times [a]) \\
&\qquad\quad \sigma(\beta)\ (xss, yss) = \\
&\qquad\qquad \textbf{case}\ (xss, yss)\ \textbf{of} \\
&\qquad\qquad\quad (\beta{\cdot}(2, (x, xs)), y : ys) \to (2, ((x, y), (ys, xs))) \\
&\qquad\qquad\quad (_\qquad\qquad\ , _\quad\ ) \to (1, ())
\end{aligned}$$

In the above definition we can see that σ does not match the intended type. Notice that variable ys is of type $[a]$ if we look at the pattern where it is bound; nonetheless, when used in a term it is expected to be of polymorphic type b.

7.2 Folds as Unfolds

Let us suppose we have the composition $zf\ p = zip\ (filter\ p\ xs)\ ys$. This composition is problematic because we have that *zip* is a hylomorphism of the form $[\![\phi, \sigma(out_F)]\!]_F$, while there is no way to restructure *filter* p so it shows as an unfold. The best we can restructure *filter* p is as follows:

$$\begin{aligned}
&filter\ p = [\![[\,]\nabla(:)\nabla id, \psi]\!]_F \\
&\quad \textbf{where}\ F = \overline{\mathbf{1}} + \overline{a} \times I + I \\
&\qquad\quad \psi\ [\,]\qquad\ = (1, ()) \\
&\qquad\quad \psi\ (x : xs) = \textbf{if}\ p\ x\ \textbf{then}\ (2, (x, xs)) \\
&\qquad\qquad\qquad\qquad\ \ \textbf{else}\ (3, xs)
\end{aligned}$$

However, we could achieve fusion in this case if we rewrite the definitions in this form:

$$\begin{aligned}
&\textbf{data}\ [a] = [\,] \mid (:)\ a\ [a] \mid D\ [a] \\
&zip :: [a] \to [b] \to [a \times b] \\
&zip\ (x : xs)\ (y : ys) = (x, y) : zip\ xs\ ys \\
&zip\ (D\ xs)\ \ ys\qquad\ = zip\ xs\ ys \\
&zip\ _\qquad\ _\qquad\ = [\,] \\
&filter : (a \to Bool) \to [a] \to [a]
\end{aligned}$$

$$
\begin{array}{l}
\mathit{filter}\ p\ [\,] \qquad = [\,] \\
\mathit{filter}\ p\ (x : xs) = \ \mathbf{if}\ p\ x \\
\qquad\qquad\qquad\qquad\ \ \mathbf{then}\ x : \mathit{filter}\ p\ xs \\
\qquad\qquad\qquad\qquad\ \ \mathbf{else}\ D\ (\mathit{filter}\ p\ xs)
\end{array}
$$

We have introduced an artificial constructor D in the type of lists, and then extended our definitions to make use of it.

We now can proceed to derive the corresponding hylomorphisms and the transformer σ obtaining:

$$
\begin{array}{l}
F = \overline{\mathbf{1}} + \overline{a} \times I + I \\
\mathit{zip} :: [\,a\,] \times [\,b\,] \to [\,a \times b\,] \\
\mathit{zip} = [\![\phi, \sigma(\mathit{out}_F)]\!]_G \\
\quad \mathbf{where}\ G = \overline{\mathbf{1}} + \overline{a \times b} \times I + I \\
\qquad\quad \sigma(\beta)\ (xss, yss) = \\
\qquad\qquad\quad \mathbf{case}\ (\beta\ xss, yss)\ \mathbf{of} \\
\qquad\qquad\qquad\quad ((2, (x, xs)), y : ys) \to (2, ((x, y), (xs, ys))) \\
\qquad\qquad\qquad\quad ((3, xs), ys) \qquad\ \to (3, (xs, ys)) \\
\qquad\qquad\qquad\quad (\_, \_) \qquad\qquad\ \to (1, ()) \\
\qquad\qquad \phi\ (1, ()) \qquad = [\,] \\
\qquad\qquad \phi\ (2, (x, xs)) = x : xs \\
\qquad\qquad \phi\ (3, xs) \qquad = xs \\
\mathit{filter}\ p = [\![\mathit{in}_F, \psi]\!]_F
\end{array}
$$

In the above definitions, notice how the occurrences of the constructor D only appear inside in_F and out_F. Now we can perform fusion since with the introduction of D filter is an unfold. As result of fusion we get $[\![\phi, \sigma(\psi)]\!]_F$, which does no longer hold D!

Inlining the result we get

$$
\begin{array}{l}
\mathit{zf}\ p\ [\,]\ \_ = [\,] \\
\mathit{zf}\ p\ (x : xs)\ yss = \mathbf{if}\ p\ x \\
\qquad\qquad\qquad\qquad\ \ \mathbf{then\ case}\ yss\ \mathbf{of} \\
\qquad\qquad\qquad\qquad\qquad\quad (y : ys) \to (x : y) : \mathit{zf}\ p\ xs\ ys \\
\qquad\qquad\qquad\qquad\qquad\quad \_ \to [\,] \\
\qquad\qquad\qquad\qquad\ \ \mathbf{else}\ \mathit{zf}\ p\ xs\ yss
\end{array}
$$

Inserting an artificial constructor like we have done here works well for a couple of fortunate coincidences. First, the algebra components of the original *filter* definition are either a constructor or the identity function. And second, the patterns used by *zip* on the first argument (i.e. $x : xs$ and $\_$) do not contain nested constructor applications in recursive positions. Provided that these two conditions hold for any pair of definitions, the trick can be played again. In fact, we have implemented this technique in HFusion.

7.3 Flexibility through mutual hylos

Some definitions that recurse over multiple arguments may seem not fusable at first, but they can be fused if they are conveniently rewritten using mutual recursion.

Consider, for example, the case of zip' for which we could not derive a σ transformer in Section 7.1. The problem can be overcome by rewriting the original definition in terms of mutual recursion.

$$
\begin{array}{l}
zip'_1 :: [a] \to [a] \to [a \times a] \\
zip'_1\ (x : xs)\ (y : ys) = (x, y) : zip'_2\ ys\ xs \\
zip'_1\ \_\ \ \ \_\ \ \ = [\,] \\
zip'_2 :: [a] \to [a] \to [a \times a] \\
zip'_2\ (x : xs)\ (y : ys) = (x, y) : zip'_1\ ys\ xs \\
zip'_2\ \_\ \ \ \_\ \ \ = [\,]
\end{array}
$$

In this new definition zip'_1 and zip'_2 mirror the definitions of zip' while calling to each other. Writing (zip'_1, zip'_2) as a mutual unfold, we obtain:

$$
\begin{array}{l}
(zip'_1, zip'_2) = [\![\psi]\!]_{\langle G_1, G_2\rangle} \\
\quad \textbf{where}\ G_1 = \overline{\mathbf{1}} + (a \times a) \times \Pi_2 \\
\qquad\quad\ \ G_2 = \overline{\mathbf{1}} + (a \times a) \times \Pi_1 \\
\qquad\quad\ \ \psi :: ([a] \times [a], [a] \times [a]) \to \langle G_1, G_2\rangle\ ([a] \times [a], [a] \times [a]) \\
\qquad\quad\ \ \psi = (\psi_1, \psi_2) \\
\qquad\quad\ \ \psi_1 :: [a] \times [a] \to G_1\ ([a] \times [a], [a] \times [a]) \\
\qquad\quad\ \ \psi_1\ (xss, yss) = \textbf{case}\ (xss, yss)\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad (x : xs, y : ys) \to (2, ((x, y), (ys, xs))) \\
\qquad\qquad\qquad\qquad\qquad (\_\ \ \ , \_\ \ \ ) \to (1, ()) \\
\qquad\quad\ \ \psi_2 :: [a] \times [a] \to G_2\ ([a] \times [a], [a] \times [a]) \\
\qquad\quad\ \ \psi_2\ (xss, yss) = \textbf{case}\ (xss, yss)\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad (x : xs, y : ys) \to (2, ((x, y), (ys, xs))) \\
\qquad\qquad\qquad\qquad\qquad (\_\ \ \ , \_\ \ \ ) \to (1, ())
\end{array}
$$

Suppose we want to fuse $zm'\ f = zip' \circ map\ f$. In order to use the definition of zip'_1, we rewrite the composition as $(zm'_1, zm'_2) = (zip'_1, zip'_2) \circ map'\ f$ where we define $map'\ f$) as follows:

$$
\begin{array}{l}
map'\ f = [\![(\psi', \psi')]\!]_F \\
\quad \textbf{where}\ F = \langle \overline{\mathbf{1}} + \overline{a} \times \Pi_2, \overline{\mathbf{1}} + \overline{a} \times \Pi_1 \rangle \\
\qquad\quad\ \ \psi'\ (x : xs) = (2, (f\ x, xs)) \\
\qquad\quad\ \ \psi'\ [\,]\ \ \ \ \ \ = (1, ())
\end{array}
$$

The mutual unfold $map'\ f$ can be derived from $map\ f$ with a technique like that shown in Section 6.5.

Now, σ can be derived from the coalgebra of (zip'_1, zip'_2):

$$
\begin{aligned}
&(zip'_1, zip'_2) = [\![\sigma(out_F)]\!]_{\langle G_1, G_2\rangle} \\
&\quad \textbf{where } \sigma(\beta) = (\sigma_1(\beta), \sigma_2(\beta)) \\
&\qquad\quad \sigma_1 :: \forall b_1 b_2.\ ((b_1, b_2) \to F\ (b_1, b_2)) \to \\
&\qquad\qquad\quad (b_1 \times [a], [a] \times b_2) \to G_1\ (b_1 \times [a], [a] \times b_2) \\
&\qquad\quad \sigma_1(\beta_1, \beta_2)\ (xss, yss) = \textbf{case } (xss, yss) \textbf{ of} \\
&\qquad\qquad\quad (\beta_1 \cdot (2, (x, xs)), y : ys) \to (2, ((x, y), (ys, xs))) \\
&\qquad\qquad\quad (\_\qquad\qquad\quad, \_\qquad) \to (1, ()) \\
&\qquad\quad \sigma_2 :: \forall b_1 b_2.\ ((b_1, b_2) \to F\ (b_1, b_2)) \to \\
&\qquad\qquad\quad (b_1 \times [a], [a] \times b_2) \to G_2\ (b_1 \times [a], [a] \times b_2) \\
&\qquad\quad \sigma_2(\beta_1, \beta_2)\ (xss, yss) = \textbf{case } (xss, yss) \textbf{ of} \\
&\qquad\qquad\quad (x : xs, \beta_2 \cdot (2, (y, ys))) \to (2, ((x, y), (ys, xs))) \\
&\qquad\qquad\quad (\_\qquad, \_\qquad\qquad\quad) \to (1, ())
\end{aligned}
$$

By applying Acid Rain for the hylo-unfold case, we obtain:

$$[\![\sigma(\psi', \psi')]\!]_{\langle G_1, G_2\rangle}$$

Inlined, the result is:

$$
\begin{aligned}
&zm'_1 :: (a \to b) \to [a] \to [b] \to [b \times b] \\
&zm'_1\ f\ (x : xs)\ (y : ys) = (f\ x, y) : zm'_2\ f\ ys\ xs \\
&zm'_1\ f\ \_\qquad\ \_\qquad\ = [\,] \\
&zm'_2 :: (a \to b) \to [b] \to [a] \to [b \times b] \\
&zm'_2\ f\ (x : xs)\ (y : ys) = (x, f\ y) : zm'_1\ f\ ys\ xs \\
&zm'_2\ f\ \_\qquad\ \_\qquad\ = [\,]
\end{aligned}
$$

We are working here with mutual hylomorphisms that recurse over multiple arguments. The needed algorithms are a mix of the variations introduced to the original algorithms both for mutual recursion and recursion over multiple arguments, so they can be built from the pieces we have provided with moderate effort.

Another problematic definition is

$$
\begin{aligned}
&zip'' :: [a] \to [a] \to [a \times a] \\
&zip''\ (x : xs)\ (y : ys) = (x, y) : zip''\ xs\ xs \\
&zip''\ \_\qquad\ \_\qquad\ = [\,]
\end{aligned}
$$

Written as unfold we have:

$$
\begin{aligned}
&zip'' = [\![\psi]\!]_F \\
&\quad \textbf{where } F = \overline{\mathbf{1}} + (a \times a) \times I \\
&\qquad\quad \psi\ (x : xs, y : ys) = (2, ((x, y), (xs, xs))) \\
&\qquad\quad \psi\ (\_\qquad, \_\qquad) = (1, ())
\end{aligned}
$$

For similar reasons to the previous case, the coalgebra cannot be used as input to derive a transformer σ. However, we can rewrite the definition as follows:

$$
\begin{aligned}
&zip''_1 :: [a] \to [a] \to [a \times a] \\
&zip''_1\ (x : xs)\ (y : ys) = (x, y) : zip''_2\ xs\ xs \\
&zip''_1\ \_\ \quad \_\ \quad = [\,] \\
&zip''_2 :: [a] \to [a] \to [a \times a] \\
&zip''_2\ (x : xs)\ (y : ys) = (x, y) : zip''_2\ xs\ xs \\
&zip''_2\ \_\ \quad \_\ \quad = [\,]
\end{aligned}
$$

If we wanted to fuse $zm'' = (zip''_1, zip''_2) \circ map'\ f$, we could derive σ in a mutual unfold:

$$
\begin{aligned}
&(zip''_1, zip''_2) = [\![\sigma out.]\!]_{\langle G_1, G_1 \rangle} \\
&\quad \textbf{where}\ G_1 = \overline{\mathbf{1}} + (a \times a) \times \Pi_2 \\
&\qquad\quad \sigma(\beta) = (\sigma_1(\beta), \sigma_2(\beta)) \\
&\qquad\quad \sigma_1 :: \forall b_1 b_2.\ ((b_1, b_2) \to F\ (b_1, b_2)) \to \\
&\qquad\qquad\quad (b_1 \times [a], b_2 \times b_2) \to G_1\ (b_1 \times [a], b_2 \times b_2) \\
&\qquad\quad \sigma_1(\beta_1, \beta_2)\ (xss, yss) = \textbf{case}\ (xss, yss)\ \textbf{of} \\
&\qquad\qquad\qquad\quad (\beta_1 \cdot (x, xs), y : ys) \to (2, ((x, y), (xs, xs))) \\
&\qquad\qquad\qquad\quad (\_\ \quad\quad , \_\ \quad) \to (1, ()) \\
&\qquad\quad \sigma_2 :: \forall b_1 b_2.\ ((b_1, b_2) \to F\ (b_1, b_2)) \to \\
&\qquad\qquad\quad (b_1 \times [a], b_2 \times b_2) \to G_1\ (b_1 \times [a], b_2 \times b_2) \\
&\qquad\quad \sigma_2(\beta_1, \beta_2)\ (xss, yss) = \textbf{case}\ (xss, yss)\ \textbf{of} \\
&\qquad\qquad\qquad\quad (\beta_2 \cdot (x, xs), \beta_2 \cdot (y, ys)) \to (2, ((x, y), (xs, xs))) \\
&\qquad\qquad\qquad\quad (\_\ \quad\quad , \_\ \quad\quad) \to (1, ())
\end{aligned}
$$

As a result of applying Acid Rain for the hylo-unfold case, we obtain:

$$[\![\sigma(\psi', \psi')]\!]_{\langle G_1, G_1 \rangle}$$

And inlining, we arrive to the following definition:

$$
\begin{aligned}
&zm''_1 :: (a \to b) \to [a] \to [b] \to [b \times b] \\
&zm''_1\ f\ (x : xs)\ (y : ys) = (f\ x, y) : zm''_2\ f\ xs\ xs \\
&zm''_1\ f\ \_\ \quad \_\ \quad = [\,] \\
&zm''_2 :: (a \to b) \to [a] \to [a] \to [b \times b] \\
&zm''_2\ f\ (x : xs)\ (y : ys) = (f\ x, f\ y) : zm''_2\ f\ xs\ xs \\
&zm''_2\ f\ \_\ \quad \_\ \quad = [\,]
\end{aligned}
$$

Through these (rather contrived) examples, we argue that, by combining mutual recursion and recursion over multiple arguments, it is possible to have more flexible algorithms than the ones given for the separated schemes. When deriving a coalgebra transformer, HFusion is able to derive mutually recursive definitions to enable fusion as shown above.

7.4 Duplication of computations

When mixing mutual recursion and recursion over multiple arguments, we have found some cases where computations are duplicated after fusion.

Looking at the definition of zm''_2 from the previous section, we can see that now f is being applied twice over the same element which may be referenced by both x and y in the right hand side of the first equation of zm''_2. This is an unfortunate duplication, in order to avoid it, we should modify the definition of zip''_2 to use only one parameter:

$$
\begin{aligned}
&zip''_1 :: [a] \to [a] \to [a \times a] \\
&zip''_1\ (x : xs)\ (y : ys) = (x, y) : zip''_2\ xs \\
&zip''_1\ \_\ \quad \_\ \quad = [\,] \\
&zip''_2 :: [a] \to [a \times a] \\
&zip''_2\ (x : xs)\ = (x, x) : zip''_2\ xs \\
&zip''_2\ \_\ \quad = [\,]
\end{aligned}
$$

Performing the fusion with this definition should yield:

$$
\begin{aligned}
&zm''_1 :: (a \to b) \to [a] \to [b] \to [b \times b] \\
&zm''_1\ f\ (x : xs)\ (y : ys) = (f\ x, y) : zm''_2\ f\ xs \\
&zm''_1\ f\ \_\ \quad \_\ \quad = [\,] \\
&zm''_2 :: (a \to b) \to [a] \to [b \times b] \\
&zm''_2\ f\ (x : xs)\ = \mathbf{let}\ fx = f\ x\ \mathbf{in}\ (fx, fx) : zm''_2\ f\ xs \\
&zm''_2\ f\ \_\ \quad = [\,]
\end{aligned}
$$

This does not duplicate the application of f.

Another case of duplicated computations is obtained by fusing

$$zff\ p\ p'\ xs\ ys = zf\ p\ xs\ (filter\ p'\ ys)$$

where zf is the definition we obtained in Section 7.2. That yields:

$$
\begin{aligned}
&zff\ p\ p'\ [\,]\ yss \qquad\qquad\quad = [\,] \\
&zff\ p\ p'\ xss@(x : xs)\ yss = \\
&\quad \mathbf{if}\ p\ x \\
&\qquad \mathbf{then\ case}\ yss\ \mathbf{of} \\
&\qquad\qquad [\,] \to [\,] \\
&\qquad\qquad y : ys \to \mathbf{if}\ p'\ y \\
&\qquad\qquad\qquad\qquad \mathbf{then}\ (x, y) : zff\ p\ p'\ xs\ ys \\
&\qquad\qquad\qquad\qquad \mathbf{else}\ zff\ p\ p'\ xss\ ys \\
&\qquad \mathbf{else}\ zff\ p\ p'\ xs\ yss
\end{aligned}
$$

Some terms may be evaluated more than once. Following the flow of evaluation when $p\ x$ evaluates to true and $p'\ y$ evaluates to false, it can be seen

that $p\ x$ may be evaluated again. There is a way to rewrite this definition to avoid the duplication, which consists in using mutual recursion.

$$
\begin{aligned}
&\mathit{zff}\ p\ p'\ [\,]\ \mathit{yss} &&= [\,]\\
&\mathit{zff}\ p\ p'\ (x : \mathit{xs})\ \mathit{yss} &&= \mathbf{if}\ p\ x\ \mathbf{then}\ \mathit{zff}'\ p\ p'\ x\ \mathit{xs}\ \mathit{yss}\\
&&&\quad\ \mathbf{else}\ \mathit{zff}\ p\ p'\ \mathit{xs}\ \mathit{yss}\\
&\mathit{zff}'\ p\ p'\ x\ \mathit{xs}\ [\,] &&= [\,]\\
&\mathit{zff}'\ p\ p'\ x\ \mathit{xs}\ (y : \mathit{ys}) &&= \ \mathbf{if}\ p'\ y\ \mathbf{then}\ (x, y) : \mathit{zff}\ p\ p'\ \mathit{xs}\ \mathit{ys}\\
&&&\quad\ \mathbf{else}\ \mathit{zff}'\ p\ p'\ x\ \mathit{xs}\ \mathit{ys}
\end{aligned}
$$

This definition can be obtained by rewriting *zip* as shown below, and performing then the fusion on each of its arguments.

$$
\begin{aligned}
&\mathit{zip} :: [\,a\,] \to [\,b\,] \to [\,a \times b\,]\\
&\mathit{zip}\ (x : \mathit{xs})\ \mathit{yss} = \mathit{zip}'\ x\ \mathit{xs}\ \mathit{yss}\\
&\mathit{zip}\ \_\quad\ \_\quad = [\,]\\
&\mathit{zip}'\ x\ \mathit{xs}\ (y : \mathit{ys}) = (x, y) : \mathit{zip}\ \mathit{xs}\ \mathit{ys}
\end{aligned}
$$

Rewriting definitions using mutual recursion to avoid duplicating computations lends to automation. This transformation has not been implemented in HFusion yet.

7.5 A normal form for recursive definitions

Interestingly enough, the techniques and problems stated in the previous sections hint for a normal form to which recursive program definitions should be rewritten in order to have better chances to be fused. Though we have not implemented such rewriting yet, it could be automated and may contribute to simplify the implementation.

We describe the normal form through the following rules.

Rule 1 Using mutual recursion, definitions which recurse over multiple arguments should be split into definitions that do not make pattern matching over more than one argument at a time. That is, all equations should be in the form:

$$f\ p_1\ \ldots\ p_n = \ldots$$

where there is at most one index i such that p_i is not a variable.

Definitions which make recursion over multiple arguments are to be converted to the form above much in the same way as we did with *zip* in the previous section.

Rule 2 Definitions which contain nested constructor applications in the algebra should be rewritten using mutual recursion to eliminate the nesting if any of the arguments of those constructors reference recursive variables. That is, all equations should be in the form $f\ p_1\ \ldots\ p_n = g$, where g is in the following normal form:

- it is the result of a recursive call; or
- it is a term in the form $C\ t_1\ \ldots\ t_m$ where C is a constructor and each t_i is either a recursive call, or a term not referencing recursive calls; or
- it is a term in the form **case** t_0 **of** $p_1 \to t_1; \ldots; p_m \to t_m$ where $t_1, \ldots, t_m$ are in this normal form, and t_0 does not reference variables used as arguments of recursive calls.

To illustrate this rule consider the following definition:

$$
\begin{aligned}
&\mathit{intersperse} :: a \to [a] \to [a]\\
&\mathit{intersperse}\ e\ [\,] = [\,]\\
&\mathit{intersperse}\ e\ (x : xs) = x : e : \mathit{intersperse}\ e\ xs
\end{aligned}
$$

This definition can be written as a fold, but not as an unfold because of the nested application of constructor (:) in the second equation. However we can rewrite this definition as

$$
\begin{aligned}
&\mathit{intersperse}\ e\ [\,] = [\,]\\
&\mathit{intersperse}\ e\ (x : xs) = x : \mathit{intersperse}'\ e\ xs\\
&\mathit{intersperse}'\ e\ xs = e : \mathit{intersperse}\ e\ xs
\end{aligned}
$$

And now we can write this definition as an unfold

$$
\begin{aligned}
&\mathit{intersperse}\ e = [\![(\psi_1, \psi_2)]\!]_F\\
&\quad \textbf{where}\ F = \langle \overline{\mathbf{1}} + \overline{a} \times \Pi_2, \overline{a} \times \Pi_1 \rangle\\
&\qquad\quad \psi_1\ [\,] = (1, ())\\
&\qquad\quad \psi_1\ (x : xs) = (2, (x, xs))\\
&\qquad\quad \psi_2\ xs = (1, (e, xs))
\end{aligned}
$$

without missing the ability to write it as a fold

$$
\begin{aligned}
&\mathit{intersperse}\ e = (\!|(\phi_1, \phi_2)|\!)_F\\
&\quad \textbf{where}\ F = \langle \overline{\mathbf{1}} + \overline{a} \times \Pi_2, \Pi_1 \rangle\\
&\qquad\quad \phi_1\ (1, ()) = [\,]\\
&\qquad\quad \phi_1\ (2, (x, xs)) = x : xs\\
&\qquad\quad \phi_2\ (1, xs) = e : xs
\end{aligned}
$$

Rule 3 Dually to the previous rule, definitions which would contain nested constructor applications in the coalgebra patterns should be rewritten with mutual recursion to eliminate the nesting if any of the arguments of those constructors are referenced in the arguments of recursive calls. That is, all equations should be in the form

$$f\ p_1\ \ldots\ p_n = g$$

where each p_i is either a variable or a pattern in the form $C\ p'_1\ \ldots\ p'_m$ where each p'_i is a variable referenced in a recursive call or it is not a recursive position of C. To illustrate this requirement, consider the following definition:

$$\begin{aligned}
&odds :: [a] \to [a]\\
&odds\ [\,] \ \ = [\,]\\
&odds\ [x] = [x]\\
&odds\ (x : _ : xs) = x : odds\ xs
\end{aligned}$$

We cannot fuse $odds \circ filter\ p$ using the technique in Section 7.2 because of the nesting of constructor (:) in the pattern of the second equation. However, we can rewrite this definition as the mutual $(even, odd)$ appearing in Section 6.5, which is a mutual fold and can be fused with *filter*.

It remains to be examined in more detail whether all definitions that can be fused with the current algorithms can be also expressed using this normal form. The main benefit of fusing programs in this form is that the hylo-unfold law becomes applicable in most of the cases. There won't be any composition $[\![\phi, \sigma(out_F)]\!]_F \circ [\![\tau(in_F), \psi]\!]_F$ that could prevent Acid Rain to be applied, provided that the involved hylos are properly restructured.

Chapter 8

Measuring fusion

In this chapter we will compare the performance of some sample programs with the performance of their fused versions. Our aim is to provide evidence that our transformations can improve programs.

We compiled all of our programs with the GHC compiler version 6.8.3, feeding it with the `-O` option, except when explicitly stating otherwise. At this stage, the GHC compiler implements a fusion technique known as short-cut fusion, which is an approach which subsumes ours (see Section 9.5). But this GHC implementation can only fuse compositions of list functions defined in the standard libraries.

8.1 Deforesting all the data structures

We start with a program building and traversing binary trees.

$$
\begin{aligned}
&\textbf{data}\ \mathit{Tree}\ a = \mathit{Node}\ a\ (\mathit{Tree}\ a)\ (\mathit{Tree}\ a) \mid \mathit{Empty}\\
&\mathit{main} :: \mathit{IO}\ ()\\
&\mathit{main} = \mathit{print}\ (\mathit{sumTree}\ \$\ \mathit{mapTree}\ (1+)\ \$\ \mathit{genTree}\ 23)\\
&\mathit{mapTree} :: (a \to b) \to \mathit{Tree}\ a \to \mathit{Tree}\ b\\
&\mathit{mapTree}\ f\ \mathit{Empty} = \mathit{Empty}\\
&\mathit{mapTree}\ f\ (\mathit{Node}\ a\ l\ r) = \mathit{Node}\ (f\ a)\ (\mathit{mapTree}\ f\ l)\ (\mathit{mapTree}\ f\ r)\\
&\mathit{sumTree} :: \mathit{Tree}\ \mathit{Int} \to \mathit{Int}\\
&\mathit{sumTree}\ \mathit{Empty} = 0\\
&\mathit{sumTree}\ (\mathit{Node}\ i\ l\ r) = i + \mathit{sumTree}\ l + \mathit{sumTree}\ r\\
&\mathit{genTree} :: \mathit{Int} \to \mathit{Tree}\ \mathit{Int}\\
&\mathit{genTree}\ 0 = \mathit{Empty}\\
&\mathit{genTree}\ i = \mathit{Node}\ i\ (\mathit{genTree}\ (i-1))\ (\mathit{genTree}\ (i-1))
\end{aligned}
$$

And here's the fused program

$$
\begin{aligned}
&\mathit{main} :: \mathit{IO}\ ()\\
&\mathit{main} = \mathit{print}\ (\mathit{sumTree\_mapTree\_genTree}\ (1+)\ 23)
\end{aligned}
$$

```
sumTree_mapTree_genTree :: (Int → Int) → Int → Int
sumTree_mapTree_genTree f 0 = 0
sumTree_mapTree_genTree f v2 =
  f v2 + sumTree_mapTree_genTree f (v2 − 1) +
    sumTree_mapTree_genTree f (v2 − 1)
```

- Original program. Total time: 9.87 seconds. Total bytes allocated: 1,636,009,068. Maximum residency bytes: 40,960. GC time 62.8%.
- Fused program. Total time: 0.002 seconds. Total bytes allocated: 35,768. Maximum residency bytes: 45.056. GC time 0%.

The difference in performance is so abysmal that we also provide the measures of the programs compiled without optimizations (that is without `-O`):

- Original program. Total time: 13.63 seconds. Total bytes allocated: 2,903,416,884. Maximum residency bytes: 45,056. GC time 49.3%.
- Fused program. Total time: 4.03 seconds. Total bytes allocated: 572,992. Maximum residency bytes: 40.960. GC time 2.8%.

The results are so good because we are completely eliminating the need to produce trees. Also, from the first test we can infer that the compiler is well prepared to optimize the output of the fusion.

8.2 A normalizer for lambda calculus

This is a program that normalizes untyped lambda expressions. In contrast with the previous program the intermediate data structures cannot be fully eliminated due to a complex recursion pattern used by function *eval*.

```
import List (union)
main :: IO ()
main = print (eval 500000 (genexp (freshvars 0) "v" n))
  -- Abstract syntax for lambda expressions
type Variable = String
data Exp  = Eapp Exp Exp          -- e e
          | Elamb Variable Exp    -- λv.e
          | Evar Variable         -- v
  deriving Show
  -- Substitution of of variables by terms
subst :: [(Variable, Exp)] → Exp → Exp
subst ls e@(Evar var) = case lookup var ls of
                          Just e → e
                          Nothing → e
subst ls (Elamb v e)  = Elamb v (subst (filter ((v ≠) ∘ fst) ls) e)
subst ls (Eapp e0 e1) = Eapp (subst ls e0) (subst ls e1)
  -- Alpha conversions
  -- The first parameter indicates the variables in scope (bounded).
alphaConv :: [Variable] → [(Variable, Variable)] → Exp → Exp
alphaConv bs ls (Evar v) =
```

```
  if elem v bs
    then case lookup v ls of
           Just v' → Evar v'
           Nothing → Evar v
    else Evar v
alphaConv bs ls (Elamb v e) =
  case lookup v ls of
    Just v' → Elamb v' (alphaConv (v : bs) ls e)
    Nothing → Elamb v (alphaConv bs ls e)
alphaConv bs ls (Eapp e0 e1) = Eapp (alphaConv bs ls e0) (alphaConv bs ls e1)

  -- Free variables of an expression
  -- The first parameter indicates the variables in scope (bounded).
fv :: [Variable] → Exp → [Variable]
fv bs (Evar v) | elem v bs = [v]
               | otherwise = []
fv bs (Elamb v e) = fv (ins v bs) e
  where ins a as | elem a as = as
                 | otherwise = a : as
fv bs (Eapp e0 e1) = union (fv bs e0) (fv bs e1)

  -- a list of fresh variables, the parameter is the seed.
freshvars :: Int → [String]
freshvars i = ("u" ++ show i) : freshvars (i + 1)

  -- normalizing function
eval :: Int → Exp → Exp
eval gen (Eapp e0 e1) =
  case eval gen e0 of
    Elamb v e0' → eval gen $ subst [(v, e1)] $ alphaConv [] (zip (fv [] e1) (freshvars gen)) e0'
    e0' →  Eapp e0' (eval gen e1)
eval gen (Elamb v e) = Elamb v (eval gen e)
eval _ e = e

  -- a generator of expressions
genexp :: [Variable] → Variable → Int → Exp
genexp us v 0       = Evar v
genexp (u : us) v n = Eapp (Elamb u (genexp us u (n − 1))) (Evar v)
```

And here's the fused program

```
main :: IO ()
main = print (eval' 500000 (genexp (freshvars 0) "v" n))

  -- fused normalization function
eval' :: Int → Exp → Exp
eval' gen (Eapp e0 e1) =
  case eval' gen e0 of
    Elamb v e0' → eval' gen (subst_alphaConv [(v, e1)] [] (zip_freshvars (fv [] e1) gen) e0')
    e0' → Eapp e0' (eval' gen e1)
eval' gen (Elamb v e)  = Elamb v (eval' gen e)
eval' _ e              = e

zip_freshvars (a : as) i4 = (a, "u" ++ show i4) : zip_freshvars as (i4 + 1)
zip_freshvars v0 i4       = []

subst_alphaConv ls bs25 ls45 v65@(Evar v66) =
  if elem v66 bs25
    then case lookup v66 ls45 of
           Just v' →
             case lookup v' ls of
               Just e → e
               Nothing → alphaConv bs25 ls45 v65
           Nothing →
             case lookup v66 ls of
               Just e → e
               Nothing → alphaConv bs25 ls45 v65
    else case lookup v66 ls of
           Just e → e
           Nothing → alphaConv bs25 ls45 v65
```

```
subst_alphaConv ls bs25 ls45 (Elamb v67 e32) =
  case lookup v67 ls45 of
    Just v' →
      Elamb v' (subst_alphaConv (filter ((v' ≠) ∘ fst) ls) (v67 : bs25) ls45 e32)
    Nothing →
      Elamb v67 (subst_alphaConv (filter ((v67 ≠) ∘ fst) ls) bs25 ls45 e32)
subst_alphaConv ls bs25 ls45 (Eapp e33 e34) =
  Eapp (subst_alphaConv ls bs25 ls45 e33) (subst_alphaConv ls bs25 ls45 e34)
```

- Original program. Total time: 21.45 seconds. Total bytes allocated: 385, 059, 788. Maximum residency bytes: 106,532,864. GC time 97%.
- Fused program. Total time: 19.52 seconds. Total bytes allocated: 383,369,032. Maximum residency bytes: 118,153,216. GC time 96.1%.

The improvement is more modest in this case. We should also mention that the lambda expression generated with *genexp* has been carefully chosen to exercise the part of the code that was fused. Having used other lambda expressions the improvement would be as noticeable as the amount of time that is dedicated to execution of the fused part of the program.

8.3 Fusion of list comprehensions

This is an example inspired in the one given by [CLS07]. We have made it slightly more elaborated to match better HFusion power.

$$main = print\ (sum\ [k * m \mid k \leftarrow [1\,..\,14000], m \leftarrow [1\,..\,k], mod\ k\ m == 0])$$

In order to fuse this program we desugar the list comprehension as follows:

```
main = print (sum' 0 (concat
                (map (λk → map (k*)
                                (filter (λm → mod k m == 0) (upto 1 k)))
                (upto 1 14000))))
sum' acc [] = acc
sum' acc (x : xs) = sum' (acc + x) xs
upto m n = case m > n of
  True → []
  False → m : upto (m + 1) n
```

In an intermediate phase of the transformation, we obtain the following program:

```
main = print (sumConcatMapUpto
                (λk → mapFilterUpto (k*) (λm → mod k m == 0) k 1)
                14000 0 1))
```

```
sumConcatMapUpto f n acc m =
  if m > n then acc
    else sumAppendMapUpto f n acc (m + 1) (f m)
sumAppendMapUpto f n acc m (x : xs) =
  sumAppendMapUpto f n (x + acc) m xs
sumAppendMapUpto f n acc m []       = sumConcatMapUpto f n acc m
mapFilterUpto f p n m =
  if m > n then []
    else if p m then f m : mapFilterUpto f p n (m + 1)
           else mapFilterUpto f p n (m + 1)
```

To proceed with fusion, we transform the program by inlining the argument f of *sumConcatMapUpto* and make a β-reduction:

```
main = print (sumConcatMapUpto 14000 0 1)
sumConcatMapUpto f n acc m =
  if m > n then acc
    else sumAppendMapUpto n acc (m + 1)
           (mapFilterUpto (m∗) (λm′ → mod m m′ == 0) m 1)
sumAppendMapUpto n acc m (x : xs) =
  sumAppendMapUpto n (x + acc) m xs
sumAppendMapUpto n acc m []       = sumConcatMapUpto n acc m
```

In order to obtain the final fused program, we fuse the remaining composition *sumAppendMapUpto* n $(m+1)$ (*mapFilterUpto* ...) in the definition of *sumConcatMapUpto*.

```
main = print (sumConcatMapUpto 14000 0 1)
sumConcatMapUpto n acc m =
  if m > n then acc
    else g n (m + 1) (m∗) (λm′ → mod m m′ == 0) m acc 1
g n m f p n′ acc m′ =
  if m′ > n′ then sumConcatMapUpto n acc m
    else if p m′
      then g n m f p n′ (acc + f m′) (m′ + 1)
      else g n m f p n′ acc (m′ + 1)
```

- Original program. Total time: 24.5 seconds. Total bytes allocated: $7,570,377,604$. Maximum residency bytes: $172,032$. GC time 2.1%.
- Fused program. Total time: 15.3 seconds. Total bytes allocated: $4,766,983,652$. Maximum residency bytes: $40,960$. GC time 2.5%.

In this example we are performing better than the shortcut fusion implementation of GHC. That's most likely for the fact that it is difficult to fuse $sum' [] \circ f$, where function f is the result of fusing some functions including *filter* without a technique like the one presented in Section 7.2.

8.4 Fusion of primitive recursive functions

Now, let us consider a program involving the insertion in binary search trees, which is a paramorphism.

```
main = print (countNodes (insert 30 (genTree 24)))

countNodes Empty          = 0
countNodes (Node n t1 t2) = 1 + countNodes t1 + countNodes t2

insert :: Ord a ⇒ a → Tree a → Tree a
insert x Empty          = Node x Empty Empty
insert x (Node a t1 t2) = case a < x of
                            True → Node a t1 (insert x t2)
                            False → Node a (insert x t1) t2
```

As an intermediate step, we fuse *countNodes* (*insert* 30 (*genTree* 24))) to obtain:

```
main = print (countNodes_insert_genTree 30 24)

countNodes_insert_genTree x 0  =
  1 + countNodes Empty + countNodes Empty
countNodes_insert_genTree x v0 =
  if v0 < x
    then 1 + countNodes (genTree (v0 − 1)) +
      countNodes_insert_genTree x (v0 − 1)
    else 1 + countNodes_insert_genTree x (v0 − 1) +
      countNodes (genTree (v0 − 1))
```

In order to obtain the final fused program, we must fuse the composition *countNodes* (*genTree* (*v0* − 1)) in the body of *countNodes_insert_genTree*.

```
main = print (countNodes_insert_genTree 30 24)

countNodes_insert_genTree x 0  =
  1 + countNodes Empty + countNodes Empty
countNodes_insert_genTree x v0 =
  if v0 < x
    then 1 + countNodes_genTree (v0 − 1) +
      countNodes_insert_genTree x (v0 − 1)
    else 1 + countNodes_insert_genTree x (v0 − 1) +
      countNodes_genTree (v0 − 1)

countNodes_genTree 0  = 0
countNodes_genTree v0 =
  1 + countNodes_genTree (v0 − 1) + countNodes_genTree (v0 − 1)
```

- Original program. Total time: 9.44 seconds. Total bytes allocated: 1, 627, 291, 224. Maximum residency bytes: 40, 960. GC time 62%.

- Fused program. Total time: 0.001 seconds. Total bytes allocated: 45,064. Maximum residency bytes: 45,056. GC time 0%.

Deforesting *insert* is not a great optimization since it only reconstructs a single branch in a tree with a maximum depth of 24. But the paramorphism fusion allows to completely deforest the rest of the program, since it brings together the calls of *countNodes* and *genTree* which are replaced later by *countNodes_genTree*. This is a notable remark, since we are not aware of any implementation of fusion which could handle these cases.

The performance of the fused program is very similar to the one achieved for the example in section 8.1, when no intermediate data structures remain either.

8.5 Summary

In the following table we present the time comparisons in seconds. The time ratio is calculated as $100 * F/O$ where O is the running time of the original program and F is the running time of the fused program.

program	O	F	100*F/O
sumMapGenBTree	9.87	0.002	0.02%
normalizer	21.45	19.52	91%
list comprehension	28.11	13.55	48%
paramorphism	9,44	0.001	0.01%

When fusion eliminates all of the intermediate data structures the improvements are dramatical. This numbers should be considered with caution since our test programs do not perform any expensive computations like I/O. This makes the relative weight of the intermediate data structures significant to the performance.

In the following table we present the comparisons of total allocated bytes. The byte ratio is calculated as $100 * F/O$ where O is the count of bytes allocated by the original program and F is the count of bytes allocated by the fused program.

program	O	F	100*F/O
sumMapGenBTree	1,636,009,068	35,768	0.002%
normalizer	385,059,788	383,369,032	99.6%
list comprehension	7,510,826,324	4,726,050,264	62.9%
paramorphism	1,627,291,224	45,064	0.003%

Greater savings in allocated bytes seem to imply greater improvements in running times. In practice, not all the critical computations in a program may be possible to deforest, therefore, more modest results should be expected when working with real programs.

Chapter 9

Future work and conclusions

In this work we have contributed to the automation of program transformations based on fusion techniques. More specifically, we have explored the possibilities of acid rain style laws to remove intermediate data structures for various kinds of recursive programs.

We devote this chapter to state some possibilities and limitations of our implementation and our approach. In the last section, we summarize our contributions and give some final remarks.

9.1 Definitions returning multiple results

Inspired by the law for fusing definitions recursing over multiple arguments (Chapter 7), we immediately arrive at the dual case, which enables fusion of definitions producing multiple results. As for Lemma 7.2, we can state a more general law.

Lemma 9.1

$$\frac{\tau :: \forall a.\ (F\ a \to a) \to G\ (H\ a) \to H\ a}{H\ (\!|\phi|\!)_F \circ [\![\tau(in_F), \psi]\!]_G = [\![\tau(\phi), \psi]\!]_G}$$

This would enable fusion of compositions like $(map\ f \times id) \circ unzip$. In fact, the definition of *unzip*:

$$\begin{aligned}
&unzip :: [(a,b)] \to ([a],[b])\\
&unzip\ [\,] = ([\,],[\,])\\
&unzip\ ((a,b) : zs) = \mathbf{let}\ (xs, ys) = unzip\ zs\\
&\quad \mathbf{in}\ (a : xs, b : ys)
\end{aligned}$$

can be written as

$$\begin{aligned}
&unzip = (\!|\tau(in_F)|\!)_F\\
&\quad \mathbf{where}\ \tau(\alpha_1,\alpha_2)\ (1,()) = (\alpha_1,[\,])\\
&\qquad\quad \tau(\alpha_1,\alpha_2)\ (2,((a,b),(xs,ys))) = (\alpha_2\ a\ xs, b : ys)
\end{aligned}$$

which can now be fused with $map\ f \times id$, obtaining:

$$
\begin{aligned}
&mapunzip :: (a \to c) \to [(a,b)] \to ([c],[b])\\
&mapunzip\ f\ [\,] = ([\,],[\,])\\
&mapunzip\ f\ ((a,b) : zs) = \mathbf{let}\ (xs, ys) = mapunzip\ f\ zs\\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathbf{in}\ (f\ a : xs, b : ys)
\end{aligned}
$$

There is at least one implementation of shortcut fusion capable of handling these cases [Chi99]. The above law is also meaningful for other kinds of fusions [MP08, PFS09].

9.2 Monadic computations

So far we have dealt with fusion of pure functions. However, most real programs produce effects, which in the functional programming paradigm is usually associated with monadic functions. Fusion in the presence of effects has been analyzed by [GJ08, MP08, Par05, Par01]. HFusion could be extended to handle monadic functions.

9.3 Tupling

Tupling is a program transformation technique that optimizes programs using a different principle than fusion ([HITT96]). Tupling does not eliminate intermediate data structures, but optimizes functions which perform multiple traversals over a data structure. The result of tupling combines multiple traversals in a single function that performs the traversals rather simultaneously and returns a tuple.

Here is a typical example which computes the leaves that are farthest away from the root of a tree.

$$
\begin{aligned}
&deepest\ (Leaf\ a) \quad = [\,a\,]\\
&deepest\ (Join\ l\ r) = \mathbf{if}\ depthL > depthR\ \mathbf{then}\ deepest\ l\\
&\qquad\qquad\qquad\qquad\qquad\quad \mathbf{else\ if}\ depthL < depthR\ \mathbf{then}\ deepest\ r\\
&\qquad\qquad\qquad\qquad\qquad\qquad\quad \mathbf{else}\ deepest\ l +\!\!+ deepest\ r\\
&\quad \mathbf{where}\ depthL = depth\ l\\
&\qquad\qquad\ \ depthR = depth\ r\\
&depth\ (Leaf\ a) = 0\\
&depth\ (Join\ l\ r) = 1 + max\ (depth\ l)\ (depth\ r)
\end{aligned}
$$

This would be the optimized program obtained after applying tupling:

$$
\begin{aligned}
&deepest = fst \circ dd\\
&dd\ (Leaf\ a) \quad = ([\,a\,], 0)
\end{aligned}
$$

$$
\begin{aligned}
dd\ (Join\ l\ r) = \ & \textbf{if}\ dl > dr\ \textbf{then}\ (dpl, 1 + dl) \\
& \quad \textbf{else if}\ dl < dr\ \textbf{then}\ (dpr, 1 + dr) \\
& \quad\quad \textbf{else}\ (dpl +\!\!+ dpr, 1 + dl) \\
\textbf{where}\ & (dpl, dl) = dd\ l \\
& (dpr, dr) = dd\ r
\end{aligned}
$$

In general, tupling may cause an improvement in the efficiency of a program. In the example, the original program is quadratic in the amount of nodes of the tree, while the transformed version is almost linear (appending the results may still make the program quadratic in the amount of leaves).

Most notably, the resulting function is inlined from a fold, which would leave open the opportunity for fusion in a later pass.

Integrating tupling in a transformation system like HFusion would require analyzing how it fits, among others, with mutual recursion, recursion over multiple arguments, primitive recursion and accumulators.

9.4 Composition discovering

To date, HFusion is not capable of searching compositions automatically. Finding all the compositions in a program may require special inlining rules to make the compositions evident while avoiding to duplicate computation.

Consider as an example the following expression:

$$
\begin{aligned}
&\textbf{let}\ l' = map\ f\ l \\
&\textbf{in}\ \ zip\ l'\ l'
\end{aligned}
$$

If we inline l' to make the compositions evident computing the expression will apply function f twice to each element in list l, even after performing fusion.

$$zip\ (map\ f\ l)\ (map\ f\ l)$$

Therefore, care must be taken to inline only if applying fusion would still improve the program. The result of fusion is $zmm\ f\ f\ l\ l$ where:

$$
\begin{aligned}
zmm\ f\ g\ (x : xs)\ (y : ys) &= (f\ x, g\ y) : zmm\ f\ g\ xs\ ys \\
zmm\ f\ g\ \_\quad\ \_\quad\ &= [\,]
\end{aligned}
$$

If function f is expensive enough to compute, just applying fusion may not be desirable. Note, however, that $zmm\ f\ f\ l\ l$ is equivalent to $[(fx, fx) \mid x \leftarrow l, fx = f\ x]$, where the later expression does not recompute f. It might be possible to analyze the output of fusion with other techniques to improve the result.

Replacing a composition by a fusion is also convoluted if one wants to achieve source to source transformations as the ones implemented in HFusion.

The system can be used as a kind of refactorer which takes two definitions and fuses them. However, when the functions being fused use local definitions that are not globally available, it is not obvious where to place the definition produced by the fusion. Sometimes there is not a place to put the fusion result, as all the needed local definitions are never simultaneously in scope.

Another subtle problem of fusion is how to handle instances of type classes. If a class method appears in a composition, it may not be possible to determine immediately which instance should be considered for the fusion.

Despite these technicalities, it is planed to incorporate this feature to HFusion in the near future.

9.5 Relationship with shortcut fusion

We will start by giving an overview of what shortcut fusion is about [TM95, GLPJ93, Gil96]. In a sense, shortcut fusion can be thought of as a generalization of acid rain laws.

Let's consider acid rain for the fold-hylo case:

$$\tau :: \forall b.(F\ b \to b) \to G\ b \to b \quad \Rightarrow \quad (\!|\phi|\!)_F \circ [\![\tau(in_F), \psi]\!]_G = [\![\tau(\phi), \psi]\!]_G$$

Shortcut fusion generalizes this with its popular build-cata rule:

$$f :: \forall b.(F\ b \to b) \to a \to b \quad \Rightarrow \quad (\!|\phi|\!)_F \circ f\ in_F = f\ \phi$$

To get the fold-hylo law from this one, function f must be instantiated as $\lambda\alpha \to [\![\tau(\alpha), \psi]\!]_G$.

From our perspective, the most important aspect of shortcut fusion is that it does not require any special knowledge about the recursive scheme used by function f. All it requires from f is that it meets the proper polymorphic type, much in the same way as we require the polymorphic type for τ. The generalization implies that f can abstract away constructors in the whole body of the definition, while τ only allows to abstract the constructors occurring in the algebra of a hylomorphism.

We can establish the dual generalization for the hylo-unfold case:

$$\sigma :: \forall b.(b \to F\ b) \to b \to G\ b \quad \Rightarrow \quad [\![\phi, \sigma(out_F)]\!]_G \circ (\!|\psi|\!)_F = [\![\phi, \sigma(\psi)]\!]_G$$

which in the shortcut fusion approach can be expressed in terms of the destroy-unfold rule:

$$f :: \forall b.(b \to F\ b) \to b \to a \quad \Rightarrow \quad f\ out_F \circ (\!|\psi|\!)_F = f\ \psi$$

by taking $f = \setminus\ \beta \to [\![\phi, \sigma(\beta)]\!]_G$. This law is equally practical for similar reasons to those used for the build-cata rule.

A shortcut fusion law for multiple arguments Now we could take this generalization a step forward. Let us recall our law for fusing definitions which recurse over multiple arguments.

$$\sigma :: \forall b.(b \to F\ b) \to H\ b \to G\ (H\ b) \quad \Rightarrow$$
$$[\![\phi, \sigma(out_F)]\!]_G \circ H\ [\![\psi]\!]_F = [\![\phi, \sigma(\psi)]\!]_G$$

What happens when we rewrite this rule in shortcut fusion style?

$$f :: \forall b.(b \to F\ b) \to H\ b \to a \quad \Rightarrow \quad f\ out_F \circ H\ [\![\psi]\!]_F = f\ \psi$$

We get a formulation of shortcut fusion for multiple arguments! This law appears in [GJ08] though it is proposed for fusion of monadic programs, and the dual appears in [MP08, PFS09].

In [Sve02] it is shown how shortcut fusion can deal with recursion over multiple arguments by means of the destroy-unfold rule. It remains to be analyzed the strengths and weaknesses of both approaches with respect to each other.

Recognizing fold and unfolds As shortcut fusion involves recognizing folds and unfolds to apply its rules, all the machinery we have presented to restructure and rewrite hylomorphisms is potentially useful. For this sake, we remark the following contents:

- Sections 3.2.1 and 3.2.2 contain the basic restructures that allow to put definitions in fold or unfold form.
- Chapter 5 about partial deforestation shows how to recognize folds and unfolds when the definitions traverse only part of the intermediate data structures, or when they traverse different parts in different cases.
- Sections 7.2 and 7.5 contain techniques that rewrite definitions so that functions that do not seem to be unfolds can be seen as such in some situations.

9.6 Limitations of hylo fusion

When we started the implementation of HFusion, one of our strongest ambitions was surpassing the power of the shortcut fusion implementations we knew about.

As we progressed on understanding the problems fusion may face, it turned out that our approach has some shortcomings that shortcut fusion doesn't. In what follows, we present some composition examples we believe unlikely to be worked out without resorting to other techniques than fusion through acid rain.

9.6.1 Definitions with nested calls

Let's consider function *flatten* that returns a list containing the nodes of a tree in pre-order.

$$
\begin{aligned}
&\mathit{flatten} :: \mathit{Tree}\ a \to [a] \to [a] \\
&\mathit{flatten}\ \mathit{Empty}\ \mathit{acc} = \mathit{acc} \\
&\mathit{flatten}\ (\mathit{Node}\ a\ l\ r)\ \mathit{acc} = a : \mathit{flatten}\ l\ (\mathit{flatten}\ r\ \mathit{acc})
\end{aligned}
$$

This function cannot be expressed as a hylomorphism returning a value of type $[a]$ because it contains nested recursive calls. However, fusion of $\mathit{map}\ f \circ \mathit{flatten}\ t$ could be written as

$$
\begin{aligned}
&\mathit{mapflatten}\ f\ t\ \mathit{xs} = \mathit{mflatten}\ f\ t\ (\mathit{map}\ f\ \mathit{xs}) \\
&\mathit{mflatten} :: (a \to b) \to \mathit{Tree}\ a \to [b] \to [b] \\
&\mathit{mflatten}\ f\ \mathit{Empty}\ \mathit{acc} = \mathit{acc} \\
&\mathit{mflatten}\ f\ (\mathit{Node}\ a\ l\ r)\ \mathit{acc} = f\ a : \mathit{mflatten}\ f\ l\ (\mathit{mflatten}\ f\ r\ \mathit{acc})
\end{aligned}
$$

We have no law to achieve such a result. However, shortcut fusion has a better chance to do it since it does not need to mess with the recursion scheme [MP09, Mar09].

9.6.2 Definitions with accumulators

Consider now functions *reverse* and *sum*.

$$
\begin{aligned}
&\mathit{reverse} :: [a] \to [a] \to [a] \\
&\mathit{reverse}\ \mathit{acc}\ [\,] \quad\ = \mathit{acc} \\
&\mathit{reverse}\ \mathit{acc}\ (x : \mathit{xs}) = \mathit{reverse}\ (x : \mathit{acc})\ \mathit{xs} \\
&\mathit{sum} :: [\mathit{Int}] \to \mathit{Int} \\
&\mathit{sum}\ [\,] \quad\ = 0 \\
&\mathit{sum}\ (x : \mathit{xs}) = x + \mathit{sum}\ \mathit{xs}
\end{aligned}
$$

We have shown in Chapter 7 that we can fuse compositions like $\mathit{reverse}\ \mathit{xs} \circ \mathit{map}\ f$, because *reverse* is a function that has an accumulator and therefore can be expressed as a hylomorphism that recurses over multiple arguments, leading to the result:

$$
\begin{aligned}
&\mathit{reversem} :: (a \to b) \to [b] \to [b] \to [b] \\
&\mathit{reversem}\ f\ \mathit{acc}\ [\,] = \mathit{acc} \\
&\mathit{reversem}\ f\ \mathit{acc}\ (x : \mathit{xs}) = \mathit{reversem}\ f\ (f\ x : \mathit{acc})\ \mathit{xs}
\end{aligned}
$$

However, having a composition $\mathit{sum} \circ \mathit{reverse}\ \mathit{xs}$ we achieve the following result that does not deforest anything.

$$
\begin{aligned}
&\mathit{sreverse} :: [\mathit{Int}] \to [\mathit{Int}] \to \mathit{Int}\\
&\mathit{sreverse}\ f\ \mathit{acc}\ [\,] = \mathit{sum}\ \mathit{acc}\\
&\mathit{sreverse}\ f\ \mathit{acc}\ (x : xs) = \mathit{sreverse}\ (x : \mathit{acc})\ xs
\end{aligned}
$$

The problem here, is that the constructor application that should be eliminated is in the accumulator argument of the recursive call, whereas with acid rain we only can abstract constructors in the algebra. Recent work on shortcut fusion has managed to handle these cases [MP09, Mar09], where the result of the fusion is:

$$
\begin{aligned}
&\mathit{sreverse}\ \mathit{acc}\ xs = \mathit{srev}\ (\mathit{sum}\ \mathit{acc})\ xs\\
&\mathit{srev} :: \mathit{Int} \to [\mathit{Int}] \to \mathit{Int}\\
&\mathit{srev}\ \mathit{acc}\ [\,] = \mathit{acc}\\
&\mathit{srev}\ \mathit{acc}\ (x : xs) = \mathit{srev}\ (x + \mathit{acc})\ xs
\end{aligned}
$$

There are other approaches to fusion that handle this cases too [Voi04, Nis04].

9.6.3 Fusing inside recursive definitions

There are cases where shortcut fusion can be used to optimize recursive definitions, and even improve the order of the algorithm they perform. For example, Chitil [Chi99] starts with the following quadratic definition of *reverse*:

$$
\begin{aligned}
&\mathit{reverse} :: [a] \to [a]\\
&\mathit{reverse}\ [\,] = [\,]\\
&\mathit{reverse}\ (x : xs) = \mathit{reverse}\ xs +\!\!+ [x]
\end{aligned}
$$

and by fusion transforms $(+\!\!+[x]) \circ \mathit{reverse}$ into the definition of fast *reverse*, which is the linear, accumulative definition presented in Section 9.6.2.

We will not go into the details of how fusion proceeds in this case. It suffices to say that it is very unlikely that we will ever come up with a similar trick for hylo fusion, since acid rain does not change the amount of recursive arguments of a definition.

Another example where this ability may be useful appeared in Section 8.2. Perhaps shortcut fusion could fuse *union* (*fv bs e0*) (*fv bs e1*) in the definition of *fv*, and later fuse *zip_freshvars* (*fv* [ ] *e1*) *gen* in the definition of *eval′*.

9.6.4 Handling compositions of reverse

Consider the composition *reverse* [ ] ∘ *reverse* [ ] where we are using the fast *reverse* accumulative definition. Despite we know that the composition is equivalent to the *id* function, it cannot be deforested using acid rain. And it is dubious, also, that shortcut fusion would improve the composition since it involves treatment of *reverse* as a higher order fold.

This inability blocks fusion in a program like the following, which takes the last n elements of a given list, preserving the order.

$$f\ n = reverse\ [\,] \circ take\ n \circ reverse\ [\,]$$

HFusion could fuse the leftmost composition, however fusing later the other composition is not less difficult than fusing $reverse\ [\,] \circ reverse\ [\,]$.

As another example, consider the following program, which takes a number and tells if it is a palindrome.

$$\begin{aligned}
&palindrome :: Int \to Bool\\
&palindrome\ n = n == revnum\ n\\
&\quad \textbf{where}\ revnum = eval\ 0 \circ reverse\ [\,] \circ digits\ [\,]\\
&digits :: [Int] \to Int \to [Int]\\
&digits\ acc\ 0 = acc\\
&digits\ acc\ n = digits\ (r : acc)\ d\\
&\quad \textbf{where}\ (d, r) = divMod\ n\ 10\\
&eval :: Int \to [Int] \to Int\\
&eval\ acc\ [\,] \quad\ = acc\\
&eval\ acc\ (x : xs) = eval\ (10 * acc + x)\ xs
\end{aligned}$$

Suppose that we want to fuse $reverse\ [\,] \circ digits\ [\,]$. It turns out that this composition cannot be deforested using acid rain. To explain why, note that *digits* can be decomposed into more simple functions as:

$$\begin{aligned}
&digits :: Int \to [Int]\\
&digits = reverse\ [\,] \circ revdig\\
&revdig :: Int \to [Int]\\
&revdig\ 0 = [\,]\\
&revdig\ n = r : revdig\ d\\
&\quad \textbf{where}\ (d, r) = divMod\ n\ 10
\end{aligned}$$

So the composition $reverse\ [\,] \circ digits\ [\,]$ can be rewritten to $reverse\ [\,] \circ reverse\ [\,] \circ revdig$. We could fuse $reverse\ [\,] \circ revdig$ into $digits\ [\,]$ again using HFusion. However, there does not seem to be a way to fuse the leftmost composition afterwards.

9.7 Conclusions

In this work we have explored the possibilities of hylo fusion for various kinds of programs by extending the acid rain laws. In doing so, we have provided algorithms and remarks from our experience implementing those extensions.

Our main contributions start in Chapter 3, where we present the restructuring and transformer derivation algorithms for normal hylomorphisms that HFusion uses. Many of these algorithms are not new [OHIT97, Sch00], but

they are expressed in a succinct way. For instance, the duality between deriving τ and σ was not easy to appreciate in previous work. Another advantage of our presentation, is that it generalizes easily to work over mutual hylos and hylos which recurse over multiple arguments. Certainly, one of the keys to the effectiveness of our algorithm descriptions was using view patterns to abstract constructors in patterns as shown in Section 3.2.4.

We have explored fusion of primitive recursive functions or paramorphisms. Most notably, the generalizations presented in Chapter 4 were the result of explaining what had been implemented in HFusion as the result of tinkering with the original algorithms and internals. Though primitive recursive functions can be written in terms of a fold, in practice this is never done, and therefore, compositions involving primitive recursive functions may not be possible to deforest without the extension.

After considering primitive recursion, we still had not covered everything HFusion was doing. Compositions of paramorphisms sometimes involve an intermediate data structure that is not completely produced or consumed by the functions being composed. It happens that other compositions not involving paramorphisms present this singularity as well, therefore calling for more techniques applicable when only a part of the intermediate data structures can be deforested. We have presented these techniques in Chapter 5.

It is debatable whether or not partial deforestation deforests enough to be worth the trouble of performing it. The key argument to consider it, is that it brings together function calls for fusion that may not be otherwise possible to join. See Section 8.4 for an example of this point.

We have implemented an extension for fusing mutually recursive definitions. Mutual hylos are worth themselves for the expressiveness they bring to enable deforestation of a broader class of functions. But in addition, they allow the treatment of hylomorphisms with regular functors (Section 6.4). Another contribution we made is the algorithms for the theoretical framework presented in [IHT98].

The last of our extensions is fusion for functions which recurse over multiple arguments (Chapter 7). This extension enable us to fuse functions like *zip*, *zipWith* and equality for recursive data types, as well as functions with accumulators (e.g. *foldl*, *take*, *drop*). This power may be similar to what Svenningsson [Sve02] claimed about through the use of the destroy-unfold rule of shortcut fusion.

Once mutual recursion and recursion over multiple arguments coexist in a same implementation, some restructures of program definitions become available to achieve fusion where the isolated approaches would not succeed (Section 7.3). In the light of these restructures, we have formulated a normal form for function definitions that could improve the result and likelihood of fusion.

We provide, also, an experimental implementation of most of the tech-

niques described in this work, which made it possible to test them on concrete examples. The implementation is available at

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

The development of our implementation has been a fundamental tool to identify and figure out the multiple extensions and manipulations we have presented. We wouldn't have earned so much intuition about automating and explaining acid rain laws, should we not had been confronted with the problem of coding the theoretical algorithms and analyzing every restriction needed on the input to guarantee a proper behavior.

Bibliography

[AJ94] 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.

[BdM97] R.S. Bird and O. de Moor. *Algebra of Programming.* Prentice Hall, UK, 1997.

[BFaF99] Gilles Barthe, Maria JoÃ£o Frade, and Maria Jo ao Frade. Constructor subtyping. In *Proceedings of ESOP'99, LNCS 1576*, pages 109–127. Springer-Verlag, 1999.

[Bir98] R. Bird. *Introduction to Functional Programming using Haskell (*2nd edition*).* Prentice-Hall, UK, 1998.

[Chi92] W.N. Chin. Safe fusion of functional expressions. In *In Proc. Conference on Lisp and Functional Programming, San Francisco, California*, pages 11–20, June 1992.

[Chi99] Olaf Chitil. Type-inference based short cut deforestation (nearly) without inlining. In *In IFL'99, Lochem, The Netherlands, Proceedings*, page pages. Springer-Verlag, 1999.

[CLS07] Duncan Coutts, Roman Leshchinskiy, and Don Stewart. Stream fusion: from lists to streams to nothing at all. *Proceedings of the ACM SIGPLAN International Conference on Functional Programming, ICFP 2007*, 42(9):315–326, 2007.

[CUV06] V. Capretta, T. Uustalu, and V. Vene. Recursive coalgebras from comonads. *Information and Computation*, 204(4):437–468, 2006.

[dMS99] O. de Moor and G. Sittampalam. Generic program transformation. In *Advanced Functional Programming*, Lecture Notes in Computer Science vol. 1608. Springer-Verlag, 1999.

[Dom04] Facundo Domínguez. Implementación de un sistema de fusión, 2004. Final year project. Facultad de Ingeniería, Universidad de la República, Uruguay.

[DP06a] Facundo Domínguez and Alberto Pardo. Automatización de leyes de fusión de programas. In *XXXII Conferencia Latinoamericana de Informática (CLEI 2006)*, August 2006.

[DP06b] Facundo Domínguez and Alberto Pardo. Program fusion with paramorphisms. In *Mathematically Structured Functional Programming (MSFP'06), Electronic Workshops in Computing. British Computer Society*, 2006.

[FM91] M. Fokkinga and E. Meijer. Program Calculation Properties of Continuous Algebras. Technical Report CS-R9104, CWI, Amsterdam, January 1991.

[Fok92] M.M. Fokkinga. *Law and Order in Algorithmics*. PhD thesis, Universiteit Twente, The Netherlands, 1992.

[Gib06] J. Gibbons. Fission for Program Comprehension. In *Intl. Conf. on Mathematics of Program Construction (MPC 2006),* LNCS ??? Springer-Verlag, 2006.

[Gil96] Andrew John Gill. *Cheap Deforestation for Non-strict Functional Languages*. PhD thesis, Glasgow University, 1996.

[GJ98] J. Gibbons and G. Jones. The Under-Appreciated Unfold. In *Proc. 3rd. ACM SIGPLAN International Conference on Functional Programming*. ACM, September 1998.

[GJ08] N. Ghani and P. Johann. Short Cut Fusion of Recursive Programs with Computational Effects. In *Symposium on Trends in Functional Programming (TFP 2008)*, 2008.

[GLPJ93] Andrew Gill, John Launchbury, and Simon L. Peyton Jones. A short cut to deforestation. In *Conference on Functional Programming Languages and Computer Architecture*, pages 223–232, June 1993.

[HIT96a] Z. Hu, H. Iwasaki, and M. Takeichi. Calculating Accumulations. Technical Report METR 96-03, Faculty of Engineering, University of Tokyo, March 1996.

[HIT96b] Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. Deriving structural hylomorphisms from recursive definitions. In *Proceedings 1st ACM SIGPLAN Int. Conf. on Functional Programming,*

ICFP'96, Philadelphia, PA, USA, 24–26 May 1996, volume 31(6), pages 73–82. ACM Press, New York, 1996.

[HIT97] Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. An extension of the acid rain theorem. In T. Ida, A. Ohori, and M. Takeichi, editors, *Proceedings 2nd Fuji Int. Workshop on Functional and Logic Programming, Shonan Village Center, Japan, 1–4 Nov. 1996*, pages 91–105. World Scientific, Singapore, 1997.

[HITT96] Zhenjiang Hu, Hideya Iwasaki, Masato Takeichi, and Akihiko Takano. Tupling calculation eliminates multiple data traversals. In *Proceedings 2nd ACM SIGPLAN Int. Conf. on Functional Programming, ICFP'97, Amsterdam, The Netherlands, 9–11 June 1997*, volume 32(8), pages 164–175. ACM Press, New York, 1996.

[IHT98] Hideya Iwasaki, Zhenjiang Hu, and Masato Takeichi. Towards manipulation of mutually recursive functions. In *Fuji International Symposium on Functional and Logic Programming*, pages 61–79, 1998.

[JL98] Patricia Johann and John Launchbury. Warm fusion for the masses: Detailing virtual data structure elimination in fully recursive languages. In *SDRR Project Phase II, Final Report*, Computer Science and Engineering Department,Oregon Gradute Institute, USA, 1998.

[Jon03] Simon Peyton Jones. *Haskell 98 Language and Libraries.* Cambridge University Press, 2003.

[JV04] Patricia Johann and Janis Voigtländer. Free theorems in the presence of seq. In *POPL '04: Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 99–110, New York, NY, USA, 2004. ACM.

[Mal90] G. Malcolm. *Algebraic Data Types and Program Transformation.* PhD thesis, Dept. of Computer Science, University of Groningen, The Netherlands, 1990.

[Mar09] Mónica Martínez. Fusión en presencia de acumuladores. Master's thesis, Pedeciba Informática, Universidad de la República, Uruguay, 2009. Forthcoming.

[Mee92] Lambert G. L. T. Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–424, 1992.

[MFP91] E. Meijer, M. Fokkinga, and R. Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In *Proceedings of Functional Programming Languages and Computer Architecture'91,* LNCS 523. Springer-Verlag, August 1991.

[MP08] C. Manzino and A. Pardo. Short Cut Fusion of Monadic Programs. In *Brazilian Symposium on Programming Languages (SBLP 2008)*, 2008.

[MP09] Mónica Martínez and Alberto Pardo. A Short Cut Fusion Approach to Accumulations. 2009. Submitted.

[Nis04] Susumu Nishimura. Fusion with stacks and accumulating parameters. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Partial Evaluation and Semantics–based Program Manipulation, 2004, Verona, Italy, August 24–25*, pages 101–112, 2004.

[OHIT97] Yoshiyuki Onoue, Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. A calculational fusion system HYLO. In *Proceedings of the IFIP TC 2 WG 2.1 international workshop on Algorithmic languages and calculi*, pages 76–106, London, UK, UK, 1997. Chapman & Hall, Ltd.

[Par01] Alberto Pardo. Fusion of recursive programs with computational effects. *Theoretical Computer Science*, 260:207, 2001.

[Par05] Alberto Pardo. Combining datatypes and effects. In *Advanced Functional Programming*, volume 3622 (2005) of *Lecture Notes in Computer Science*, pages 171–209. Springer Berlin / Heidelberg, 2005.

[PFS09] Alberto Pardo, João Paulo Fernandes, and João Saraiva. Shortcut fusion rules for the derivation of circular and higher-order monadic programs. In *PEPM '09: Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation*, pages 81–90, New York, NY, USA, January 2009. ACM.

[Sch00] J. Schwartz. Eliminating intermediate lists in pH. Master's thesis, Massachusetts Institute of Technology, USA, May 2000.

[SF93] Tim Sheard and Leonidas Fegaras. A fold for all seasons. In *Proceedings 6th ACM SIGPLAN/SIGARCH Int. Conf. on Functional Programming Languages and Computer Architecture, FPCA'93, Copenhagen, Denmark, 9–11 June 1993*, pages 233–242. ACM Press, New York, 1993.

[Sve02] Josef Svenningsson. Shortcut fusion for accumulating parameters & zip-like functions. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN international conference on Functional programming*, pages 124–132, New York, NY, USA, October 2002. ACM.

[TM95] Akihiko Takano and Erik Meijer. Shortcut deforestation in calculational form. In *Conf. Record 7th ACM SIGPLAN/SIGARCH Int. Conf. on Functional Programming Languages and Computer Architecture, FPCA'95, La Jolla, San Diego, CA, USA, 25–28 June 1995*, pages 306–313. ACM Press, New York, 1995.

[Voi04] Janis Voigtländer. Using circular programs to deforest in accumulating parameters. *Higher–Order and Symbolic Computation*, 17:129–163, 2004. Previous version appeared in *ASIA–PEPM 2002*, Proceedings, pages 126–37, ACM Press, 2002.

[Wad87] P. Wadler. Views: a way for pattern matching to cohabit with data abstraction. In *POPL '87: Proceedings of the 14th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 307–313, New York, NY, USA, 1987. ACM.

[Wad88] P. Wadler. Deforestation: Transforming programs to eliminate trees. In *ESOP '88. European Symposium on Programming, Nancy, France, 1988 (Lecture Notes in Computer Science, vol. 300)*, pages 344–358. Berlin: Springer-Verlag, 1988.

[Wad89] Philip Wadler. Theorems for free! In *Proceedings 4th Int. Conf. on Funct. Prog. Languages and Computer Arch., FPCA'89, London, UK, 11–13 Sept 1989*, pages 347–359. ACM Press, New York, 1989.

Appendix A

Counterexamples

We have made some modifications to the derivation algorithm for τ proposed in [OHIT97], since we found it didn't work in some cases. We reworked a derivation algorithm for σ because the original algorithm was incorrect in its handling of the coalgebra pattern matching. We also restricted a little the form of the terms which are accepted as input.

This appendix is devoted to present the counterexamples that point at the pitfalls of the original proposal for the derivation algorithm for τ, together with a counterexample showing why we have restricted the input to the derivation algorithm for σ.

A.1 Derivation of τ

Here's the normal form proposed for the terms inside algebra components ϕ_i in the original algorithm:

1. the term is an input variable of ϕ_i;
2. it is a constructor application of terms in normal form;
3. it is a hylomorphism application $[\![\phi'_1 \triangledown \cdots \triangledown \phi'_n, out_F]\!]_F\ v'$ where each ϕ'_i appears in the same normal form than ϕ_i, and v is a recursive variable;
4. it is in the form $f\ t_1\ \ldots\ t_n$ where f is a global function and the terms $t_1, \ldots, t_n$ do not reference any recursive variable.

We will show that allowing terms in the form of the second or third clauses may lead to erroneous derivations.

Counterexample A.1 (Second clause) Let's consider the following definitions

$$mapl :: (a \to a) \to Tree\ a \to Tree\ a$$
$$mapl\ f\ Empty = Empty$$

$$
\begin{array}{l}
mapl\ f\ (Node\ a\ l\ r)) = Node\ (f\ a)\ (mapl\ f\ l)\ r \\
prunel :: (a \to Bool) \to Tree\ a \to Tree\ a \\
prunel\ p\ Empty = Empty \\
prunel\ p\ (Node\ a\ l\ r) = \\
\quad \textbf{case}\ p\ a\ \textbf{of} \\
\quad\quad True \to prunel\ p\ r \\
\quad\quad False \to Node\ a\ (prunel\ p\ l)\ (prunel\ p\ r)
\end{array}
$$

If we derive the corresponding hylomorphisms we get

$$
\begin{array}{l}
mapl\ f = (\!|Empty \triangledown (\lambda(i, t1, t2) \to Node\ (f\ i)\ l\ r)|\!)_F \\
\quad \textbf{where}\ F = \overline{\mathbf{1}} + \overline{Int} \times I \times \overline{Tree\ a} \\
prunel\ p = (\!|\phi|\!)_H \\
\quad \textbf{where}\ H = \overline{\mathbf{1}} + \overline{a} \times I \times I \\
\qquad\quad \phi\ (1, ()) = Empty \\
\qquad\quad \phi\ (2, (a, v_1, v_2)) = \textbf{case}\ p\ a\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad True \to v_2 \\
\qquad\qquad\qquad\qquad\qquad False \to Node\ a\ v_1\ v_2
\end{array}
$$

If we want to fuse $mapl\ f \circ prunel\ p$ we need to restructure *prunel*.

$$
\begin{array}{l}
prunel\ p = [\![\phi_1 \triangledown (id \triangledown \phi_3), \eta \circ \psi]\!]_G \\
\quad \textbf{where}\ G = \overline{\mathbf{1}} + (I + \overline{a} \times I \times I) \\
\qquad\quad \psi\ Empty = (1, ()) \\
\qquad\quad \psi\ (Node\ a\ l\ r) = (2, (a, l, r)) \\
\qquad\quad \eta :: H \Rightarrow G \\
\qquad\quad \eta = id + \eta_2 \\
\qquad\quad \eta_2\ (a, v_1, v_2) = \textbf{case}\ p\ a\ \textbf{of} \\
\qquad\qquad\qquad\qquad\qquad True \to (2, (1, v_2)) \\
\qquad\qquad\qquad\qquad\qquad False \to (2, (2, (a, v_1, v_2))) \\
\qquad\quad \phi_1\ () = Empty \\
\qquad\quad \phi_3\ (a, v_1, v_2) = Node\ a\ v_1\ v_2
\end{array}
$$

Since the algebra of *prunel* after restructuring is not in_F, we have to derive τ by using the functor F of *mapl*. Note that in this case the terms in the algebra of *prunel* satisfy the normal form of Onoue et al. [OHIT97] but not ours. In fact, none of the clauses of our normal form are satisfied, specifically the second clause is not satisfied by ϕ_3. The argument v_2 is non-recursive according to F but it is a recursive variable according to G.

If we continue to derive τ according to the original algorithm we get

$$\tau(\alpha_1 \triangledown \alpha_2) = \alpha_1 \triangledown id \triangledown (\lambda(a, v_1, v_2) \to \alpha_2\ (a, v_1, v_2))$$

Note that the expected type of τ is $(F\ b \to b) \to G\ b \to b$. However, the real type of the derived τ is $(H\ b \to b) \to G\ b \to b$

At this point there is no fusion rule we could apply, but it is instructive inspecting the outcome of proceeding with the fusion despite the typing error.

$$mpl\ f\ p = [\![\tau(Empty \triangledown (\lambda(i, v_1, v_2) \to Node\ (f\ i)\ v_1\ v_2)), \eta \circ \psi]\!]_G$$

Inlining the above we get:

$$\begin{array}{l} mpl :: (a \to a) \to (a \to Bool) \to Tree\ a \to Tree\ a \\ mpl\ f\ p\ Empty \qquad = Empty \\ mpl\ f\ p\ (Node\ a\ l\ r) = \\ \quad \textbf{case}\ p\ a\ \textbf{of} \\ \quad\quad True \to mpl\ f\ p\ t2 \\ \quad\quad False \to Node\ (f\ a)\ (mpl\ f\ p\ l)\ (mpl\ f\ p\ r) \end{array}$$

Then, we can verify that

$$\begin{array}{l} mpl\ (+1)\ (==8)\ (Node\ 1\ (Node\ 1\ Empty\ Empty) \\ \qquad\qquad\qquad\qquad\qquad\qquad (Node\ 1\ Empty\ Empty)) = \\ \quad (Node\ 2\ (Node\ 2\ Empty\ Empty)\ (Node\ 2\ Empty\ Empty)) \end{array}$$

while the original composition returns

$$\begin{array}{l} mapl\ (+1) \circ prunel\ (==8)\ (Node\ 1\ (Node\ 1\ Empty\ Empty) \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (Node\ 1\ Empty\ Empty)) = \\ \quad (Node\ 2\ (Node\ 2\ Empty\ Empty)\ (Node\ 1\ Empty\ Empty)) \end{array}$$

Summarizing, the derivation algorithm for τ can fail if applied over algebras whose terms are admitted by the second clause of the normal form proposed by Onoue et al. [OHIT97]. □

Counterexample A.2 (Third clause) Consider the following definitions

$$\begin{array}{l} concatr :: [[a]] \to [a] \\ concatr\ [\,] = [\,] \\ concatr\ (a : as) = append\ a\ (concatr\ as) \\ append :: [a] \to [a] \to [a] \\ append\ l\ [\,] = l \\ append\ l\ (a : as) = a : append\ l\ as \end{array}$$

The function *append* concatenates its arguments. The function *concatr* does the same as the composition $concat \circ reverse$, i.e. given a list of lists, first it is reversed and then the inner lists are concatenated.

We write the above definitions as hylomorphisms.

$$\begin{array}{l} concatr = [\![[\,]\triangledown(\lambda(a, v) \to append\ a\ v), out_{F'}]\!]_{F'} \\ \quad \textbf{where}\ F' = \overline{\mathbf{1}} + \overline{[a]} \times I \end{array}$$

$$
\begin{aligned}
&\mathit{append}\ l = (\!|(\lambda() \to l) \triangledown (\lambda(a, v) \to a : v)|\!)_F \\
&\quad \textbf{where}\ F = \overline{\mathbf{1}} + \overline{a} \times I
\end{aligned}
$$

Let's say we want to fuse $\mathit{map}\ f \circ \mathit{concatr}$. We will have to derive τ from the algebra of *concatr*. Our normal form does not accept this algebra, but Onoue normal form does. According to the original algorithm the derived τ is:

$$
\begin{aligned}
&\tau :: (F\ b \to b) \to F'\ b \to b \\
&\tau(\alpha_1 \triangledown \alpha_2) = \alpha_1 \triangledown \phi_2 \\
&\quad \textbf{where}\ \phi_2\ (a, v) = (\!|\tau'(\alpha_1 \triangledown \alpha_2)|\!)_F\ v \\
&\qquad\qquad \textbf{where}\ \tau'(\alpha_1 \triangledown \alpha_2) = (\lambda() \to a) \triangledown \alpha_2
\end{aligned}
$$

Note how this τ is ill-typed again. The type of variable v to which $(\!|\tau'(\alpha_1 \triangledown \alpha_2)|\!)_F$ is applied should be the carrier type of out_F (i.e. $[a]$). However, the type of τ indicates that it should be of a polymorphic type b. Ignoring this type error is not possible since the resulting fusion would not be well-typed.

Summarizing, the derivation algorithm for τ may fail if applied over algebras whose terms satisfy the third clause of the normal form of Onoue et al. [OHIT97]. □

A.2 Derivation of σ

In the restrictions for the input of the derivation algorithm for σ, we require that the recursive terms of the coalgebra must be variables. Here's a counterexample justifying this restriction.

Counterexample A.3 (Recursive terms must be variables)
Consider the following definitions.

$$
\begin{aligned}
&\mathit{mapT} :: (a \to b) \to \mathit{Tree}\ a \to \mathit{Tree}\ b \\
&\mathit{mapT}\ f\ \mathit{Empty} = \mathit{Empty} \\
&\mathit{mapT}\ f\ (\mathit{Node}\ a\ l\ r) = \mathit{Node}\ (f\ a)\ (\mathit{mapT}\ f\ l)\ (\mathit{mapT}\ f\ r) \\
&\mathit{addDepth} :: \mathit{Tree}\ \mathit{Int} \to \mathit{Tree}\ \mathit{Int} \\
&\mathit{addDepth}\ \mathit{Empty} = \mathit{Empty} \\
&\mathit{addDepth}\ (\mathit{Node}\ a\ l\ r) = \mathit{Node}\ a\ (\mathit{addDepth}\ (\mathit{mapT}\ (+1)\ l)) \\
&\qquad\qquad\qquad\qquad\qquad\qquad (\mathit{addDepth}\ (\mathit{mapT}\ (+1)\ r))
\end{aligned}
$$

Function *addDepth* takes a value of type *Tree Int*, and sums to each tree node its depth. We write the above definitions as hylomorphisms.

$$
\begin{aligned}
&\mathit{mapT}\ f = (\!|\mathit{Empty} \triangledown (\lambda(i, l, r) \to \mathit{Node}\ (f\ i)\ l\ r)|\!)_F \\
&\quad \textbf{where}\ F = \overline{\mathbf{1}} + \overline{\mathit{Int}} \times I \times I
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{addDepth} = [\![\psi]\!]_G \\
&\quad \textbf{where } G = \overline{\mathbf{1}} + \overline{\mathit{Int}} \times I \times I \\
&\qquad\quad \psi :: \mathit{Tree}\ \mathit{Int} \to G\ (\mathit{Tree}\ \mathit{Int}) \\
&\qquad\quad \psi\ t = \textbf{case } t \textbf{ of} \\
&\qquad\qquad\qquad \mathit{Empty} \to (1, ()) \\
&\qquad\qquad\qquad \mathit{Node}\ (a, l, r) \to (2, (a, \mathit{mapT}\ (+1)\ l, \mathit{mapT}\ (+1)\ r))
\end{aligned}
$$

Suppose we want to fuse $\mathit{addDepth} \circ \mathit{mapT}\ f$. Given that *addDepth* is not a fold, and can not be restructured to one, we need to derive σ from ψ. Applying the sigma derivation algorithm for σ yields the following ill-typed term:

$$
\begin{aligned}
&\sigma :: (F\ b \to b) \to G\ b \to b \\
&\sigma(\beta) = \textbf{case } \beta\ t \textbf{ of} \\
&\qquad\qquad (1, ()) \to (1, ()) \\
&\qquad\qquad (2, (a, l, r)) \to (2, (a, \mathit{mapT}\ (+1)\ l, \mathit{mapT}\ (+1)\ r))
\end{aligned}
$$

This term is ill-typed because according to functor G, variable l should have the polymorphic type b, but $\mathit{mapT}\ (+1)$ is applied to l, and $\mathit{mapT}\ (+1)$ expects an argument of type *Tree Int*.

Summarizing, the derivation algorithm for σ can fail if recursive terms are not required to be variables. □

Appendix B

HFusion internals

In this chapter we present a brief description of the representation we use for hylomorphisms as well as some practical considerations about our algorithms.

To improve readability we use $\overrightarrow{vs}$ to denote a tuple or list of variables, the same notation being used for terms and patterns as well.

The functions processed by the tool are kept internally in a form which combines generalized paramorphisms with mutual hylomorphisms. A mutual generalized paramorphisms with m components

$$\{\!|(\phi_1, \ldots, \phi_m), (\psi_1, \ldots, \psi_m)|\!\}_{\langle F_1, \ldots, F_m \rangle}$$

is represented as a list of tuples

$$\begin{array}{l} \textbf{type } \mathit{Hylo}\ a\ ca = [(\mathit{Algebra}\ a, \mathit{Functor}, \mathit{Coalgebra}\ ca)] \\ [(\phi_1, F_1, \psi_1), \ldots, (\phi_m, F_m, \psi_m)] :: \mathit{Hylo}\ a\ ca \end{array}$$

The type parameters a and ca allow to distinguish the different kinds of algebras and coalgebras that the mutual generalized paramorphism can have (i.e. in_F, $\tau(\phi)$, out_F, $\sigma(\psi)$).

When first derived, a coalgebra ψ_i is in the form

$$\begin{array}{l} \lambda\overrightarrow{vs} \rightarrow \textbf{case } \overrightarrow{ts}_0 \textbf{ of} \\ \qquad\qquad \overrightarrow{ps}_1 \rightarrow (1, (t_{11}, \ldots, t_{1l_1})) \\ \qquad\qquad \ldots \\ \qquad\qquad \overrightarrow{ps}_n \rightarrow (n, (t_{n1}, \ldots, t_{1l_n})) \end{array}$$

which we represent as:

$$\begin{array}{l} \textbf{data } \mathit{Boundvar} = \mathit{Bvar}\ \mathit{Variable} \\ \qquad\qquad\qquad\qquad \mid\ \mathit{Bvtuple}\ [\mathit{Boundvar}] \\ \textbf{type } \mathit{Coalgebra}\ ca = ([\mathit{Boundvar}], [\mathit{Term}], ca) \\ \textbf{type } \mathit{Psi} = [([\mathit{Pattern}], \mathit{CoalgebraTuple})] \\ \textbf{type } \mathit{CoalgebraTuple} = [(\mathit{Variable}, \mathit{Term})] \\ \psi_i :: \mathit{Coalgebra}\ \mathit{Psi} \end{array}$$

$$\psi_i = (\overline{vs}, \overrightarrow{ts}_0, [(\overline{ps}_1, t_1), \ldots, (\overline{ps}_n, t_n)])$$
$$\textbf{where } t_i = [(u_{i1}, t_{i1}), \ldots, (u_{il_1}, t_{il_1})]$$

The variables u_{ij} are not used by the coalgebra, but are used as identifiers for the positions of the output tuples. By using those variables the functor will tell if the positions are recursive or not.

Functors are represented with the following types

$$\textbf{type } \mathit{Functor} = [(\mathit{Variable}, \mathit{Int}, \mathit{CopyTree})]$$
$$\textbf{type } \mathit{CopyTree} = \mathit{Rec}\ \mathit{Variable} \mid \mathit{NonRec}\ \mathit{Variable} \mid \mathit{CopyTuple}\ [\mathit{CopyTree}]$$

For each position in each output tuple of a coalgebra the functor tells two things:

- First, it tells if the position should be copied (for a paramorphism). For instance, $(u, 0, \mathit{CopyTuple}\ [\mathit{NonRec}\ u, \mathit{Rec}\ u])$ tells that position u should be copied as in $\lambda u \rightarrow (u, u)$.

- Second, it tells over which copies recursive calls should be applied. In a case like $(u, 0, \mathit{CopyTuple}\ [\mathit{NonRec}\ u, \mathit{Rec}\ u])$, the recursive call should be applied over the second copy, marked by constructor *Rec*. The 0 value in the triple indicates which recursive call should be made in the case of mutual recursion.

Natural transformations are represented through the following types

$$\textbf{type } \mathit{Etai} = [\mathit{EtaOp}]$$
$$\begin{aligned}\textbf{data } \mathit{EtaOp} = {} & \mathit{EOid} \\ \mid {} & \mathit{EOgeneral}\ [\mathit{Boundvar}]\ [\mathit{Term}] \\ \mid {} & \mathit{EOsust}\ [\mathit{Variable}]\ [\mathit{Term}]\ [\mathit{Boundvar}] \\ \mid {} & \mathit{EOlet}\ [\mathit{Term}]\ [\mathit{Pattern}]\ [\mathit{Variable}]\ [\mathit{Term}]\end{aligned}$$

Each constructor in the type *EtaOp* builds a different kind of natural transformation. These are the interpretations:

- *EOid* represents the identity.

- *EOgeneral* $\overline{vs}$ $\overrightarrow{ts}$ represents the lambda abstraction $\lambda\overline{vs} \rightarrow \overrightarrow{ts}$.

- *EOsust* $[v'_1, ..., v'_m]$ $[t_1, ..., t_m]$ $\overline{vs}$ represents the transformation $\lambda\overline{vs} \rightarrow \overline{vs}\ [(t_1, ..., t_m)\ /\ (v'_1, ..., v'_m)]$, where $[/]$ denotes substitution.

- *EOlet* $\overrightarrow{t0}$ $\overline{ps}$ $\overline{vs}$ $\overrightarrow{ts}$ represents the transformation

$$\lambda\overline{vs} \rightarrow \textbf{case } \overrightarrow{t0} \textbf{ of } \overline{ps} \rightarrow \overrightarrow{ts}$$

A value $[e_1, \ldots, e_n]$::*Etai* represents a composition of natural transformations $e_1 \circ \cdots \circ e_n$.

The representation for algebras:

$$(\lambda \overrightarrow{vs}_1 \to t_1) \triangledown \cdots \triangledown (\lambda \overrightarrow{vs}_m \to t_m)$$

is of the form:

$$\begin{array}{l}
\textbf{type}\ \mathit{Acomponent}\ a = ([\mathit{Boundvar}], \mathit{TermWrapper}\ a) \\
\textbf{type}\ \mathit{Algebra}\ a = [\mathit{Acomponent}\ a] \\
\textbf{data}\ \mathit{TermWrapper}\ a = \mathit{TWcase}\ \mathit{Term}\ [\mathit{Pattern}]\ [\mathit{TermWrapper}] \\
\qquad\qquad\qquad\qquad\quad \mid\ \mathit{TWeta}\ (\mathit{TermWrapper}\ a)\ \mathit{Etai} \\
\qquad\qquad\qquad\qquad\quad \mid\ \mathit{TWsimple}\ a \\
\qquad\qquad\qquad\qquad\quad \mid\ \mathit{TWacomp}\ (\mathit{Acomponent}\ a) \\
\qquad\qquad\qquad\qquad\quad \mid\ \mathit{TWbottom} \\
[(\overrightarrow{vs}_1, t'_1), \ldots, (\overrightarrow{vs}_m, t'_m)] :: \mathit{Algebra}\ a
\end{array}$$

where t'_i is a value of type *TermWrapper a* representing term t_i. The type parameter a serves to distinguish algebras of the form in_F, $\tau(\phi)$ and generic. For the generic case, the type parameter a is instantiated as the type *Term*, which represents terms in our core language.

The type *TermWrapper a* describes a term where certain subterms have been tagged as belonging to natural transformations. We translate values of type *TermWrapper a* into core language terms relative to a given tuple of input variables. The notation $\ll \cdot \gg$ is used in this context to indicate translation. Terms and meta expressions appear mixed in the definition below.

$$\begin{array}{l}
\ll \cdot \gg :: \mathit{Acomponent}\ a \to \mathit{Term} \\
\ll (\overrightarrow{vs}, t) \gg = \lambda \overrightarrow{vs} \to \ll t \gg_{\overrightarrow{vs}}\ \ (v_1, \ldots, v_n) \\
\ll \mathit{TWcase}\ t_0\ [p_1, \ldots, p_n]\ [t_1, \ldots, t_n] \gg_{\overrightarrow{vs}} = \\
\qquad ((\lambda(\overrightarrow{vs}, \overrightarrow{bv(p_1)}) \to \ll t_1 \gg_{\overrightarrow{vs}}\ \ \overrightarrow{vs}) \\
\qquad\quad + \cdots + \\
\qquad\quad (\lambda(\overrightarrow{vs}, \overrightarrow{bv(p_n)}) \to \ll t_n \gg_{\overrightarrow{vs}}\ \ \overrightarrow{vs}) \\
\qquad)\circ \\
\qquad (\lambda \overrightarrow{vs} \to \textbf{case}\ t_0\ \textbf{of}\ p_1 \to (1, (\overrightarrow{vs}, \overrightarrow{bv(p_1)})) \\
\qquad\qquad\qquad\qquad\quad \vdots \\
\qquad\qquad\qquad\qquad\quad p_n \to (n, (\overrightarrow{vs}, \overrightarrow{bv(p_n)}))) \\
\ll \mathit{TWeta}\ t\ \eta \gg_{\overrightarrow{vs}} \quad = \ll t \gg_{\overrightarrow{vs}} \circ \eta \\
\ll \mathit{TWsimple}\ a \gg_{\overrightarrow{vs}} \quad = \lambda \overrightarrow{vs} \to \ll a \gg \quad \text{-- to be specified later} \\
\ll \mathit{TWacomp}\ acomp \gg_{\overrightarrow{vs}} \ = \ \ll acomp \gg \\
\ll \mathit{TWbottom} \gg_{\overrightarrow{vs}} \quad = \bot
\end{array}$$

The expression $\overrightarrow{bv(p_i)}$ denotes a tuple with the variables bound by pattern p_i which may appear in term t_i. We require that t_0 does not contain any

reference to recursive variables (i.e. results of recursive calls). Thus, we have that the lambda abstraction containing the **case** is a natural transformation.

The main purpose of type *TermWrapper a* is to handle branching cases in natural transformations. In every hylomorphism, there is one algebra component (*Acomponent a*) for each coalgebra alternative. A value of type *TermWrapper a* allows to branch off (via *TWcase* values) an alternative into others.

Having presented these constructs, we will show next some sample functions that can be expressed in terms of them. First, we will show function *map*.

$$
\begin{aligned}
&map\ f = [\![\phi, out_F]\!]_F \\
&\quad \textbf{where } F = \overline{\mathbf{1}} + a \times I \\
&\qquad\quad \phi\ (1, ()) = [\,] \\
&\qquad\quad \phi\ (2, (x, xs)) = f\ x : xs
\end{aligned}
$$

In our internal representation, this function can be expressed as

$$
\begin{aligned}
&map :: Hylo\ \ Term\ \ Psi \\
&map\ f = [([\phi_1, \phi_2], [(u_1, 0, Rec\ u_1)], \psi)] \\
&\quad \textbf{where } \phi_1 = ([\,], TWsimple\ [\,]) \\
&\qquad\quad \phi_2 = ([x, u_1], [\,TWeta\ (TWsimple\ (x : u_1)) \\
&\qquad\qquad\qquad\qquad (EOsust\ [x]\ [f\ x]\ [x, u_1])]) \\
&\qquad\quad \psi = ([ls], [ls], [([[\,]], [\,]), ([x : xs], [(u_0, x), (u_1, xs)])])
\end{aligned}
$$

For the sake of readability, we are showing terms and meta-expressions mixed within the same syntax.

Another example is the primitive recursive function *dropWhile*:

$$
\begin{aligned}
&dropWhile\ p\ [\,] \quad\ = [\,] \\
&dropWhile\ p\ (x : xs) = \textbf{if } p\ x \textbf{ then } dropWhile\ p\ xs \\
&\qquad\qquad\qquad\qquad\quad \textbf{else } x : xs
\end{aligned}
$$

which can be expressed as the following paramorphism

$$
\begin{aligned}
&dropWhile\ p = \langle\!|\phi|\!\rangle_F \\
&\quad \textbf{where } F = \overline{\mathbf{1}} + a \times I \\
&\qquad\quad \phi\ (1, ()) = [\,] \\
&\qquad\quad \phi\ (2, (x, (xs, vs))) = \textbf{if } p\ x \textbf{ then } vs \\
&\qquad\qquad\qquad\qquad\qquad\quad \textbf{else } x : xs
\end{aligned}
$$

In our internal representation, it can be written as follows:

$$
\begin{aligned}
&dropWhile :: Hylo\ \ Term\ \ Psi \\
&dropWhile\ p = [([\phi_1, \phi_2], \\
&\qquad\qquad\quad [(u_1, 0, CopyTuple\ [NonRec\ u_1, Rec\ u_1])], \psi)] \\
&\quad \textbf{where } \phi_1 = ([\,], TWsimple\ [\,]) \\
&\qquad\quad \phi_2 = ([x, (xs, u_1)], TWcase\ (p\ x)
\end{aligned}
$$

$$
\begin{array}{l}
\qquad\qquad\qquad\qquad\qquad\qquad\quad [\,True, False\,] \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad [\,TWsimple\ u_1,\ TWsimple\ (x : u_1)\,]) \\
\psi = ([ls], [ls], [([[\,]], [\,]), ([x : xs], [(u_0, x), (u_1, xs)])])
\end{array}
$$

Let us consider the *zip* function as an example of a definition which recurses over multiple arguments.

$$
\begin{array}{l}
zip :: ([a], [b]) \to [(a, b)] \\
zip = [\![in_F, \psi]\!]_F \\
\quad \textbf{where } F = \overline{\mathbf{1}} + (a \times b) \times I \\
\qquad\quad \psi\ (a : as, b : bs) = (2, ((a, b), (as, bs))) \\
\qquad\quad \psi\ (\_, \_) = (1, ())
\end{array}
$$

In our internal representation it is expressed as:

$$
\begin{array}{l}
zip :: Hylo\ Term\ Psi \\
zip = [([\phi_1, \phi_2], [(u_1, 0, Rec\ u_1)], \psi)] \\
\quad \textbf{where } \phi_1 = ([z, u_1], TWsimple\ [z : u_1]) \\
\qquad\quad \phi_2 = ([\,], TWsimple\ [\,]) \\
\qquad\quad \psi = ([xss, yss], [xss, yss], \\
\qquad\qquad [([x : xs, y : ys], [(u_0, (x, y)), (u_1, (xs, ys))]), ([\_, \_], [\,])])
\end{array}
$$

Now let us consider an example involving mutually recursive definitions.

$$
\begin{array}{l}
(sumRose, sumForest) = [\![\phi_1, \phi_2, out_F]\!]_F \\
\quad \textbf{where } F = \langle \overline{Int} \times \Pi_2, \overline{\mathbf{1}} + \Pi_1 \times \Pi_2 \rangle \\
\qquad\quad \phi_1\ (1, (a, fr)) = a + fr \\
\qquad\quad \phi_2\ (1, ()) \qquad\ = 0 \\
\qquad\quad \phi_2\ (2, (r, fr)) = r + frsumRose :: Rose\ Int \to Int
\end{array}
$$

In the internal representation:

$$
\begin{array}{l}
sumRose :: Hylo\ Term\ Psi \\
sumRose = [([([a, u_1], TWsimple\ (a + u_1))], [(u_1, 1, Rec\ u_1)], \psi_1) \\
\qquad\qquad , ([([\,], TWsimple\ 0), ([u_0, u_1], TWsimple\ (u_0 + u_1))] \\
\qquad\qquad\ , [(u_0, 0, Rec\ u_0), (u_1, 1, Rec\ u_1)], \psi_2)] \\
\quad \textbf{where } \psi_1 = ([t], [t], [([Branch\ a\ fr], [(u_0, a), (u_1, fr)])]) \\
\qquad\quad \psi_2 = ([ls], [ls], [([[\,]], [\,]), ([r : fr], [(u_0, r), (u_1, fr)])])
\end{array}
$$

B.1 Law application

We start by showing the result of applying the fold-unfold law to the composition $map\ f \circ map\ g$:

$$
\begin{aligned}
&mm\ f\ g = (\!|\phi|\!)_F \\
&\quad \textbf{where } F = \overline{\mathbf{1}} + a \times I \\
&\qquad\quad \phi\ (1, ()) = [\,] \\
&\qquad\quad \phi\ (2, (x, xs)) = f\ (g\ x) : mm\ f\ g\ xs
\end{aligned}
$$

which in the internal representation is given by:

$$
\begin{aligned}
&mm :: Hylo\ Term\ Psi \\
&mm\ f\ g = [([\phi_1, \phi_2], [(u_1, 0, Rec\ u_1)], \psi)] \\
&\quad \textbf{where } \phi_1 = ([\,], TWacomp\ ([\,], TWsimple\ [\,])) \\
&\qquad\quad \phi_2 = ([x, u_1], TWeta\ (TWacomp \\
&\qquad\qquad\qquad\qquad ([x, u_1], TWsimple\ (f\ x : u_1)) \\
&\qquad\qquad\qquad\qquad (EOsust\ [x]\ [g\ x]\ [x, u_1]))) \\
&\qquad\quad \psi = ([ls], [ls], [([[\,]], [\,]), ([x : xs], [(u_0, x), (u_1, xs)])])
\end{aligned}
$$

In order to represent the result of applying the fold-hylo law, we need other declarations, which allow to represent algebra transformers τ.

$$
\begin{aligned}
&\textbf{data } Tau = Tauphi\ (TauTerm\ Term) \\
&\qquad\qquad\ \ |\ Tautau\ (TauTerm\ Tau) \\
&\textbf{data } TauTerm\ a = Taucons\ Constructor\ [TauTerm\ a] \\
&\qquad\qquad\qquad\qquad\qquad\qquad ([Boundvar], TermWrapper\ a) \\
&\qquad\qquad\qquad\quad |\ Tausimple\ Term \\
&\qquad\qquad\qquad\quad |\ Taupair\ Term\ (TauTerm\ a) \\
&\qquad\qquad\qquad\quad |\ Taucata\ (Term \to Term)\ (TauTerm\ a)
\end{aligned}
$$

We can represent algebras $\tau(\phi)$ with values of type *Algebra Tau*, which will allow us to express the result of the fold-hylo law. The following is the translation of values of type *TauTerm a* into our core language:

- $\ll Taucons\ c\ ts\ \phi_i \gg = \ll \phi_i \gg\ \ll ts \gg$
  This term stands for an abstraction of the constructor c when the transformer was derived. The term ϕ_i is the algebra component used to substitute the constructor.

- $\ll Tausimple\ t \gg = t$

- $\ll Taupair\ t\ \tau \gg = (t, \ll \tau \gg)$

- $\ll Taucata\ f\ t \gg = f\ \ \ll t \gg$
  The function f constructs a *Term* application of some catamorphism over the t argument. It is used while deriving τ to handle the case where a term in the algebra does not contain references to recursive variables but it is used in recursive positions of the abstracted constructors.

The following is the result of fusing $map\ f \circ dropWhile\ p$.

$$
\begin{array}{l}
dm\ f\ p = \langle\!|\tau(\phi_1 \triangledown \phi_2)|\!\rangle_F \\
\quad \textbf{where}\ F = \overline{\mathbf{1}} + a \times I \\
\qquad\quad \phi_1\ () = [\,] \\
\qquad\quad \phi_2\ (x, xs) = f\ x : xs \\
\qquad\quad \tau(\alpha_1 \triangledown \alpha_2)\ (1, ()) = \alpha_1 \\
\qquad\quad \tau(\alpha_1 \triangledown \alpha_2)\ (2, (x, (xs, vs))) = \textbf{if}\ p\ x\ \textbf{then}\ vs \\
\qquad\qquad\qquad\qquad\qquad\qquad\quad \textbf{else}\ \alpha_2\ (x, xs)
\end{array}
$$

We can write this paramorphism in our internal representation as follows

```
dm f p :: Hylo Tau Psi
dm f p = [([φ1, φ2], [eid, eid],
           [(u1, 0, CopyTuple [NonRec u1, Rec u1])], ψ)]
  where eid = Eta [] []
        φ1 = ([], TWsimple (Tauphi (Taucons "[]" [] ([], TWsimple [])
                                                    EOid)))
        φ2 = ([x, (xs, u1)]
              , TWcase (p x)
                       [True, False]
                       [TWsimple (Tauphi (Tausimple u1))
                       , TWsimple
                         (Tauphi (Taucons ":"
                                          [Tausimple x, Tausimple u1]
                                          ([x, u1], TWsimple (f x : u1))))])
        ψ = ([ls], [ls], [([[]], []), ([x : xs], [(u0, x), (u1, xs)])])
```

Next, we present the data structures for representing coalgebras in the form $(\psi'_1, \ldots, \psi'_m) = \sigma(\psi_1, \ldots, \psi_k)$ resulting from applying the hylo-unfold law to compositions of the form $\{\!|\phi, \sigma(in_F)|\!\}_G \circ_j [\![(\psi_1, \ldots, \psi_n)]\!]_F$, where $\circ_j$ is defined as follows:

$$
\begin{array}{l}
(f_1, \ldots, f_n) \circ_j (g_1, \ldots, g_n) = (h_1, \ldots, h_n) \\
\quad \textbf{where}\ h_i\ (a_1, \ldots, a_m) = f_i\ (a_1, \ldots, g_i\ a_j, \ldots, a_m)
\end{array}
$$

A coalgebra

$$
\begin{array}{l}
\psi'_i = \lambda \overrightarrow{vs} \rightarrow \textbf{case}\ \overrightarrow{t0}\ \textbf{of} \\
\qquad\qquad (\overrightarrow{p_{11}}, \ldots, p_{1k}) \rightarrow (1, (t_{11}, \ldots, t_{1l_1})) \\
\qquad\qquad \ldots \\
\qquad\qquad (p_{n1}, \ldots, p_{nk}) \rightarrow (n, (t_{n1}, \ldots, t_{1l_n}))
\end{array}
$$

is represented as:

```
newtype Sigma =
        Sigma ([[TupleTerm]], [[PatternS]]
```

$$
\begin{aligned}
&\qquad\qquad\qquad\qquad , [\mathit{Maybe}\ (\mathit{Int}\\
&\qquad\qquad\qquad\qquad\qquad\quad , [([\mathit{Boundvar}], \mathit{TermWrapper\ Constructor})]\\
&\qquad\qquad\qquad\qquad\qquad\quad , \mathit{WrappedCA}, \mathit{Int} \to \mathit{Term} \to \mathit{Term})\\
&\qquad\qquad\qquad\qquad\ ])\\
&\textbf{data}\ \mathit{PatternS} = \mathit{PcaseS\ Pattern}\\
&\qquad\qquad\qquad\ \mid \mathit{PcaseSana\ Int\ Pattern}\\
&\qquad\qquad\qquad\ \mid \mathit{PcaseR\ Int\ Constructor}\ [\mathit{PatternS}]\\
&\textbf{data}\ \mathit{WrappedCA} = \mathit{WCApsi}\ (\mathit{Coalgebra\ Psi})\\
&\qquad\qquad\qquad\quad\ \mid \mathit{WCAsigma}\ (\mathit{Coalgebra\ Sigma})\\
&\psi'_i :: \mathit{Coalgebra\ Sigma}\\
&\psi'_i = (\overrightarrow{vs}, \overrightarrow{t0}, \mathit{Sigma}\ ([t_1,\dots,t_n],[[p'_{11},\dots,p'_{1k}],\dots,[p'_{n1},\dots,p'_{nk}]], l_{\psi'_i}))\\
&\quad \textbf{where}\ \ t_i = [(u_{i1},t_{i1}),\dots,(u_{il_1},t_{il_1})]
\end{aligned}
$$

The expression p'_{ij} denotes a value of type *PatternS* representing pattern p_{ij}. The j^{th} element in the list $l_{\psi'_i}$ is the coalgebra component of the mutual unfold in the original composition which was fused over the j^{th} recursive argument of the i^{th} component of $\sigma(out_F)$. Each coalgebra component in $l_{\psi'_i}$ has attached the index identifying its position in the mutual unfold, the corresponding algebra component of the mutual unfold, and a function f that constructs a *Term* application of a given component of the mutual unfold to a given *Term*.

The following is the translation of values of type *PatternS* to our core language patterns. Each value of type *PatternS* is associated with a specific recursive argument of a component of $\sigma(out_F)$.

- $\ll \mathit{PcaseS}\ p \gg = p$
- $\ll \mathit{PcaseSana}\ i\ p \gg = [\![\psi_i]\!]_F \cdot p$ where ψ_i is the coalgebra component in the mutual unfold in the original composition.
- $\ll \mathit{PcaseR}\ i\ c_j\ [p_1,\dots,p_n] \gg = \psi_i \cdot (j, (\ll p_1 \gg, \dots, \ll p_n \gg))$ where ψ_i is the same as above.

We show now the result of fusing $zip \circ map\ f$:

$$
\begin{aligned}
&\mathit{zm}\ f = [\![in_G, \sigma(\psi)]\!]_G\\
&\quad \textbf{where}\ G = \overline{\mathbf{1}} + (\overline{a} \times \overline{b}) \times I\\
&\qquad\qquad \sigma(\beta)\ (\beta\ (2,(x,xs)))\ (y:ys) = (2,((x,y),(xs,ys)))\\
&\qquad\qquad \sigma(\beta)\ \_\ \_ = (1,())\\
&\qquad\qquad \psi\ [\,] = (1,())\\
&\qquad\qquad \psi\ (x:xs) = (2,(f\ x, xs))
\end{aligned}
$$

which in our internal representation looks like:

$$
\begin{aligned}
&\mathit{zm}\ f :: \mathit{Hylo\ Term\ Sigma}\\
&\mathit{zm}\ f = [([\phi_1,\phi_2],[(u_1,0,\mathit{Rec}\ u_1)],([l_1,l_2],[l_1,l_2],\mathit{Sigma}\ \sigma))]
\end{aligned}
$$

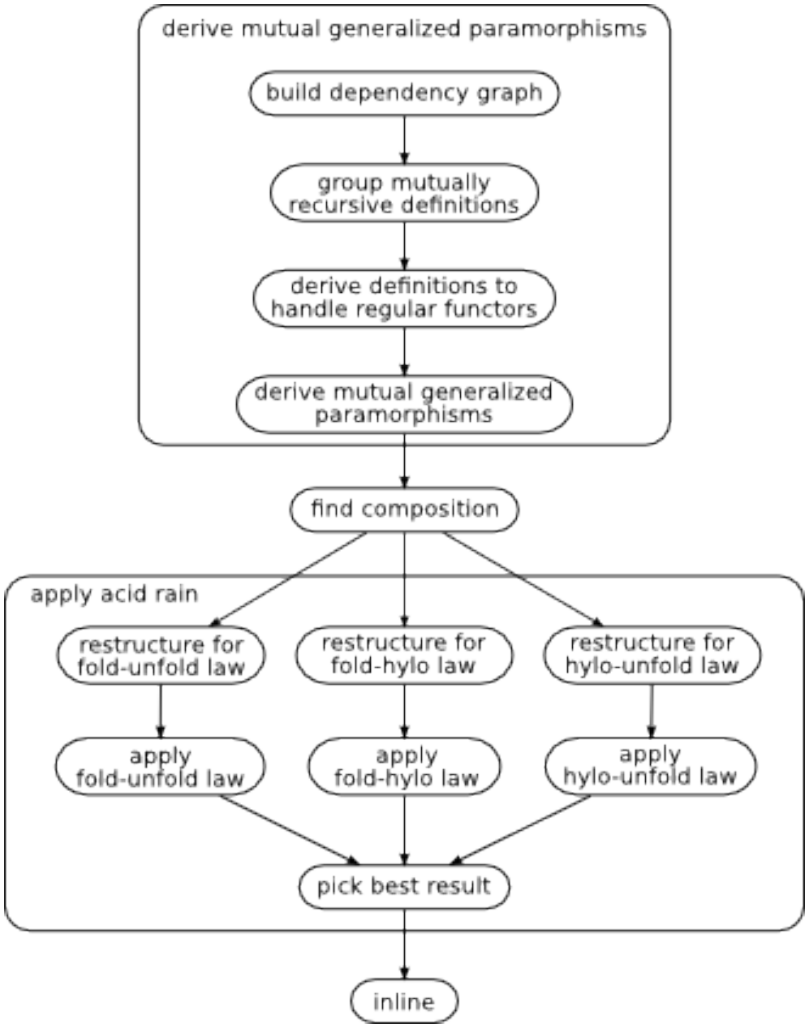

Figure B.1: HFusion stages

enough that this functor does not match the intermediate data type signature, because the fold or the unfold does not traverse all of the recursive positions of the constructors. Deriving transformers τ and σ also requires a similar knowledge.

It could happen that no law is applicable. This can happen if the hylomorphisms in the composition are the outcome of applying the fold-hylo or the hylo-unfold cases (e.g. in the form $(\!|\phi|\!)_F \circ [\![\tau(\phi'), \psi]\!]_G$ or $[\![\phi, \sigma(\psi)]\!]_G \circ (\!|\psi'|\!)_F$). In such a case, the hylomorphisms are inlined and rederived. The inlining performs certain simplifications of the hylomorphism structures that may allow to apply again acid rain laws in some cases.