Generic Accumulations for Program Calculation

Mauro Jaskelioff
Facultad de Cs. Exactas, Ingeniería y Agrimensura
Universidad Nacional de Rosario
Rosario - Argentina
mauro@fceia.unr.edu.ar

December 2004

Supervised by:

Alberto Pardo
Instituto de Computación
Facultad de Ingeniería
Julio Herrera y Reissig 565 - Piso 5
11300 Montevideo
Uruguay

Abstract

Accumulations are recursive functions widely used in the context of functional programming. They maintain intermediate results in additional parameters, called accumulators, that may be used in later stages of computing. In a former work [Par02] a generic recursion operator named *afold* was presented. *Afold* makes it possible to write accumulations defined by structural recursion for a wide spectrum of datatypes (lists, trees, etc.). Also, a number of algebraic laws were provided that served as a formal tool for reasoning about programs with accumulations.

In this work, we present an extension to *afold* that allows a greater flexibility in the kind of accumulations that may be represented. This extension, in essence, provides the expressive power to allow accumulations to have more than one recursive call in each subterm, with different accumulator values —something that was not previously possible. The extension is conservative, in the sense that we obtain similar algebraic laws for the extended operator. We also present a case study that illustrates the use of the algebraic laws in a calculational setting and a technique for the improvement of fused programs that do not eliminate all intermediate structures.

Contents

Chapter 1

Introduction

The aim of this work is to present a theoretical framework and associated techniques that help in the calculation of programs with accumulators. Program calculation includes the derivation and optimization of programs as well as the verification of their properties. We will see programs as algebraic structures that can be manipulated by algebraic laws, and focus on one kind of algebraic structure that models recursion operators.

Recursion operators on datatypes are a common tool that functional programmers use to structure programs. These operators abstract common patterns of recursion according to the data structure they manipulate. By expressing a program with these encapsulated patterns of recursion, a number of associated laws are obtained for free. Another benefit of using recursion operators is that they can be parameterised by the structure of the datatype they use, making programs more general. This approach, known as generic programming, consists of an algebraic model of datatypes and programs that allows us to obtain an abstract description of datatypes and to define programs that operate on this abstract description. Given an instance of the abstract datatype we will have an instance of the abstract program for that specific datatype. This algebraic approach also serves as a formal basis to obtain algebraic laws and a smooth proof framework suitable for the calculation of functional programs.

Functional programs are usually obtained by gluing together the solutions to subproblems by means of functional composition [Hug89]. This compositional style is favored by programmers because it has the advantage of producing modular and easy to understand programs. Nevertheless, it is often the case that programs written in this style are not efficient. In a functional composition $f \circ g$, an intermediate data structure has to be generated by g only to be consumed immediately by f. This source of inefficiency can often be removed by a technique called *deforestation* [Wad90], which makes it possible, under certain conditions, to derive a program that does not build the intermediate structures. One of the advantages of using recursion operators is that they provide a class of algebraic laws that correspond to deforestation, called *fusion* laws.

Another technique frequently used by functional programmers is the generalization of functions by the addition of an extra parameter that is used to pass intermediate results to recursive calls. These functions that keep intermediate results in additional parameters are called *accumulations*. Accumulations are usually introduced to gain expressiveness or to optimize an inefficient function.

This thesis provides an extension to a recursion operator for accumulations called *afold* [Par02] and its algebraic laws. A special emphasis has been put on the pragmatics of these laws for program calculation. Accordingly, a case study is provided showing the use of these laws in a practical situation.

The study of fusion in accumulations has a long history. In the seminal work [Bir84], the fusion of accumulations was introduced as an optimization technique. More recently, there has been a considerable amount of research activity focused on the fusion of accumulations. In [HIT96], higher-order folds are used to represent accumulations. In [CDPR98, Cor99] the fusion of programs with accumulating parameters is based on the descriptional composition of attribute grammars. In [VK04] the fusion of accumulations is obtained by means of macro tree transducers.

1.1 The *fold* Recursion Operator

The *fold* recursion operator encapsulates the pattern of recursion of functions that are structured according to the data structure that they consume [Bir98, Hut99].

Folds over lists correspond to the well-known foldr operator:

$$\begin{array}{lcl} \mathsf{foldr} & : & (\alpha \to \beta \to \beta) \to \beta \to [\alpha] \to \beta \\ \mathsf{foldr}\ \oplus\ e\,[\,] & = & e \\ \mathsf{foldr}\ \oplus\ e\,(x : xs) & = & x \oplus (\mathsf{foldr}\ \oplus\ e\ xs) \end{array}$$

Functions that are defined by structural recursion on lists can be expressed with foldr. For example, sum, the function that sums all elements of a list of natural numbers, can be expressed as:

$$\begin{array}{lcl} \mathsf{sum} & : & [\mathsf{nat}] \to \mathsf{nat} \\ \mathsf{sum} & = & \mathsf{foldr}\ (+)\ 0 \end{array}$$

One of the *fusion laws* of foldr is:

$$f\ (a \oplus b) = a \otimes f\ b \qquad \Rightarrow \qquad f \circ \mathsf{foldr}\ \oplus\ e = \mathsf{foldr}\ \otimes\ (f\ e)$$

Using the foldr fusion law we can prove that

$$(n+) \circ \mathsf{foldr}\ (+)\ 0 = \mathsf{foldr}\ (+)\ n$$

where $(n+)\colon \mathsf{nat} \to \mathsf{nat}$ is the function that adds n to its argument.

Another law associated with fold is the *map-fold fusion*:

$$\mathsf{foldr}\ \oplus\ e \circ \mathsf{map}\ f = \mathsf{foldr}\ \otimes\ e \qquad\qquad \text{where } x \otimes y = f x \oplus y$$

Here $\mathsf{map}\colon (\alpha \to \beta) \to [\alpha] \to [\beta]$ is the function that applies a given function f to every element of a list:

$$\mathsf{map}\ f\ [x_1, \ldots, x_n] = [f\ x_1, \ldots, f\ x_n]$$

Consider the constant function one:

$$\begin{array}{lcl} \mathsf{one} & : & \alpha \to \mathsf{nat} \\ \mathsf{one}\ a & = & 1 \end{array}$$

We can calculate the length of a list with the length function,

$$\mathsf{length} \ = \ \mathsf{sum} \circ \mathsf{map}\ \mathsf{one}$$

Using map-fold fusion we can obtain a definition of length that does not create an intermediate structure.

$$\begin{array}{lcl} \mathsf{length} & = & \mathsf{foldr}\ \otimes\ 0 \\ & & \text{where } x \otimes y = 1 + y \end{array}$$

1.2 Accumulators

Accumulations are recursive functions that keep intermediate results in additional parameters, called *accumulating parameters*. The use of accumulations in functional programming is widespread, and the associated accumulation technique is well known [Bir84, Bir98]. To define an accumulation two techniques may be used. One is by currying [Bir98, Tho99], a standard technique based on the higher order feature of modern functional programming languages. Using this technique one may think of any function on multiple arguments as a function on one argument that returns another function as a result. The relation between these two ways of representing functions on multiple arguments can be expressed bt means of the curry-uncurry isomorphism.

$$\begin{array}{lcl} \mathsf{curry} & : & ((\alpha,\beta)) \to \gamma) \to (\alpha \to \beta \to \gamma) \\ \mathsf{curry}\ f\ x\ y & = & f\ (x,y) \\ \\ \mathsf{uncurry} & : & (\alpha \to \beta \to \gamma) \to ((\alpha,\beta) \to \gamma) \\ \mathsf{uncurry}\ f\ (x,y) & = & f\ x\ y \end{array}$$

This isomorphism means that $(\alpha,\beta) \to \gamma \cong \alpha \to \beta \to \gamma$.

Using currying an accumulation may be defined as a higher-order fold. Consider, for example, the linear-time function that reverses a list,

$$\begin{array}{lcl} \mathsf{reverse} & : & [\alpha] \to [\alpha] \\ \mathsf{reverse}\ xs & = & \mathsf{rev}\ xs\ [\,] \\ \\ \mathsf{rev} & : & [\alpha] \to [\alpha] \to [\alpha] \\ \mathsf{rev}\ [\,]\ ys & = & ys \\ \mathsf{rev}\ (x : xs)\ ys & = & \mathsf{rev}\ xs\ (x : ys) \end{array}$$

The function rev may be defined using a higher-order fold:

$$\mathsf{rev} = \mathsf{foldr}\ (\lambda x\ f\ ys.f\ (x : ys))\ \mathsf{id}$$

The alternative to currying is tupling. Functions defined in this manner cannot be written in terms of a fold, since fold cannot express functions with multiple arguments unless currying and higher-order are used as it was shown before. This means that to express rev with tupling we need a new operator that acts as a sort of *fold with accumulators*.

A fold with accumulators, named *afold*, was introduced in [Par00]. This operator is able to express functions with accumulations without resorting to higher-order. For example, let us consider the expression for an afold on lists.

$$\begin{array}{lcl} \mathsf{afold}(h_1,h_2,\psi)\ ([\,],x) & = & h_1(x) \\ \mathsf{afold}(h_1,h_2,\psi)\ ((a:\ell),x) & = & h_2(a,\mathsf{afold}(h_1,h_2,\psi)(\ell,\psi(a,x)),x) \end{array}$$

We can define rev in terms of afold:

$$\mathsf{rev} = \mathsf{afold}(\mathsf{id},\mathsf{snd},(:)) \qquad \text{where}\ \ \mathsf{snd}(x,y,z) = y$$

The pattern of recursion of the fold operator follows the recursive structure of the input datatype, i.e. each recursive call matches up with a recursive instance in the definition of the input datatype.

In accumulations, the pattern of recursion is not only determined by the input datatype, but also by the accumulating parameter. When defining a fold with accumulators, we have to make a choice of whether we are going to allow a given recursive call to be made with different accumulating functions or not. In the generic recursive operator *afold*, each recursive call may only have one accumulating function. Consider the definition of th afold for lists given above. If we wanted to define a function with two recursive calls on ℓ with different accumulator values, we would not be able to express this function as an afold. For example the following function cannot be expressed as an afold.

$$\begin{aligned} \mathsf{subs}\,([\,], y) &= [[y]] \\ \mathsf{subs}\,((x : \ell), y) &= \mathsf{subs}\,(\ell, y) \mathbin{+\!\!+} \mathsf{map}\,(y\!:)\,(\mathsf{subs}\,(\ell, x)) \end{aligned}$$

In this thesis, we present an extension to the generic definition of accumulations provided by *afold* which allows us to materialize the structure of the input datatype making it possible to express functions such as the one above.

1.3 Contributions

This work proposes an extension to the existing recursion operator afold, and it shows that this extension is conservative, in the sense that algebraic laws for the extended operator are similar to the ones for the existing operator. Also, a case study that illustrates the use of the newly presented operator and its laws in a program derivation setting is presented.

Several results are provided that aid the calculation of programs by simplifying certain equations that frequently appear when calculating with accumulations. Additionally, an example of the existing operator for a regular datatype is given —previous examples were limited to datatypes whose signature is captured by polynomial functors. Finally, a technique based on one of the obtained algebraic laws is introduced. This technique is useful for the improvement of fusions that do not eliminate all intermediate structures.

1.4 Overview of the Thesis

The remainder of the thesis is organized as follows:

- Chapter 2 introduces the mathematical framework the paper is based on. We review those notions of category theory that are used throughout the work. Then, we describe the category-theoretical modelling of datatypes and present the generic operator fold, along with its algebraic laws.
- Chapter 3 reviews the definition of the afold operator and algebraic laws presented in [Par02]. We also present some new laws that help in the calculation of programs. This chapter serves as preamble for the definition of the extended afold operator.
- Chapter 4 presents a technique for the optimization of functions that result from certain kinds of fusions that do not eliminate all intermediate structures. We present two examples to illustrate this technique.
- Chapter 5 presents the motivation and definition of our extension to afold, along with a reformulation of the laws in chapter 3 to cope with our proposed extension, as well as some new laws.

- Chapter 6 is a case study that shows the power of the extension applied to a well known accumulation [Bir84, HIT96]. This case study also serves as a guide to the pragmatics of some laws presented in the previous chapter.
- Chapter 7 summarizes this thesis.
- Appendix A lists some simple properties that were used in the case study in chapter 6.
- Appendix B provides the proofs of all the results in chapter 5.

Chapter 2

Preliminaries

This chapter introduces the mathematical tools and notation that will be used throughout the thesis.

We want to be able to model programs that are abstract in the sense that they are parameterised by one or more datatypes. We also want to be able to reason about programs. The category-theoretical model of type and programs gives us a generic representation of datatypes and an appropiate framework for reasoning algebraically about programs. This model is standard and has proved to be a fruitful approach to genericity.

The aim of this chapter is to introduce the concepts of category theory that we will be using and the use of these concepts in the construction of our generic representation of types and programs. In particular, we will introduce the generic fold operator, which is a generalisation of the classical foldr operator of functional languages and its associated algebraic laws.

Several introductions to this categorical approach are available (see e.g. [BJJM99, Hin99, LS81, MA86, JR97]) as well as to its applications to program calculation [Mal90, MFP91, Fok92, Jeu93, BdM97]. A brief introduction to category theory can be found in [Pie91]. More complete introductions can be found in, for example, [BW99, AL91]. The standard reference in category theory is [Lan71].

2.1 Categories

We will begin by defining the notion of category and presenting a variety of examples.

Definition 2.1 A category $\mathcal{C}$ comprises

1. a collection $Obj(C)$ of objects;

2. a collection of *arrows* or *morphisms*;

3. two total operations called *source* and *target*, which assign an object to an arrow. We shall write $f\colon A \to B$ to show that $source\ f = A$ and $target\ f = B$; the collection of all arrows with source A and target B is written $\mathcal{C}(A, B)$;

4. a composition operator assigning to each pair of arrows f and g with $target\ f = source\ g$ a composite arrow $g \circ f\colon source\ f \to target\ g$, which is associative: $f \circ (g \circ h) = (f \circ g) \circ h$;

5. for each object A, an *identity* arrow $\mathsf{id}_A\colon A \to A$ satisfying the *identity law*, which is that for any arrow $f\colon A \to B$, $\mathsf{id}_B \circ f = f$ and $f \circ \mathsf{id}_A = f$.

□

The following examples should give a more concrete idea of what a category looks like.

Example 2.2 *The category* **Set** *has sets as objects and total functions between sets as arrows. Composition of arrows is set-theoretical function composition. Identity arrows are identity functions.*

It should be noted that here the concept of function is strongly typed. What we know informally as the *square* function —the function that takes every real number r to r^2— may represent different arrows in **Set**. For example $square \colon \mathbb{R} \to \mathbb{R}$ is a different arrow from $square \colon \mathbb{R} \to \mathbb{R}^+$.

Example 2.3 *A* partial ordering $\leq_P$ *on a set* P *is a reflexive, transitive, and antisymmetric relation on the elements of* P*. An* order preserving *function from* $(P, \leq_P)$ *to* $(Q, \leq_Q)$ *is a function* $f \colon P \to Q$ *such that if* $p \leq_P p'$ *then* $f(p) \leq_Q f(p')$*.*

The category **Poset** *has partially-ordered sets as objects and order-preserving total functions as arrows.*

Categories corresponding to algebraic structures (monoids, groups, etc.) are common examples of categories.

Example 2.4 *A* monoid $(M, \cdot, e)$ *is an underlying set* M *equipped with a binary operation* $\cdot$ *from pairs of elements of* M *into* M *such that* $(x \cdot y) \cdot z = x \cdot (y \cdot z)$ *for all* $x, y, z \in M$ *and a distinguished element* e *such that* $e \cdot x = x = x \cdot e$ *for all* $x \in M$*. An homomorphism between two monoids* $(M, \cdot, e)$ *and* $(M', \odot, e')$ *is a function* $f \colon M \to M'$ *such that* $f(x \cdot y) = f(x) \odot f(y)$ *and* $f(e) = e'$*.*

The category **Mon** *has monoids as objects and monoid homomorphisms as arrows.*

In our model of types and programs, types are represented by objects and programs by arrows. The underlying category may be **Set** or another category like **Cpo**, the category that has complete partial orders as objects and continuous functions as arrows.

Definition 2.5 The *product* of two categories, $\mathcal{C}$ and $\mathcal{D}$, denoted by $\mathcal{C} \times \mathcal{D}$ has as objects pairs (A, B) of a $\mathcal{C}$-object A and a $\mathcal{D}$-object B and as arrows pairs (f, g) of a $\mathcal{C}$-arrow f and a $\mathcal{D}$-arrow g. Composition and identity arrows are defined pairwise: $(f, g) \circ (h, i) = (f \circ h, g \circ i)$ and $\mathsf{id}_{(A,B)} = (\mathsf{id}_A, \mathsf{id}_B)$.□

The product of categories can be generalized to n components. We will write $\mathcal{C}^n$ to denote the n-ary product $\mathcal{C} \times \ldots \times \mathcal{C}$.

2.1.1 Diagrams

As it was pointed out before, each arrow has a unique target and source. Writing the source and target of an arrow every time we refer to it may quickly become cumbersome. For this reason it is quite common to refer to an arrow $f \colon A \to B$ simply by the identifer f, when the type information is clear from context. A useful device for recording type information is a *diagram*. In a diagram an arrow $f \colon A \to B$ is represented as $A \xrightarrow{f} B$, and its composition with an arrow $g \colon C \to A$ is represented as $C \xrightarrow{g} A \xrightarrow{f} B$. For example, one can depict the type information in the equation $\mathsf{id}_B \circ f = f$ as

$$
\begin{tikzcd}
A \arrow[rr, "f"] \arrow[dr, "f"'] & & B \\
& B \arrow[ur, "\mathsf{id}_B"'] &
\end{tikzcd}
$$

Since any two paths in the diagram between the same pairs of objects depicts the same arrow, the diagram is said to *commute*.

Another example of a commuting diagram is the diagram that depicts the equation $h \circ f = k \circ g$. Note that we are not giving the type of the arrows, the type information can be obtained from the diagram.

$$\begin{tikzcd} A \arrow[r, "g"] \arrow[d, "f"'] & B \arrow[d, "k"] \\ C \arrow[r, "h"] & D \end{tikzcd}$$

2.1.2 Initial and Terminal Objects

Definition 2.6 An arrow $f\colon A \to B$ is an *isomorphism* if there is an arrow $f^{-1}\colon B \to A$, called the *inverse* of f, such that $f^{-1} \circ f = \mathsf{id}_A$ and $f \circ f^{-1} = \mathsf{id}_B$. In that case, A and B are said to be *isomorphic* and written $A \cong B$. When two objects are isomorphic it is often said that they are identical *up to isomorphism*. □

In **Set** the notion of isomorphism corresponds to the notion of bijection.

Definition 2.7 An object 0 of a category is called an *initial object* if, for every object A, there is exactly one arrow from 0 to A. □

Definition 2.8 An object 1 of a category is called a *terminal object* if, for every object A, there is exactly one arrow from A to 1, denoted by $!_A\colon A \to 1$. □

Example 2.9 *In* **Set** *the empty set* $\{\}$ *is the only initial object; for every set* S*, the empty function is the unique function from* $\{\}$ *to* S*. Every singleton set* $\{u\}$*, for some* u*, happens to be a final object.*

Many categorical notions, including initiality and terminality, are defined up to isomorphism. For example, for initiality, this means that all initial objects in a category are isomorphic to each other. Accordingly, we can choose any of them as a representative of the class as isomorphic objects are indistinguishable.

The following law is a direct consequence of finality:

$$A \xrightarrow{\;f\;} B \xrightarrow{\;!_B\;} 1 \quad = \quad A \xrightarrow{\;!_A\;} 1$$

In **Set**, arrows from a singleton set $\{u\}$ to the set A are in one-to-one correspondence with the elements of A. Because of this, arrows of the form $a\colon 1 \to A$ are usually thought of as *elements* of A. From this point of view, every application $f(a)$, for $f\colon A \to B$ and $a \in A$, is in one-to-one correspondence with a composition $f \circ a\colon 1 \to B$. This correspondence expresses the relationship between the pointwise and the point-free style for expressing functions. Whereas in the pointwise style a function is described by its application to arguments, in the point-free style a function is described exclusively in terms of functional composition. Reasoning about functions in point-free style is essentially algebraic manipulation of functional composition.

We will underline expressions that denote constant functions. For example the constant $7\colon \mathsf{Int}$ in our categorical notation will be $\underline{7}\colon 1 \to \mathsf{Int}$. An exception to this notation will be type constructors—e.g. $\mathsf{zero}\colon 1 \to \mathsf{nat}$.

Example 2.10 *In a typical functional programming language the following diagram would commute:*

$$
\begin{tikzcd}
\mathsf{Unit} \arrow[rr, "\mathsf{true}"] \arrow[dr, "\mathsf{zero}"'] & & \mathsf{Bool} \\
& \mathsf{Nat} \arrow[ur, "\mathsf{iszero}"'] &
\end{tikzcd}
$$

where zero *and* true *are constants and* iszero *is a predicate that tests for zero. Note that* zero *and* true *are not underlined since they are type constructors (of the* nat *and* bool *datatypes, respectively).*

Here, Unit *is a terminal object. The isomorphism between a constant* $a\colon A$ *and a constant arrow* $\underline{a}\colon \mathsf{Unit} \to A$ *can be made explicit:*

$$
\begin{array}{lcl}
tof & : & [\mathsf{Unit} \to A] \to A \\
tof\ \underline{a} & = & \underline{a}\ () \\
fromf & : & A \to [\mathsf{Unit} \to A] \\
fromf\ a & = & \underline{a}
\end{array}
$$

2.2 Functors and Natural Transformations

Functors are structure-preserving maps between categories.

Definition 2.11 Given two categories $\mathcal{C}$ and $\mathcal{D}$, a *functor* $F\colon \mathcal{C} \to \mathcal{D}$ is a map taking each $\mathcal{C}$-object A to a $\mathcal{D}$-object FA and each $\mathcal{C}$-arrow $f\colon A \to B$ to a $\mathcal{D}$-arrow $F(f)\colon FA \to FB$, such that for all $\mathcal{C}$-objects A and composable $\mathcal{C}$-arrows f and g the following conditions are satisfied:

1. $F(\mathsf{id}_A) = \mathsf{id}_{FA}$
2. $F(g \circ f) = Fg \circ Ff$

□

Example 2.12 *For each category* $\mathcal{C}$ *there exists an* identity functor $I\colon \mathcal{C} \to \mathcal{C}$ *that takes very* $\mathcal{C}$*-object and every* $\mathcal{C}$*-arrow to itself.*

Example 2.13 *The constant functor* $\underline{A}\colon \mathcal{C} \to \mathcal{D}$ *maps all* $\mathcal{C}$*-objects to the* $\mathcal{D}$*-object* A*, and all* $\mathcal{C}$*-arrows to the identity on* A*.*

Example 2.14 *The projection functors* $\Pi_1\colon \mathcal{C} \times \mathcal{D} \to \mathcal{C}$ *and* $\Pi_2\colon \mathcal{C} \times \mathcal{D} \to \mathcal{D}$ *are defined as the first projection and second projection respectively on both arrows and objects. That is,* $\Pi_1(C, D) = C$*,* $\Pi_1(f, g) = f$*,* $\Pi_2(C, D) = D$ *and* $\Pi_2(f, g) = g$*.*

Example 2.15 *The composition of two functors* $F\colon \mathcal{C} \to \mathcal{D}$ *and* $G\colon \mathcal{D} \to \mathcal{E}$ *is written as* GF *and defined by* $GFA = G(FA)$ *and* $GFf = G(Ff)$*.*

A functor from a category $\mathcal{C}$ to itself is called an *endofunctor*. One with a product category as source (like the projection functors) is called a *bifunctor* (as opposed to unary functors or *monofunctors*). By fixing the first argument of a bifunctor $F\colon \mathcal{C} \times \mathcal{D} \to \mathcal{E}$ on a $\mathcal{C}$-object C, one gets the unary functor $F(C, -)$, written F_C, such that $F_C D = F(C, D)$ and $F_C f = F(\mathsf{id}_C, f)$.

2.2.1 Natural Transformations

Natural transformations are structure-preserving maps between functors.

Definition 2.16 Let $\mathcal{C}$ and $\mathcal{D}$ be categories. Given two functors $F\colon \mathcal{C} \to \mathcal{D}$ and $G\colon \mathcal{C} \to \mathcal{D}$, a *natural transformation* $\tau\colon F \Rightarrow G$ is a function that assigns to every $\mathcal{C}$-object A a $\mathcal{D}$-arrow $\tau_A\colon FA \to GA$ such that for any $\mathcal{C}$-arrow $f\colon A \to B$ the following diagram commutes in $\mathcal{D}$.

$$\begin{tikzcd}
FA \arrow[r, "\tau_A"] \arrow[d, "Ff"'] & GA \arrow[d, "Gf"] \\
FB \arrow[r, "\tau_B"'] & GB
\end{tikzcd}$$

□

We will refer to this diagram as the *naturality condition* of τ. The subscripts will often be omitted when the objects involved are clear from the context.

Example 2.17 *For any functor F, the components of the identity natural transformation $\mathsf{id}_F\colon F \Rightarrow F$ are the identity arrows of the objects in the image of F, that is, $\mathsf{id}_A = \mathsf{id}_{FA}$.*

From the viewpoint of programming languages, we will use naturality as a synonym for parametric polymorphism. The relationship between natural transformations and polymorphic functions is formally described in [Wad89] which in turn is derived from [Rey83].

Example 2.18 *Let $\mathcal{C}$ and $\mathcal{D}$ be categories. Let F, G, and H be functors from $\mathcal{C}$ to $\mathcal{D}$. Let $\sigma\colon F \Rightarrow G$ and $\tau\colon G \Rightarrow H$ be natural transformations. Then for each $\mathcal{C}$-arrow $f\colon A \to B$ we can draw the following composite diagram:*

$$\begin{tikzcd}
FA \arrow[r, "\sigma_A"] \arrow[d, "Ff"'] \arrow[dr, phantom, "(I)"] & GA \arrow[r, "\tau_A"] \arrow[d, "Gf"'] \arrow[dr, phantom, "(II)"] & HA \arrow[d, "Hf"] \\
FB \arrow[r, "\sigma_B"'] & GB \arrow[r, "\tau_B"'] & HB
\end{tikzcd}$$

By the naturality condition of σ and τ, both (I) and (II) commutes, so the outer rectangle also commutes. This shows that the composite transformation $(\tau \circ \sigma)\colon F \Rightarrow H$ defined by $(\tau \circ \sigma)_A = \tau_A \circ \sigma_A$ is a natural transformation.

2.3 Product and Sum

In most programming languages, new types can be built by tupling existing datatypes or by taking their disjoint union. In this section, we present their categorical definition and some of their properties.

Definition 2.19 The *product* of two objects A and B in a category $\mathcal{C}$ is given by an object $A \times B$ together with two *projection arrows* $\pi_1 : A \times B \to A$ and $\pi_2 : A \times B \to B$ such that for any object C and pair of arrows $f\colon C \to A$ and $g\colon C \to B$ there is exactly one arrow $\langle f, g\rangle\colon C \to A \times B$ that makes the following diagram commute:

$$
\begin{tikzcd}
A & A \times B \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] & B \\
& C \arrow[ul, "f"] \arrow[u, "\langle f,g\rangle"'] \arrow[ur, "g"'] &
\end{tikzcd}
$$

□

Example 2.20 *The cartesian product*

$$A \times B = \{(a, b) | a \in A, b \in B\}$$

is a categorical product in a category like **Set**, *for instance.*

From the definition of product, the *identity* law and the *fusion* law can be deduced:

$$\langle \pi_1, \pi_2\rangle = \mathsf{id} \qquad\qquad \langle f, g\rangle \circ h = \langle f \circ h, g \circ h\rangle$$

The product can be made into a bifunctor $\times\colon \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ by defining its action on arrows. For $f\colon A \to A'$ and $g\colon B \to B'$,

$$f \times g = A \times B \xrightarrow{\langle f \circ \pi_1, g \circ \pi_2\rangle} A' \times B'$$

Being a functor, it has to satisfy the following conditions:

$$\mathsf{id} \times \mathsf{id} = \mathsf{id} \qquad\qquad (f \times g) \circ (h \times k) = (f \circ h) \times (g \circ k)$$

Other standard properties of the product are:

$$\pi_1 \circ (f \times g) = f \circ \pi_1 \qquad\qquad \pi_2 \circ (f \times g) = g \circ \pi_2 \qquad\qquad (f \times g) \circ \langle h, k\rangle = \langle f \circ h, g \circ k\rangle$$

The first two laws state that π_1 and π_2 are natural transformations. The third one is called the *absorption* law, and it represents a fusion between the product and the pairing of arrows.

Product associativity is defined by the following natural isomorphism:

$$\alpha_{A,B,C} = A \times (B \times C) \xrightarrow{\langle \mathsf{id}_A \times \pi_1, \pi_2 \circ \pi_2\rangle} (A \times B) \times C$$

Products can be generalised to n components in an obvious way. If each pair of objects in $\mathcal{C}$ has a product, one says that $\mathcal{C}$ has *products*.

Let $F, G\colon \mathcal{C} \to \mathcal{D}$ be two functors. If $\mathcal{D}$ has products, then we can define a functor $F \times G$ by defining its action on objects as $(F \times G)A = FA \times GA$ and its action on arrows as $(F \times G)f = Ff \times Gf$.

Example 2.21 *In a functional programming language we could define a datatype for pairs*

$$\textbf{data}\ A \times B \ = \ (A, B)$$

In our category-theoretical model we would represent this datatype with the functor $\times$.

Note that the constructor function is implicit, we do not bother naming it. Since we only care for equality up to an isomorphism, datatypes that are isomorphic like

$$\textbf{data}\ A \otimes B = \ Pair_1\ (A, B)$$

and

$$\textbf{data}\ A \times' B = \ Pair_2\ (A, B)$$

are the same to us and there is no need to distinguish them with different constructor's names.

Definition 2.22 A *coproduct* or *sum* of two objects A and B in a category $\mathcal{C}$ is an object $A + B$, together with two injection arrows $\mathsf{inl}\colon A \to A + B$ and $\mathsf{inr}\colon B \to A + B$ such that for any object C and pair of arrows $f\colon A \to C$ and $g\colon B \to C$ there is exactly one arrow $[f, g]\colon A + B \to C$ making the following diagram commute:

$$\begin{tikzcd}
A \arrow[r, "\mathsf{inl}"] \arrow[dr, "f"'] & A + B \arrow[d, "{[f,g]}"] & B \arrow[l, "\mathsf{inr}"'] \arrow[dl, "g"] \\
& C &
\end{tikzcd}$$

□

Example 2.23 *In* **Set**, *the* disjoint union *of two sets A and B happens to be a coproduct. The disjoint union of sets A and B is the set formed by obtaining a set A' isomorphic to A and a set B' isomorphic to B such that A' and B' are disjoint. The usual way this is done is as follows: let*

$$A' = \{(a, 0) | a \in A\} \qquad \textit{and} \qquad B' = \{(b, 1) | b \in B\}$$

The sets are disjoint since the first is a set of ordered pairs each of whose second entries is 0, *while the second set is a set of ordered pairs each of whose second entries is* 1. *The arrow* $\mathsf{inl}\colon A \to A' \cup B'$ *takes a to $(a, 0)$ and* $\mathsf{inr}\colon B \to A' \cup B'$ *takes b to $(b, 1)$. A case analysis $[f, g]$ is such that:*

$$[f, g](a, 0) = f(a) \qquad\qquad [f, g](b, 1) = g(b)$$

Example 2.24 *In a functional programming language the following could be defined:*

$$\textbf{data}\ \mathsf{Either}(A, B) = \mathsf{Left}\ A \mid \mathsf{Right}\ B$$

In our category-theoretical model we would represent this datatype with the functor $+$.

In a functional language, case analysis is usually written as:

$$\begin{aligned}
[f, g](x) \ = \ \ & \textbf{case}\ x\ \textbf{of} \\
& \quad \mathsf{inl}(a) \to f(a) \\
& \quad \mathsf{inr}(b) \to g(b)
\end{aligned}$$

We can make the sum a bifunctor $+ : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ by defining its action on arrows: For $f\colon A \to A'$ and $g\colon B \to B'$,

$$f + g = A + B \xrightarrow{[\mathsf{inl} \circ f, \mathsf{inr} \circ g]} A' + B'$$

The functoriality conditions in this case are:

$$\mathsf{id} + \mathsf{id} = \mathsf{id} \qquad\qquad (f+g) \circ (h+k) = (f \circ h) + (g \circ k)$$

Some properties of coproducts are:

$$[\mathsf{inl}, \mathsf{inr}] = \mathsf{id}$$
$$h \circ [f, g] = [h \circ f, h \circ g]$$
$$(f+g) \circ \mathsf{inl} = \mathsf{inl} \circ f$$
$$[f, g] \circ (h + k) = [f \circ h, g \circ k]$$
$$(f+g) \circ \mathsf{inr} = \mathsf{inr} \circ g$$

Coproducts can be generalized to n components in the obvious way. Let $F, G\colon \mathcal{C} \to \mathcal{D}$ be two functors. Analogously to products, if $\mathcal{D}$ has sums, we can define a functor $F + G$ as $(F+G)A = FA + GA$ and $(F+G)f = Ff + Gf$.

As an example of the use of the previous properties of products and coproducts we will provide the proof of the following *exchange law*:

Example 2.25

$$[\langle f, h\rangle, \langle g, k\rangle] = \langle [f, g], [h, k]\rangle$$

Proof. *To prove this property we will prove* $\pi_1 \circ [\langle f, h\rangle, \langle g, k\rangle] = [f, g]$ *and* $\pi_2 \circ [\langle f, h\rangle, \langle g, k\rangle] = [h, k]$.

$$\begin{array}{ll} & \pi_1 \circ [\langle f, h\rangle, \langle g, k\rangle] \\ = & \quad \{ \ \text{Coproducts} \ \} \\ & [\pi_1 \circ \langle f, h\rangle, \pi_1 \circ \langle g, k\rangle] \\ = & \quad \{ \ \text{Products} \ \} \\ & [f, g] \end{array} \qquad\qquad \begin{array}{ll} & \pi_2 \circ [\langle f, h\rangle, \langle g, k\rangle] \\ = & \quad \{ \ \text{Coproducts} \ \} \\ & [\pi_2 \circ \langle f, h\rangle, \pi_2 \circ \langle g, k\rangle] \\ = & \quad \{ \ \text{Products} \ \} \\ & [h, k] \end{array}$$

By definition of products our proposition is proved. Equivalently, we could have started from the other side of the equation and proved $\langle [f, g], [h, k]\rangle \circ \mathsf{inl} = \langle f, h\rangle$ *and* $\langle [f, g], [h, k]\rangle \circ \mathsf{inr} = \langle g, k\rangle$. □

2.4 Distributive Categories

Along the thesis we will assume that the underlying category $\mathcal{C}$ is **distributive**. This means that product distributes over coproduct in the following sense: For any A, B and C, the arrow

$$[\mathsf{inl} \times \mathsf{id}_C, \mathsf{inr} \times \mathsf{id}_C] : A \times C + B \times C \to (A + B) \times C$$

is a natural isomorphism with inverse:

$$d_{A,B,C} : (A+B) \times C \to A \times C + B \times C$$

There are plenty of examples of distributive categories, since every cartesian closed category with coproducts is a distributive category. Typical examples are the category **Set** of sets and total functions as well as the category **Cpo** of complete partial orders (not necessarily having a bottom element) and continuous functions.

To manipulate equations with d, a common technique is to use the fact that d is an isomorphism and reverse some d arrow and replace it by d^{-1}.

Example 2.26 *We want to prove $(\pi_1 + \pi_1) \circ d = \pi_1$. The type information is depicted in the following diagram:*

$$\begin{tikzcd}
(A+B) \times C \arrow[dd, "d"'] \arrow[dr, "\pi_1"] & \\
& A \times B \\
A \times C + B \times C \arrow[ur, "\pi_1 + \pi_1"] &
\end{tikzcd}$$

After reversing d the diagram is:

$$\begin{tikzcd}
(A+B) \times C \arrow[dr, "\pi_1"] & \\
& A \times B \\
A \times C + B \times C \arrow[uu, "d^{-1}"] \arrow[ur, "\pi_1 + \pi_1"] &
\end{tikzcd}$$

And now we calculate

$$\begin{array}{ll}
& \pi_1 \circ d^{-1} \\
= & \quad \{ \text{ Definition of } d^{-1} \ \} \\
& \pi_1 \circ [\mathsf{inl} \times \mathsf{id}, \mathsf{inr} \times \mathsf{id}] \\
= & \quad \{ \text{ Coproducts } \} \\
& [\pi_1 \circ (\mathsf{inl} \times \mathsf{id}), \pi_1 \circ (\mathsf{inr} \times \mathsf{id})] \\
= & \quad \{ \text{ Products } \} \\
& [\mathsf{inl} \circ \pi_1, \mathsf{inr} \circ \pi_1] \\
= & \quad \{ \text{ Coproducts } \} \\
& \pi_1 + \pi_1
\end{array}$$

Example 2.27 *We are going to prove that* $[g \times f, h \times f] \circ d = [g, h] \times f$. *If we post-multiply both sides of the equation by* d^{-1} *we obtain*

$$
\begin{array}{ll}
& [g \times f, h \times f] \circ d \circ d^{-1} = ([g,h] \times f) \circ d - 1 \\
= & \quad \{ \text{ Isomorphisms } \} \\
& [g \times f, h \times f] = ([g,h] \times f) \circ d - 1
\end{array}
$$

Now we calculate

$$
\begin{array}{ll}
& ([g,h] \times f) \circ d - 1 \\
= & \quad \{ \text{ Definition of } d^{-1} \} \\
& ([g,h] \times f) \circ [\mathsf{inl} \times \mathsf{id}, \mathsf{inr} \times \mathsf{id}] \\
= & \quad \{ \text{ Coproducts } \} \\
& [([g,h] \times f) \circ (\mathsf{inl} \times \mathsf{id}), ([g,h] \times f) \circ (\mathsf{inr} \times \mathsf{id})] \\
= & \quad \{ \text{ Products } \} \\
& [([g,h] \circ \mathsf{inl}) \times f, ([g,h] \circ \mathsf{inr}) \times f] \\
= & \quad \{ \text{ Coproducts } \} \\
& [g \times f, h \times f]
\end{array}
$$

2.4.1 Conditional operator

In a distributive category it is possible to define a conditional operator. The object of boolean values can be defined as a sum $\mathsf{bool} = 1 + 1$. The truth constants are the inclusions into this sum:

$$1 \xrightarrow{\mathsf{true}} \mathsf{bool} \xleftarrow{\mathsf{false}} 1$$

In a distributive category, the **conditional operator** $\mathsf{cond}(p, f, g) \colon A \to C$ is defined by:

$$
\begin{tikzcd}
A \arrow[rr, "\mathsf{cond}(p,f,g)"] \arrow[d, "\langle p, \mathsf{id} \rangle"'] & & C \\
\mathsf{bool} \times A \arrow[r, "d"'] & 1 \times A + 1 \times A \arrow[r, "\pi_2 + \pi_2"'] & A + A \arrow[u, "{[f,g]}"']
\end{tikzcd}
$$

where $p \colon A \to \mathsf{bool}$ is a predicate, and $f, g \colon A \to C$.

In pointwise style, the application of the conditional operator to a value is usually written as:

$$\mathsf{cond}(p, f, g)\,(a) = \textbf{if } p(a) \textbf{ then } f(a) \textbf{ else } g(a)$$

The conditional operator satisfies the following laws:

$$
\begin{array}{rcl}
h \circ \mathsf{cond}(p, f, g) & = & \mathsf{cond}(p, h \circ f, h \circ g) \\
\mathsf{cond}(p, f, g) \circ h & = & \mathsf{cond}(p \circ h, f \circ h, g \circ h) \\
\mathsf{cond}(p, f, f) & = & f
\end{array}
$$

2.5 Polynomial functors

Polynomial functors are functors built from identities, constants, products and sums. They can be inductively defined by the following grammar:

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

Example 2.28 *The functor F defined by $FX = A + X \times A$ and $Fh = \mathsf{id}_A + h \times \mathsf{id}_A$ is a polynomial functor because:*

$$F = \underline{A} + I \times \underline{A}$$

2.6 Inductive Types

We have showed how to construct certain datatypes using functors. Nevertheless, we have not been able to define a recursively defined datatype yet. In this section we will describe the category-theoretical modelling of *inductive types*, such as finite lists or trees.

2.6.1 Algebras

We will now try to develop an intuition that will help to understand our modelling of datatypes.

Definition 2.29 An *algebra* is a set, called the *carrier* of the algebra, together with a number of operations that return values in that set. □

Some concrete example of algebras are:

$$\begin{array}{lll}
(\mathbb{N}, 0, +), & \text{with} \quad 0 \colon 1 \to \mathbb{N}, & + \colon \mathbb{N} \times \mathbb{N} \to \mathbb{N} \\
(\mathbb{R}, 1, \times), & \text{with} \quad 1 \colon 1 \to \mathbb{R}, & \times \colon \mathbb{R} \times \mathbb{R} \to \mathbb{R} \\
(\mathsf{list}\,(A), \mathsf{nil}, +\!\!+), & \text{with} \quad \mathsf{nil} \colon 1 \to \mathsf{list}\,(A), & +\!\!+ \colon \mathsf{list}\,(A) \times \mathsf{list}\,(A) \to \mathsf{list}\,(A)
\end{array}$$

A recursively defined datatype determines, in a natural way, an algebra. A simple example is the datatype nat defined by

$$\mathbf{data}\ \mathsf{nat} \ = \ \mathsf{zero} \mid \mathsf{succ}\ \mathsf{nat}$$

whose corresponding algebra is

$$(\mathsf{nat}, \mathsf{zero}, \mathsf{succ}), \qquad \text{with} \quad \mathsf{zero} \colon 1 \to \mathsf{nat}, \quad \mathsf{succ} \colon \mathsf{nat} \to \mathsf{nat}$$

Another example is:

$$\mathbf{data}\ \mathsf{Natlist} \ = \ \mathsf{nil} \mid \mathsf{cons}\ \mathsf{nat}\ \mathsf{Natlist}$$

whose corresponding algebra is

$$(\mathsf{Natlist}, \mathsf{nil}, \mathsf{cons}), \qquad \text{with} \quad \mathsf{nil} \colon 1 \to \mathsf{Natlist}, \quad \mathsf{cons} \colon \mathsf{nat} \times \mathsf{Natlist} \to \mathsf{Natlist}$$

These examples illustrate the general idea: An inductive datatype determines an algebra in which

- the carrier of the algebra is the datatype itself, and

- the operations of the algebra are the constructors of the datatype.

Definition 2.30 A *homomorphism* between two algebras is a function between their carrier sets that respect the structure of the algebras. □

For example, the function $\mathsf{exp}\colon \mathbb{N} \to \mathbb{R}$ is a homomorphism with *source algebra* $(\mathbb{N}, 0, +)$ and *target algebra* $(\mathbb{R}, 1, \times)$. Respecting the structure means:

$$\begin{array}{lcl} \mathsf{exp}\ 0 & = & 1 \\ \mathsf{exp}\ (x+y) & = & (\mathsf{exp}\ x) \times (\mathsf{exp}\ y) \end{array}$$

Now, let's consider the dataype nat again.

$$\mathbf{data}\ \mathsf{nat}\ =\ \mathsf{zero} \mid \mathsf{succ\ nat}$$

Since constructors names are not important to us, we can give the following isomorphic definition of the datatype:

$$\mathbf{data}\ \mathsf{nat}\ =\ \mathsf{inl}\ 1 \mid \mathsf{inr\ nat}$$

The choice here could be written as a sum:

$$\mathbf{data}\ \mathsf{nat}\ =\ in_N\ (1 + \mathsf{nat})$$

in which there is only one constructor left, called in_N. The process to obtain this formulation may become clearer by looking at the following figure.

$$\begin{array}{llcccll} \mathsf{zero} & : & 1 & & & \to & \mathsf{nat} \\ \mathsf{succ} & : & & & \mathsf{nat} & \to & \mathsf{nat} \\ \hline \mathsf{in} & : & 1 & + & \mathsf{nat} & \to & \mathsf{nat} \end{array}$$

Now, let's consider a functor $N = 1 + I$ whose action on objects is $NA = 1 + A$ and whose action on arrows is $Nf = id + f$. Then, we have that

$$\mathbf{data}\ \mathsf{nat} = in_N\ (N\ \mathsf{nat}).$$

Apparently, the functor N captures the pattern of inductive information in nat.

If we consider the datatype Natlist, and procceed in the same manner, we obtain

$$\mathbf{data}\ \mathsf{Natlist}\ =\ in_L\ (1 + \mathsf{nat} \times \mathsf{Natlist})$$

The functor $L_{\mathsf{nat}} = 1 + \underline{\mathsf{nat}} \times I$, whose action on objects is $L_{\mathsf{nat}}A = 1 + \mathsf{nat} \times A$ and whose action on arrows is $L_{\mathsf{nat}}f = \mathsf{id} + \mathsf{id}_{\mathsf{nat}} \times f$ captures the pattern information in Natlist.

$$\mathbf{data}\ \mathsf{Natlist} = in_L\ (L_{\mathsf{nat}}\ \mathsf{Natlist})$$

So far, we have seen that an inductive datatype determines an algebra and a functor. We have also seen that we can construct an arrow that packs all the operations in an algebra.

2.6.2 Initial Algebras

We will now formalise the previous intuitions in our categorical framework.

In the following, let $F\colon \mathcal{C} \to \mathcal{C}$ be an endofunctor.

Definition 2.31 An F-*algebra* is a pair (A, h) such that A is a $\mathcal{C}$-object and $h\colon FA \to A$, the object A being the carrier of the algebra and the arrow h packing all the operations in the algebra. □

Example 2.32 *The algebra* $(\mathsf{nat}, +)$ *of the natural numbers and addition is an algebra of the functor* $FA = A \times A$ *and* $Fh = h \times h$*.*

Definition 2.33 An F-*homomorphism* between two algebras (A, h) and (B, h') is an arrow $f\colon A \to B$ between the carriers that commutes with the operations, that is, $f \circ h = h' \circ Ff$.

$$\begin{tikzcd} FA \arrow[r, "Ff"] \arrow[d, "h"'] & FB \arrow[d, "h'"] \\ A \arrow[r, "f"'] & B \end{tikzcd}$$

□

Given an endofunctor $F\colon \mathcal{C} \to \mathcal{C}$ that captures the recursive shape of a datatype, the recursive type will be understood as the least solution to the type equation $X \cong FX$.

Example 2.34 *For the datatype of natural numbers,*

$$\mathsf{nat} = \mathsf{zero} \mid \mathsf{succ}\ \mathsf{nat}$$

the signature is captured by the functor $N = \underline{1} + I$*, that is,*

$$N\,A = 1 + A \qquad\qquad N\,f = \mathsf{id}_1 + f$$

Every N*-algebra is a case analysis* $[h_1, h_2]\colon 1 + A \to A$*, where* $h_1\colon 1 \to A$ *and* $h_2\colon A \to A$*. A homomorphism between two* N*-algebras* $h\colon NA \to A$ *and* $k\colon NB \to B$ *is an arrow* $f\colon A \to B$ *such that:*

$$\begin{tikzcd} 1 \arrow[r, "\mathsf{id}_1"] \arrow[d, "h_1"'] & 1 \arrow[d, "k_1"] \\ A \arrow[r, "f"'] & B \end{tikzcd} \qquad\qquad \begin{tikzcd} A \arrow[r, "f"] \arrow[d, "h_2"'] & B \arrow[d, "k_2"] \\ A \arrow[r, "f"'] & B \end{tikzcd}$$

Example 2.35 *For the datatype of lambda-expressions,*

$$\mathsf{lam} = \mathsf{var}\ V \mid \mathsf{app}\ \mathsf{lam}\ \mathsf{lam} \mid \mathsf{abs}\ V\ \mathsf{lam}$$

the signature is captured by the functor $M = \underline{V} + I \times I + \underline{V} \times I$, *that is,*

$$M\,A = V + A \times A + V \times A \qquad\qquad M\,f = \mathsf{id}_V + f \times f + id_V \times f$$

where V *is the type of variable identifiers.*

Every M*-algebra is a case analysis* $[h_1, h_2, h_3]\colon V + A \times A + V \times A \to A$, *where* $h_1\colon V \to A$, $h_2\colon A \times A \to A$ *and* $h_3\colon V \times A$. *A homomorphism between two* M*-algebras* $h\colon MA \to A$ *and* $k\colon MB \to B$ *is an arrow* $f\colon A \to B$ *such that:*

$$\begin{tikzcd} V \arrow[r, "\mathsf{id}_V"] \arrow[d, "h_1"'] & V \arrow[d, "k_1"] \\ A \arrow[r, "f"'] & B \end{tikzcd} \qquad \begin{tikzcd} A \times A \arrow[r, "f \times f"] \arrow[d, "h_2"'] & B \times B \arrow[d, "k_2"] \\ A \arrow[r, "f"'] & B \end{tikzcd} \qquad \begin{tikzcd} V \times A \arrow[r, "\mathsf{id}_V \times f"] \arrow[d, "h_3"'] & V \times B \arrow[d, "k_3"] \\ A \arrow[r, "f"'] & B \end{tikzcd}$$

Definition 2.36 The *category of* F*-algebras*, denoted by $\mathbf{Alg}(F)$, is formed by the F-algebras as objects and F-homomorphisms as arrows. Composition and identities are inherited from $\mathcal{C}$. □

Definition 2.37 An *initial algebra* is the initial object, if it exists, of a category $\mathbf{Alg}(F)$. □

For many functors, including the polynomial functors of **Set**, this category has an initial object. The initial algebra, if it exists, is the algebra that corresponds to the inductive type whose signature is captured by F. We shall denote the initial algebra by $(\mu F, in_F)$, where the arrow $in_F : F\,\mu F \to \mu F$ encodes the constructors of the inductive type.

2.7 Fold

Initiality permits to associate an operator with each inductive type, which is used to represent functions defined by structural recursion on that type. This operator, usually called *fold* [Bir98] or *catamorphism* [MFP91], is originated by the unique homomorphism that exists between the initial algebra in_F and any other F-algebra $h : FA \to A$. We shall denote it by $\mathsf{fold}_F(h) : \mu F \to A$. Fold is thus the unique arrow that makes the following diagram commute:

$$\begin{tikzcd} F\mu F \arrow[r, "F\mathsf{fold}_F(h)"] \arrow[d, "in_F"'] & FA \arrow[d, "h"] \\ \mu F \arrow[r, "\mathsf{fold}_F(h)"'] & A \end{tikzcd}$$

or equivalently the unique arrow that makes the following equation hold:

$$\mathsf{fold}_F(h) \circ in_F = h \circ F\;\mathsf{fold}_F(h) \tag{2.1}$$

Example 2.38 For the natural numbers, the initial algebra is given by

$$in_N = [\mathsf{zero}, \mathsf{succ}]\colon 1 + \mathsf{nat} \to \mathsf{nat}$$

where $\mathsf{zero}\colon 1 \to \mathsf{nat}$ and $\mathsf{succ}\colon \mathsf{nat} \to \mathsf{nat}$; nat stands for μN. For each algebra $h = [h_1, h_2]$, fold is the unique arrow $f = \mathsf{fold}_N(h) : \mathsf{nat} \to A$ such that

$$\begin{array}{lcl} f\ (\mathsf{zero}) & = & h_1 \\ f\ (\mathsf{succ}(n)) & = & h_2\ (f(n)) \end{array}$$

□

Example 2.39 For the lambda expressions, the initial algebra is given by

$$in_M = [\mathsf{var}, \mathsf{app}, \mathsf{abs}]\colon V + \mathsf{lam} \times \mathsf{lam} + V \times \mathsf{lam} \to \mathsf{lam}$$

where $\mathsf{var}\colon V \to \mathsf{lam}$, $\mathsf{app}\colon \mathsf{lam} \times \mathsf{lam} \to \mathsf{lam}$ and $\mathsf{abs}\colon V \times \mathsf{lam} \to \mathsf{lam}$; lam stands for μM. For each algebra $h = [h_1, h_2, h_3]$, fold is the unique arrow $f = \mathsf{fold}_M(h) : \mathsf{lam} \to A$ such that

$$\begin{array}{lcl} f\ (\mathsf{var}\ v) & = & h_1\ (v) \\ f\ (\mathsf{app}\ (t, u)) & = & h_2\ (f(t), f(u)) \\ f\ (\mathsf{abs}\ (v, t)) & = & h_3\ (v, f(t)) \end{array}$$

□

Lists, trees as well as many other datatypes are usually parameterised. The signature of those datatypes is captured by a bifunctor $F : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$. By fixing the first argument of a bifunctor F one can get a unary functor $F(A, -)$, to be written F_A, such that $F_A\, B = F(A, B)$ and $F_A\, f = F(\mathsf{id}_A, f)$. The functor F_A induces a (polymorphic) inductive type $DA = \mu F_A$, least solution of the equation $X \cong F(A, X)$, with constructors given by the initial algebra $in_{F_A} : F_A(DA) \to DA$.

Example 2.40 (i) Lists with elements over A can be declared by:

$$\mathsf{list}\,(A) = \mathsf{nil} \mid \mathsf{cons}(A \times \mathsf{list}\,(A))$$

We will often write A^* for $\mathsf{list}\,(A)$. The signature of lists is captured by the functor $L_A = \underline{1} + \underline{A} \times I$. The initial algebra is given by $[\mathsf{nil}, \mathsf{cons}] : 1 + A \times A^* \to A^*$. For each algebra $h = [h_1, h_2] : 1 + A \times B \to B$, fold is the unique arrow $f = \mathsf{fold}_{L_A}(h) : A^* \to B$ such that

$$f(\mathsf{nil}) = h_1 \qquad\qquad f(\mathsf{cons}(a, \ell)) = h_2(a, f(\ell))$$

This instance if fold corresponds to the standard foldr operator used in functional programming [Bir98].

(ii) Leaf-labelled binary trees can be declared by

$$\mathsf{btree}\,(A) = \mathsf{leaf}\ A \mid \mathsf{join}\,(\mathsf{btree}\,(A) \times \mathsf{btree}\,(A))$$

Their signature is captured by the functor $B_A = \underline{A} + I \times I$. For each algebra $h = [h_1, h_2] : A + C \times C \to C$, fold is the unique arrow $f = \mathsf{fold}_{B_A}(h) : \mathsf{btree}\,(A) \to C$ such that

$$f(\mathsf{leaf}(a)) = h_1(a) \qquad\qquad f(\mathsf{join}(t, u)) = h_2(f(t), f(u))$$

(iii) Binary trees with information in the nodes can be declared by

$$\mathsf{tree}\,(A) = \mathsf{empty} \mid \mathsf{node}\,(\mathsf{tree}\,(A) \times A \times \mathsf{tree}\,(A))$$

Their signature is captured by the functor $T_A = \underline{1} + I \times \underline{A} \times I$. For each algebra $h = [h_1, h_2] : 1 + C \times A \times C \to C$, fold is the unique arrow $f = \mathsf{fold}_{T_A}(h) : \mathsf{tree}\,(A) \to C$ such that

$$f(\mathsf{empty}) = h_1 \qquad f(\mathsf{node}(t, a, u)) = h_2(f(t), a, f(u))$$

□

2.7.1 Standard Laws for Fold

Fold enjoys many algebraic laws that are useful for program transformation.

The *identity* law states that a fold applied to the constructors of the datatypes gives as a result the identity function.

Theorem 2.41 (Fold Identity)

$$\mathsf{fold}_F(in_F) = \mathsf{id}_{\mu F}$$

The following law is the *fusion* law, a very important law for program calculation. In chapter 1, we already saw an instance of this law for lists. Fold Fusion states that the composition of a fold with an algebra homomorphism is again a fold.

Theorem 2.42 (Fold Fusion)

$$f \circ h = g \circ F f \;\Rightarrow\; f \circ \mathsf{fold}_F(h) = \mathsf{fold}_F(g)$$

Acid rain removes intermediate data structures that are produced by folds whose target algebra is built out of the constructors of the data structure by the application of a *transformer* [Fok96]. A transformer is a polymorphic function $\boldsymbol{T}\colon \forall A.\,(FA \to A) \to (GA \to A)$ that converts F-algebras into G-algebras. Since $\boldsymbol{T}$ has to be polymorphic the following naturality condition has to hold:

For $f : A \to B$, $h : FA \to A$ and $h' : FB \to B$,

$$f \circ h = h' \circ F f \quad \Rightarrow \quad f \circ \boldsymbol{T}(h) = \boldsymbol{T}(h') \circ G f$$

Intuitively, a transformer $\boldsymbol{T}$ may be thought of as a polymorphic function that builds algebras of one class out of algebras of another class.

Theorem 2.43 (Acid Rain: Fold-Fold Fusion)

$$\boldsymbol{T}\colon \forall A.\,(FA \to A) \to (GA \to A) \;\;\Rightarrow\;\; \mathsf{fold}_F(h) \circ \mathsf{fold}_G(\boldsymbol{T}(in_F)) = \mathsf{fold}_G(\boldsymbol{T}(h))$$

2.7.2 Map

Let μF_A be a solution (a fixed point) of the equation $X \cong FX$. Let $DA = \mu F_A$ be a parameterised inductive type induced by a bifunctor F. We have defined the action of D on objetcs. D is a type constructor that can be made into a functor $D\colon \mathcal{C} \to \mathcal{C}$, called a type functor, by defining its action on arrows: For each $f\colon A \to B$,

$$Df = \mathsf{fold}_{F_A}(in_{F_B} \circ F(f, \mathsf{id}_{DB}))\colon DA \to DB$$

It can be proved that this definition makes D a functor.

Consequently, Df is the unique arrow that makes the following diagram commute:

$$\begin{tikzcd}
F_A(DA) \arrow[r, "F_A(Df)"] \arrow[dd, "in_{F_A}"'] & F_A(DB) \arrow[d, "F(f, \mathsf{id})"] \\
 & F_B(DB) \arrow[d, "in_{F_B}"] \\
DA \arrow[r, "Df"'] & DB
\end{tikzcd}$$

The action on arrows of the type functor corresponds to the well-known map function.

Example 2.44 *For lists, the action on arrows is given by*

$$\mathsf{list}\,(f) = \mathsf{fold}_{L_A}([\mathsf{nil}, \mathsf{cons} \circ (f \times \mathsf{id})])$$

$$\begin{array}{lcl} \mathsf{list}\,(f)(\mathsf{nil}) & = & \mathsf{nil} \\ \mathsf{list}\,(f)(\mathsf{cons}(a, \ell)) & = & \mathsf{cons}(f(a), \mathsf{list}\,(f)(\ell)) \end{array}$$

The following is a standard property of type functors.

Theorem 2.45 (Map-Fold Fusion) *For $f : A \to B$ and $h : F_B\, C \to C$,*

$$\mathsf{fold}_{F_B}(h) \circ Df = \mathsf{fold}_{F_A}(h \circ F(f, \mathsf{id}_C))$$

2.8 Regular Functors

Definition 2.46 *Regular functors* are functors built from identities, constants, products, sums and type functors. They can be inductively defined by the following grammar:

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

□

Regular functors capture the signature of *regular datatypes*, which are datatypes whose declarations contain no function spaces and have recursive occurrences with the same arguments from left-hand sides.

Example 2.47 Rose Trees *are trees with multiple branches.*

$$\mathbf{data}\ \mathsf{rose}\,(A) = \mathsf{fork}\ (A \times \mathsf{list}\,(\mathsf{rose}\,(A)))$$

Its signature is captured by the regular functor $R_A = \underline{A} \times \mathsf{list}$. *This means that its action on objects is* $R_A B = A \times \mathsf{list}\,(B)$ *and its action on arrows is* $R_A f = \mathsf{id}_A \times \mathsf{list}\,(f)$. R_A*-algebras are of type* $h\colon A \times \mathsf{list}\,(B) \to B$.

The initial algebra of rose trees is

$$in_{R_A} = \mathsf{fork}\colon A \times \mathsf{list}\,(\mathsf{rose}\,(A)) \to \mathsf{rose}\,(A)$$

For every $h\colon A \times \mathsf{list}\,(B) \to B$, fold *is the unique arrow* $f = \mathsf{fold}_{R_A}(h)\colon \mathsf{rose}\,(A) \to B$ *such that:*

$$f\ (\mathsf{fork}(a, \ell)) = h\ (a, \mathsf{list}\,(f)(\ell))$$

Chapter 3

Introducing *afold*

Accumulations are recursive functions that keep intermediate results in additional parameters, known as *accumulating parameters* or *accumulators*, which are eventually used in later stages of the computation (see e.g. [Bir84, HIT96, BdM97, Gib00]). In this chapter we define a generic operator that permits us to represent structural recursive accumulations on inductive types.

Let us start with an example of an accumulation. Consider the function that computes the sums of the initial segments of a list of numbers:

$$\mathsf{initsums}(\ell) = \mathsf{isums}(\ell, \mathsf{zero})$$

where

$$\mathsf{isums}(\mathsf{nil}, e) \quad = \quad \mathsf{wrap}(e) \tag{3.1}$$

$$\mathsf{isums}(\mathsf{cons}(n, \ell), e) \quad = \quad \mathsf{cons}(e, \mathsf{isums}(\ell, e + n)) \tag{3.2}$$

where $\mathsf{wrap}(x) = \mathsf{cons}(x, \mathsf{nil})$.

To define a function of this kind by structural recursion we have two alternatives. One is to define the function as a higher-order fold of type $\mu F \to [X \to A]$, where X now corresponds to the type of accumulators (see [HIT96]). The other alternative consists of tupling the arguments and defining a function of type $\mu F \times X \to A$. For example, in the particular case of isums this corresponds to a definition of type $\mathsf{nat}^* \times \mathsf{nat} \to \mathsf{nat}^*$ in the style of (3.1) and (3.2). Accumulations defined in this manner cannot be written in terms of the standard fold operator, since fold lacks the possibility of representing functions with multiple arguments.

To express accumulations we will use an operator, called *afold*, which corresponds to a *fold with accumulators*. This operator was first presented in [Par01] as an application of the product comonad. Nevertheless, this chapter is based on the presentation of the afold operator found on [Par02], which does not use the concept of comonad.

If we analyse this function we observe:

- It has as first argument the datatype whose structure we want to follow, in this case, a list. In the nil case, there is no recursion, in the cons case, the recursive call is made on the tail of the input list, .i.e. the recursive instance in the definition of the datatype.

- As second argument it has the accumulator. In this case the accumulator holds the partial sum of the elements that appeared previously in the list.

- In each recursive step the accumulator is updated, adding to it the value at the head of the current list, and passed to the recursive call.
- It uses the value of the accumulator, in this case putting it at the head of the resulting list.

If we abstract from this particular function, we conclude that our generic recursion operator should:

- follow the recursive structure of its first argument datatype;
- pass information to the recursive instances in the second argument, this information is obtained, for each recursive call, as a result of
- calling an accumulation function whose result is the value of the new accumulator;
- possibly use the value of the accumulator.

In the next section this ideas are refined and formalised.

3.1 The *afold* Operator

In the sequel let us fix an object X that now will be regarded as the type of accumulators.

The function that produces the new value of an accumulator will be modeled by an arrow $\overline{\tau}$. Even though the form in which the parameters are modified is something that depends on each specific case, it is possible to state general conditions that an arrow $\overline{\tau}$ must satisfy to be considered proper for accumulation.

Definition 3.1 An arrow $\overline{\tau} : FA \times X \to F(A \times X)$ is said to be **proper for accumulation** if the following conditions hold:

Naturality $\overline{\tau}$ is natural in A: For any $f : A \to B$,

$$\begin{tikzcd} FA \times X \arrow[r, "\overline{\tau}_A"] \arrow[d, "Ff \times \mathsf{id}_X"'] & F(A \times X) \arrow[d, "F(f \times \mathsf{id}_X)"] \\ FB \times X \arrow[r, "\overline{\tau}_B"'] & F(B \times X) \end{tikzcd}$$

Shape and data preservation

$$\begin{tikzcd} & F(A \times X) \arrow[d, "F\,\pi_1"] \\ FA \times X \arrow[ur, "\overline{\tau}_A"] \arrow[r, "\pi_1"'] & FA \end{tikzcd}$$

$\square$

The first condition actually states a restriction to the amount of information that can be used for modifying the accumulators. Indeed, that $\overline{\tau}$ is natural (polymorphic) in A makes accumulations independent from the values in the functor's variable positions —which correspond to the substructures. This means that the only values that are available for accumulation are those contained in the nodes of the data structure, and not the substructures. This is an immediate consequence of the naturality condition. The second condition asserts that $\overline{\tau}$ cannot modify the shape of the structure of type FA nor the data contained in it.

A general form for $\overline{\tau}$ can be given in the following cases:

- When F is a constant functor $\underline{C}$ we have that $\overline{\tau} = \pi_1 : C \times X \to C$.
- When $F = G + H$,

$$\overline{\tau} = (\overline{\tau}' + \overline{\tau}'') \circ d : (GA + HA) \times X \to G(A \times X) + H(A \times X)$$

  for some $\overline{\tau}' : GA \times X \to G(A \times X)$ and $\overline{\tau}'' : HA \times X \to H(A \times X)$. This means that accumulations performed in the variants of a sum are independent from each other. This is a consequence of the hypothesis about distributivity.

Given $\overline{\tau}$ satisfying definition 3.1 we can define an extension of functor F that works on X-actions.

Definition 3.2 For $f\colon A \times X \to B$, the extension for functor F, $\overline{F}f\colon FA \times X \to FB$ is:

$$\overline{F}f \;=\; FA \times X \xrightarrow{\overline{\tau}_A} F(A \times X) \xrightarrow{Ff} FB$$

□

This extension represents the modification of the accumulators in each recursive call. An immediate consequence of the condition of shape and data preservation for $\overline{\tau}$ is that $\overline{F}$ preserves identities, i.e. $\overline{F}\,\pi_1 = \pi_1$. $\overline{F}$ preserves compositions of X-actions only if $\overline{\tau}$ satisfies the equation $\overline{\tau} \circ \langle \overline{\tau}, \pi_2 \rangle = F\langle \mathsf{id}, \pi_2 \rangle \circ \overline{\tau}$, something that we do not expect to hold in general.

Definition 3.3 ([Par01]) An initial algebra in_F is said to be **initial with accumulators** if for each object X, $\overline{\tau} : FA \times X \to F(A \times X)$ proper for accumulation, and $h : FA \times X \to A$, there exists a unique $f : \mu F \times X \to A$ that makes the following diagram commute:

$$\begin{tikzcd}
F\mu F \times X \arrow[r, "\langle \overline{F}f, \pi_2 \rangle"] \arrow[d, "in_F \times \mathsf{id}_X"'] & FA \times X \arrow[d, "h"] \\
\mu F \times X \arrow[r, "f"'] & A
\end{tikzcd}$$

We call **afold** the unique arrow that results from initiality with accumulators and denote it by

$$\mathsf{afold}_F(h, \overline{\tau})\colon \mu F \times X \to A.$$

□

Initiality with accumulators is guaranteed to exist in the presence of exponentials.

Proposition 3.4 *If $\mathcal{C}$ is a cartesian closed category, then every initial algebra is initial with accumulators.*

Therefore, accumulations can be defined in categories like $\mathbf{Set}$ or **Cpo**.

Most of the datatypes we deal with in practice are sums. The following propositions show us how to simplify certain equations into simpler ones that only take into account one addend at a time. Proofs of this propositions will not be given as they are the particular case $\sigma = \mathsf{id}$ of a more general result (proposition 5.6) whose proof can be found in appendix B.

Proposition 3.5 *Let $F = F_1 + F_2$ be a composite functor, $h = [h_1, h_2] \circ d$, $\overline{F}f = Ff \circ \overline{\tau}$, where $\overline{\tau} = (\overline{\tau}_1 + \overline{\tau}_2) \circ d$, and $k = [k_1, k_2] \circ d$. Then*

$$f \circ h = k \circ \langle \overline{F}f, \pi_2 \rangle \quad \Leftrightarrow \quad \begin{cases} f \circ h_1 = k_1 \circ \langle \overline{F}_1 f, \pi_2 \rangle \\ \\ f \circ h_2 = k_2 \circ \langle \overline{F}_2 f, \pi_2 \rangle \end{cases}$$

where $\overline{F}_1 f = Ff \circ \overline{\tau}_1$ and $\overline{F}_2 f = Ff \circ \overline{\tau}_2$

Corollary 3.6 *Let $F = F_1 + F_2$ be a composite functor, $h = [h_1, h_2]$, $\overline{F}f = Ff \circ \overline{\tau}$, where $\overline{\tau} = (\overline{\tau}_1 + \overline{\tau}_2) \circ d$, and $k = [k_1, k_2] \circ d$. Then*

$$f \circ (h \times \mathsf{id}_X) = k \circ \langle \overline{F}f, \pi_2 \rangle \quad \Leftrightarrow \quad \begin{cases} f \circ (h_1 \times id_X) = k_1 \circ \langle \overline{F}_1 f, \pi_2 \rangle \\ \\ f \circ (h_2 \times id_X) = k_2 \circ \langle \overline{F}_2 f, \pi_2 \rangle \end{cases}$$

where $\overline{F}_1 f = Ff \circ \overline{\tau}_1$ and $\overline{F}_2 f = Ff \circ \overline{\tau}_2$

Corollary 3.7 *Let $F = F_1 + F_2$ be a composite functor, $h = [h_1, h_2] \circ d$, and $k = [k_1, k_2] \circ d$. Then*

$$f \circ h = k \circ (Ff \times \mathsf{id}) \quad \Leftrightarrow \quad \begin{cases} f \circ h_1 = k_1 \circ (F_1 f \times \mathsf{id}) \\ \\ f \circ h_2 = k_2 \circ (F_2 f \times \mathsf{id}) \end{cases}$$

3.2 Examples

In this section we present instances of the afold operator for some commonly used datatypes.

Example 3.8 For the natural numbers,

$$\overline{\tau}_A = (\pi_1 + \phi) \circ d$$

where $\phi = \mathsf{id}_A \times \psi : A \times X \to A \times X$, for some $\psi : X \to X$.
Let $h = [h_1, h_2] \circ d : (1 + A) \times X \to A$ and $f = \mathsf{afold}_N(h, \overline{\tau}) : \mathsf{nat} \times X \to A$.
By definition 3.3, f is such that:

$$f \circ (in_N \times \mathsf{id}_X) = h \circ \langle \overline{N}f, \pi_2 \rangle.$$

.

Applying corollary 3.6 we obtain:

$$f \circ (\mathsf{zero} \times \mathsf{id}_X) = h_1 \circ \langle \underline{1} f \circ \pi_1, \pi_2 \rangle$$
$$f \circ (\mathsf{succ} \times \mathsf{id}_X) = h_2 \circ \langle I f \circ \phi, \pi_2 \rangle.$$

In pointwise notation,

$$\begin{array}{lcl} f(\mathsf{zero}, x) & = & h_1(x) \\ f(\mathsf{succ}(n), x) & = & h_2(f(n, \psi(x)), x) \end{array}$$

For example, addition can be defined by

$$\mathsf{add} = \mathsf{afold}_N(h, \overline{\tau})$$

where $h_1 = \pi_2$, $h_2 = \pi_1$ and $\psi = \mathsf{succ}$. That is,

$$\mathsf{add}(\mathsf{zero}, n) = n \qquad \mathsf{add}(\mathsf{succ}(m), n) = \mathsf{add}(m, \mathsf{succ}(n))$$

□

Example 3.9 For lists with elements over A,

$$\overline{\tau}_B = (\pi_1 + \phi) \circ d$$

where $\phi : (A \times B) \times X \to A \times (B \times X)$ is given by $\phi((a, b), x) = (a, (b, \psi(a, x)))$, for some $\psi : A \times X \to X$.

Let $h = [h_1, h_2] \circ d : (1 + A \times C) \times X \to C$ and $f = \mathsf{afold}_{L_A}(h, \overline{\tau}) : A^* \times X \to C$.

By definition 3.3, f is such that:

$$f \circ (in_L \times \mathsf{id}_X) = h \circ \langle \overline{L_A} f, \pi_2 \rangle.$$

Now we can use corollary 3.6 to obtain:

$$f \circ (\mathsf{nil} \times \mathsf{id}_X) = h_1 \circ \langle \underline{1} f \circ \pi_1, \pi_2 \rangle$$
$$f \circ (\mathsf{cons} \times \mathsf{id}_X) = h_2 \circ \langle (\underline{A} f \times I f) \circ \phi, \pi_2 \rangle.$$

In pointwise notation,

$$\begin{array}{lcl} f\ (\mathsf{nil}, x) & = & h_1(x) \\ f\ (\mathsf{cons}(a, \ell), x) & = & h_2(a, f(\ell, \psi(a, x)), x) \end{array}$$

For example, the function isums can be defined by

$$\begin{array}{lcl} \mathsf{isums} & : & \mathsf{nat}^* \times \mathsf{nat} \to \mathsf{nat}^* \\ \mathsf{isums} & = & \mathsf{afold}(h, \overline{\tau}) \end{array}$$

where $h_1(e) = \mathsf{wrap}(e)$, $h_2(n, \ell, e) = \mathsf{cons}(e, \ell)$, and $\psi = \mathsf{add}$.

□

Example 3.10 For leaf-labelled binary trees,

$$\overline{\tau}_C = (\pi_1 + \phi) \circ d$$

where $\phi : (C \times C) \times X \to (C \times X) \times (C \times X)$ is natural in C and preserves shape and data. This means that the c's in the output appear in the same order as in the input. Therefore, $\phi = \langle \pi_1 \times \psi, \pi_2 \times \psi' \rangle$, for some $\psi, \psi' : X \to X$ (i.e. accumulation on left and right branches may differ from each other).

Let $h = [h_1, h_2] \circ d : (A + D \times D) \times X \to D$ and $f = \mathsf{afold}_{B_A}(h, \overline{\tau}) : \mathsf{btree}\,(A) \times X \to D$.

By definition 3.3, f is such that:

$$f \circ (in_B \times \mathsf{id}_X) = h \circ \langle \overline{B}_A f, \pi_2 \rangle.$$

We use corollary 3.6 to obtain:

$$\begin{aligned} f \circ (\mathsf{leaf} \times \mathsf{id}_X) &= h_1 \circ \langle \underline{A} f \circ \pi_1, \pi_2 \rangle \\ f \circ (\mathsf{join} \times \mathsf{id}_X) &= h_2 \circ \langle (If \times If) \circ \phi, \pi_2 \rangle. \end{aligned}$$

In pointwise notation,

$$\begin{aligned} f\ (\mathsf{leaf}(a), x) &= h_1(a, x) \\ f\ (\mathsf{join}(t, u), x) &= h_2(f(t, \psi(x)), f(u, \psi'(x)), x) \end{aligned}$$

For example, the function $\mathsf{rdepth} : \mathsf{btree}\,(A) \to \mathsf{btree}\,(\mathsf{nat})$, which replaces the value at each leaf of a tree by the depth of the leaf, can be defined by

$$\mathsf{rdepth}(t) = \mathsf{down}(t, \mathsf{zero})$$

where

$$\begin{aligned} \mathsf{down} &: \mathsf{btree}\,(A) \times \mathsf{nat} \to \mathsf{btree}\,(\mathsf{nat}) \\ \mathsf{down} &= \mathsf{afold}_{B_A}(h, \overline{\tau}) \end{aligned}$$

with $h_1(a, n) = \mathsf{leaf}(n)$, $h_2(t, u, n) = \mathsf{join}(t, u)$ and $\psi = \psi' = \mathsf{succ}$. That is,

$$\begin{aligned} \mathsf{down}(\mathsf{leaf}(a), n) &= \mathsf{leaf}(n) \\ \mathsf{down}(\mathsf{join}(t, u), n) &= \mathsf{join}(\mathsf{down}(t, n+1), \mathsf{down}(u, n+1)) \end{aligned}$$

□

Example 3.11 For binary trees with information in the nodes,

$$\overline{\tau}_C = (\pi_1 + \phi) \circ d$$

where $\phi : (C \times A \times C) \times X \to (C \times X) \times A \times (C \times X)$ is natural in C and preserves shape and data. Like in the previous case, the c's in the output must appear in the same order as in the input. Therefore, $\phi((c, a, c'), x) = ((c, \psi(a, x)), a, (c', \psi'(a, x)))$, for some $\psi, \psi' : A \times X \to X$ (i.e. accumulation on left and right branches may differ from each other).

Let $h = [h_1, h_2] \circ d : (1 + D \times A \times D) \times X \to D$, $f = \mathsf{afold}_{T_A}(h, \overline{\tau}) : \mathsf{tree}\,(A) \times X \to D$.

By definition 3.3, f is such that:

$$f \circ (in_T \times \mathsf{id}_X) = h \circ \langle \overline{T}_A f, \pi_2 \rangle.$$

We use corollary 3.6 to obtain:

$$f \circ (\mathsf{empty} \times \mathsf{id}_X) = h_1 \circ \langle \underline{1} f \circ \pi_1, \pi_2 \rangle$$
$$f \circ (\mathsf{node} \times \mathsf{id}_X) = h_2 \circ \langle (If \times \underline{A} f \times If) \circ \phi, \pi_2 \rangle.$$

In pointwise notation,

$$\begin{aligned} f\ (\mathsf{empty}, x) &= h_1(x) \\ f\ (\mathsf{node}(t,a,u), x) &= h_2(f(t, \psi(a,x)), a, f(u, \psi'(a,x)), x) \end{aligned}$$

For example, the function asums : $\mathsf{tree}\,(\mathsf{nat}) \to \mathsf{tree}\,(\mathsf{nat})$, which labels each node with the sum of its ancestors, can be defined by

$$\mathsf{asums}(t) = \mathsf{sdown}(t, \mathsf{zero})$$

where

$$\begin{aligned} \mathsf{sdown} &: \mathsf{tree}\,(\mathsf{nat}) \times \mathsf{nat} \to \mathsf{tree}\,(\mathsf{nat}) \\ \mathsf{sdown} &= \mathsf{afold}_{T_{\mathsf{nat}}}(h, \overline{\tau}) \end{aligned}$$

such that $h_1(n) = \mathsf{empty}$, $h_2((t,m,u),n) = \mathsf{node}(t,n,u)$ and $\psi = \psi' = \mathsf{add}$. That is,

$$\begin{aligned} \mathsf{sdown}\ (\mathsf{empty}, n) &= \mathsf{empty} \\ \mathsf{sdown}\ (\mathsf{node}(t,m,u), n) &= \mathsf{node}(\mathsf{sdown}(t, m+n), n, \mathsf{sdown}(u, m+n)) \end{aligned}$$

□

Example 3.12 For rose trees,

$$\overline{\tau}_R\ ((a,\ell), x) = (a, \overline{\tau}^{\mathsf{list}}\ (\ell, \psi(a,x)))$$

where $\psi \colon A \times X \to X$, and $\overline{\tau}^{\mathsf{list}} : \mathsf{list}\,(B) \times X \to \mathsf{list}\,(B \times X)$ is natural in B and preserves shape and data. Therefore,

$$\begin{aligned} \overline{\tau}^{\mathsf{list}}\ (\ell, x) &= \mathsf{list}\,(g)\ \ell \\ & \quad \text{where } g\,a = (a, x) \end{aligned}$$

As we can see in its definition, $\overline{\tau}^{\mathsf{list}}$ distributes the accumulator to each element of the list.
Let $h = [h_1, h_2] \circ d : (A \times \mathsf{list}\,(C)) \times X \to C$, $f = \mathsf{afold}_{R_A}(h, \overline{\tau}) : \mathsf{rose}\,(A) \times X \to C$.
By definition 3.3, f is such that:

$$f \circ (in_R \times \mathsf{id}_X) = h \circ \langle \overline{R}_A f, \pi_2 \rangle.$$

By definition 3.2 we obtain:

$$f \circ (\mathsf{fork} \times \mathsf{id}_X) = h \circ \langle (\mathsf{id}_A \times \mathsf{list}\,(I) f) \circ \overline{\tau}, \pi_2 \rangle.$$

In pointwise notation,

$$f\ (\mathsf{fork}\ (a,r), x) = h\ (a, \mathsf{list}\,(f)\ (\overline{\tau}^{\mathsf{list}}(\ell, \psi(a,x))), x)$$

As an example, the function $\mathsf{rdepth} : \mathsf{rose}\,(A) \to \mathsf{rose}\,(\mathsf{nat})$, which replaces the value at each node of a tree by its depth, can be defined by

$$\mathsf{rdepth}(t) = \mathsf{down}(t, \mathsf{zero})$$

where

$$\begin{aligned} \mathsf{down} &: \mathsf{rose}\,(A) \times \mathsf{nat} \to \mathsf{rose}\,(\mathsf{nat}) \\ \mathsf{down} &= \mathsf{afold}_{R_A}(h, \overline{\tau}) \end{aligned}$$

with $h((a, \ell), n) = \mathsf{fork}(n, \ell)$, $\psi = \mathsf{succ}$. That is,

$$\mathsf{down}(\mathsf{fork}(a, \ell), n) \;=\; \mathsf{fork}(n, \mathsf{list}\,(\mathsf{down})(\overline{\tau}^{\mathsf{list}}\,(\ell, n+1)))$$

□

3.3 Laws for *afold*

The following are some laws for afold.

Theorem 3.13 *For any* $\overline{\tau}$*,*

$$\mathsf{fold}_F(h) \circ \pi_1 = \mathsf{afold}_F(h \circ \pi_1, \overline{\tau})$$

Theorem 3.14 (Afold Identity)

$$\mathsf{afold}_F(in_F \circ \pi_1, \overline{\tau}) = \pi_1$$

Theorem 3.15 (Afold Pure Fusion)

$$f \circ h = h' \circ (Ff \times \mathsf{id}) \;\;\Rightarrow\;\; f \circ \mathsf{afold}_F(h, \overline{\tau}) = \mathsf{afold}_F(h', \overline{\tau})$$

Theorem 3.16 (Acid Rain: Afold-Fold Fusion)

$$\begin{aligned} & T : \forall A.\, (FA \to A) \to (GA \times X \to A) \\ \Rightarrow & \\ & \mathsf{fold}_F(h) \circ \mathsf{afold}_G(\boldsymbol{T}(in_F), \overline{\tau}) = \mathsf{afold}_G(\boldsymbol{T}(h), \overline{\tau}) \end{aligned}$$

Theorem 3.17 (Fold-Afold Fusion) *For every natural transformation* $\kappa : G \Rightarrow F$*,*

$$\begin{aligned} & \kappa \circ \overline{\tau} = \overline{\tau}' \circ (\kappa \times \mathsf{id}) \\ \Rightarrow & \\ & \mathsf{afold}_F(h, \overline{\tau}') \circ (\mathsf{fold}_G(in_F \circ \kappa) \times \mathsf{id}) = \mathsf{afold}_G(h \circ (\kappa \times \mathsf{id}), \overline{\tau}) \end{aligned}$$

Theorem 3.18 (Map-Afold Fusion) *For* $f : A \to B$ *and* $DA = \mu F_A$*,*

$$\begin{aligned} & F(f, \mathsf{id}) \circ \overline{\tau} = \overline{\tau}' \circ (F(f, \mathsf{id}) \times \mathsf{id}) \\ \Rightarrow & \\ & \mathsf{afold}_{F_B}(h, \overline{\tau}') \circ (Df \times \mathsf{id}) = \mathsf{afold}_{F_A}(h \circ (F(f, \mathsf{id}) \times \mathsf{id}), \overline{\tau}) \end{aligned}$$

Theorem 3.19 (Morph-Afold Fusion) *For every $f : X \to X'$,*

$$F(\mathsf{id} \times f) \circ \overline{\tau}_A = \overline{\tau}'_A \circ (\mathsf{id} \times f)$$
$$\Rightarrow$$
$$\mathsf{afold}_F(h, \overline{\tau}') \circ (\mathsf{id} \times f) = \mathsf{afold}_F(h \circ (\mathsf{id} \times f), \overline{\tau})$$

Morph-afold fusion is particularly interesting because it relates two accumulations whose accumulating parameters are of a different type. The premise of that law states a coherence condition that must hold between the accumulators. A proof of these laws can be found in [Par01].

Example 3.20 The height of a leaf-labelled binary tree can be calculated as the maximum of the depths of the leaves in the tree:

$$\mathsf{height} = \mathsf{maxbtree} \circ \mathsf{rdepth}$$

where $\mathsf{maxbtree} = \mathsf{fold}_{B_{\mathsf{nat}}}([\mathsf{id}, \mathsf{max}]) : \mathsf{btree}\,(\mathsf{nat}) \to \mathsf{nat}$ returns the maximum value contained in a tree:

$$\begin{aligned} \mathsf{maxbtree}(\mathsf{leaf}(n)) &= n \\ \mathsf{maxbtree}(\mathsf{join}(t, u)) &= \mathsf{max}(\mathsf{maxbtree}(t), \mathsf{maxbtree}(u)) \end{aligned}$$

where $\mathsf{max}(m, n)$ returns the greater of m and n. Since $\mathsf{rdepth}(t) = \mathsf{down}(t, \mathsf{zero})$, we can write that $\mathsf{height}(t) = \mathsf{aheight}(t, \mathsf{zero})$, where

$$\begin{aligned} \mathsf{aheight} &: \mathsf{btree}\,(A) \times \mathsf{nat} \to \mathsf{nat} \\ \mathsf{aheight} &= \mathsf{maxbtree} \circ \mathsf{down} \end{aligned}$$

This two-pass definition produces an intermediate tree which can be eliminated by fusing the parts. To this end, we first observe that $\mathsf{down} = \mathsf{afold}_{B_A}(\boldsymbol{T}([\mathsf{leaf}, \mathsf{join}]), \overline{\tau})$, being $\boldsymbol{T} : (B_A C \to C) \to (B_A C \times \mathsf{nat} \to C)$ the following transformer:

$$\boldsymbol{T}(k) = [k_1 \circ \pi_2, k_2 \circ \pi_1] \circ d$$

for $k = [k_1, k_2] : A + C \times C \to C$. Therefore, by applying afold-fold fusion we obtain that:

$$\mathsf{aheight} = \mathsf{afold}_{B_A}(\boldsymbol{T}([\mathsf{id}, \mathsf{max}]), \overline{\tau})$$

That is,

$$\begin{aligned} \mathsf{aheight}(\mathsf{leaf}(a), n) &= n \\ \mathsf{aheight}(\mathsf{join}(t, u), n) &= \mathsf{max}(\mathsf{aheight}(t, n+1), \mathsf{aheight}(u, n+1)) \end{aligned}$$

Now, suppose we want to prove the following law:

$$m + \mathsf{aheight}(t, n) = \mathsf{aheight}(t, m + n)$$

In point-free style,

$$(m+) \circ \mathsf{aheight} = \mathsf{aheight} \circ (\mathsf{id} \times (m+))$$

The proof proceeds as follows:

$$\begin{array}{ll}
& \mathsf{aheight} \circ (\mathsf{id} \times (m+)) \\
= & \quad \{ \text{ morph-afold fusion; proof obligation } \} \\
& \mathsf{afold}_{B_A}(\boldsymbol{T}([\mathsf{id}, \mathsf{max}]) \circ (\mathsf{id} \times (m+)), \overline{\tau}) \\
= & \quad \{ \text{ definition of } \boldsymbol{T} \ \} \\
& \mathsf{afold}_{B_A}([\pi_2, \mathsf{max} \circ \pi_1] \circ d \circ (\mathsf{id} \times (m+)), \overline{\tau}) \\
= & \quad \{ \text{ naturality of } d \ \} \\
& \mathsf{afold}_{B_A}([\pi_2, \mathsf{max} \circ \pi_1] \circ (\mathsf{id} \times (m+) + \mathsf{id} \times (m+)) \circ d, \overline{\tau}) \\
= & \quad \{ \text{ coproduct } \} \\
& \mathsf{afold}_{B_A}([(m+) \circ \pi_2, \mathsf{max} \circ \pi_1] \circ d, \overline{\tau}) \\
= & \quad \{ \text{ afold pure-fusion; proof obligation } \} \\
& (m+) \circ \mathsf{aheight}
\end{array}$$

The proof obligation for morph-afold fusion is:

$$\overline{\tau} \circ (\mathsf{id} \times (m+)) = B_A(\mathsf{id} \times (m+)) \circ \overline{\tau}$$

which can be checked by a simple calculation that relies on naturality of d. In the case of pure-fusion the proof obligation is:

$$(m+) \circ [\pi_2, \mathsf{max} \circ \pi_1] \circ d = [(m+) \circ \pi_2, \mathsf{max} \circ \pi_1] \circ d \circ (B_A(m+) \times \mathsf{id})$$

which can be verified by a simple calculation that uses the property: $\mathsf{max} \circ ((m+) \times (m+)) = (m+) \circ \mathsf{max}$. □

Example 3.21 A typical example of accumulation is the linear-time version of reverse:

$$\mathsf{areverse}(\ell) = \mathsf{rev}(\ell, \mathsf{nil})$$

where

$$\begin{array}{rcl}
\mathsf{rev} & : & A^* \times A^* \to A^* \\
\mathsf{rev} & = & \mathsf{afold}_{L_A}([\pi_2, \pi_2 \circ \pi_1] \circ d, \overline{\tau}^{\mathsf{rev}})
\end{array}$$

with $\overline{\tau}^{\mathsf{rev}} = (\pi_1 + \phi^{\mathsf{rev}}) \circ d$ and $\phi^{\mathsf{rev}}((a, \ell), \ell') = (a, (\ell, \mathsf{cons}(a, \ell')))$. That is,

$$\mathsf{rev}(\mathsf{nil}, \ell') = \ell' \qquad\qquad \mathsf{rev}(\mathsf{cons}(a, \ell), \ell') = \mathsf{rev}(\ell, \mathsf{cons}(a, \ell'))$$

Consider also the accumulative version of the function that computes the length of a list:

$$\mathsf{alength}(\ell) = \mathsf{len}(\ell, \mathsf{zero})$$

where

$$\begin{array}{rcl}
\mathsf{len} & : & A^* \times \mathsf{nat} \to \mathsf{nat} \\
\mathsf{len} & = & \mathsf{afold}_{L_A}([\pi_2, \pi_2 \circ \pi_1] \circ d, \overline{\tau}^{\mathsf{len}})
\end{array}$$

with $\overline{\tau}^{\mathsf{len}} = (\pi_1 + \phi^{\mathsf{len}}) \circ d$ and $\phi^{\mathsf{len}}((a,\ell),n) = (a,(\ell,\mathsf{succ}(n)))$. That is,

$$\mathsf{len}(\mathsf{nil},n) = n \qquad\qquad \mathsf{len}(\mathsf{cons}(a,\ell),n) = \mathsf{len}(\ell,\mathsf{succ}(n))$$

Now, suppose we want to prove the following law:

$$\mathsf{length} \circ \mathsf{areverse} = \mathsf{alength}$$

where $\mathsf{length} = \mathsf{fold}_{L_A}([\mathsf{zero},\mathsf{succ} \circ \pi_2])$ is the usual definition of length in terms of fold. This is reduced to proving that:

$$\mathsf{length}(\mathsf{rev}(\ell,\mathsf{nil})) = \mathsf{len}(\ell,\mathsf{zero})$$

which in turn is a particular case of this more general property:

$$\mathsf{length} \circ \mathsf{rev} = \mathsf{len} \circ (\mathsf{id} \times \mathsf{length})$$

The proof proceeds as follows.

$$\begin{array}{cl}
& \mathsf{length} \circ \mathsf{rev} \\
= & \quad \{ \text{ afold pure fusion; proof obligation } \} \\
& \mathsf{afold}_{L_A}([\mathsf{length} \circ \pi_2, \pi_2 \circ \pi_1] \circ d, \overline{\tau}^{\mathsf{rev}}) \\
= & \quad \{ \text{ algebraic manipulation } \} \\
& \mathsf{afold}_{L_A}([\pi_2, \pi_2 \circ \pi_1] \circ d \circ (\mathsf{id} \times \mathsf{length}), \overline{\tau}^{\mathsf{rev}}) \\
= & \quad \{ \text{ morph-afold fusion; proof obligation } \} \\
& \mathsf{len} \circ (\mathsf{id} \times \mathsf{length})
\end{array}$$

The proof obligation for pure fusion is:

$$\mathsf{length} \circ [\pi_2, \pi_2 \circ \pi_1] \circ d = [\mathsf{length} \circ \pi_2, \pi_2 \circ \pi_1] \circ d \circ (L_A\, \mathsf{length} \times \mathsf{id})$$

which can be verified by a simple calculation. In the case of morph-afold fusion the proof obligation is:

$$L_A(\mathsf{id} \times \mathsf{length}) \circ \overline{\tau}^{\mathsf{rev}} = \overline{\tau}^{\mathsf{len}} \circ (\mathsf{id} \times \mathsf{length})$$

which is reduced to proving that

$$(\mathsf{id} \times (\mathsf{id} \times \mathsf{length})) \circ \phi^{\mathsf{rev}} = \phi^{\mathsf{len}} \circ (\mathsf{id} \times \mathsf{length})$$

This can be verified by a simple calculation. □

Finally, we present a law that relates a fold with an accumulative version of it. This law is an adaptation to our setting of a law in [HIT96] that relates a fold with a higher-order fold.

Proposition 3.22 *Let $f : A \times X \to A$ be a function with right identity e, i.e. $f(a,e) = a$, for every a. Then,*

$$f \circ (h \times \mathsf{id}_X) = k \circ \langle \overline{F} f, \pi_2 \rangle \quad \Rightarrow \quad \mathsf{fold}_F(h)(t) = \mathsf{afold}_F(k, \overline{\tau})(t,e)$$

where $\overline{F} f = F f \circ \overline{\tau}$, for $\overline{\tau}$ proper for accumulation.

The following corollary is a simpler formulation of the above theorem for the common case of sum types.

Corollary 3.23

Let $f : A \times X \to A$ be a function with right identity e, i.e. $f(a,e) = a$, for every a. Let $F = F_1 + F_2$ be a composite functor, $h = [h_1, h_2]$, $k = [k_1, k_2] \circ d$, $Ff = Ff \circ \overline{\tau}$ where $\overline{\tau} = (\overline{\tau}_1 + \overline{\tau}_2) \circ d$ proper for accumulation. Then, for every $in_F = [c_1, c_2] \colon F\mu F \to \mu F$

$$\left.\begin{array}{rcl} f \circ (h_1 \times id_X) & = & k_1 \circ \langle \overline{F}_1 f, \pi_2 \rangle \\ \\ f \circ (h_2 \times id_X) & = & k_2 \circ \langle \overline{F}_2 f, \pi_2 \rangle \end{array}\right\} \Rightarrow \left\{\begin{array}{rcl} \mathsf{fold}_F(h) \circ c_1 & = & \mathsf{afold}_F(k, \overline{\tau}) \circ \langle c_1, \underline{e} \rangle \\ \\ \mathsf{fold}_F(h) \circ c_2 & = & \mathsf{afold}_F(k, \overline{\tau}) \circ \langle c_2, \underline{e} \rangle \end{array}\right.$$

Chapter 4

Improving Fusions

In this chapter we present a technique that can be used to improve the resulting fusion in certain cases where laws like pure fusion will not give a satisfactory result.

4.1 An Example of Fusion Improvement

We present an example problem that illustrates the shortcomings of relying on simple fusion for certain fusion problems, and then we proceed to improve the fused function by the application of Morph-Afold Fusion.

4.1.1 The spex Problem

In [Voi03], the following problem was presented:

Given a datatype of lists of A's and B's,

$$\mathbf{data}\ \mathsf{ABlist} = \mathsf{nil} \mid \mathsf{A}\ \mathsf{ABlist} \mid \mathsf{B}\ \mathsf{ABlist}$$

a function $\mathsf{split} \colon (\mathsf{ABlist} \times \mathsf{ABlist}) \to \mathsf{ABlist}$ that orders an ABlist so that all A's come before the B's,

$$\begin{array}{lcl} \mathsf{split}\ (\mathsf{nil}, x) & = & x \\ \mathsf{split}\ ((\mathsf{A}\ u), x) & = & \mathsf{A}\ (\mathsf{split}\ (u, x)) \\ \mathsf{split}\ ((\mathsf{B}\ u), y) & = & \mathsf{split}\ (u, \mathsf{B}\ x) \end{array}$$

and a function $\mathsf{exch} \colon \mathsf{ABlist} \to \mathsf{ABlist}$ that exchanges all A's for B's and viceversa,

$$\begin{array}{lcll} \mathsf{exch\ nil} & = & \mathsf{nil} & (\mathsf{exch}.1) \\ \mathsf{exch}\ (\mathsf{A}\ u) & = & \mathsf{B}\ (\mathsf{exch}\ u) & (\mathsf{exch}.2) \\ \mathsf{exch}\ (\mathsf{B}\ u) & = & \mathsf{A}\ (\mathsf{exch}\ u) & (\mathsf{exch}.3) \end{array}$$

we want to calculate

$$\mathsf{main}\ t = \mathsf{exch}\ (\mathsf{split}\ (t, \mathsf{nil}))$$

The function main is inefficient, since it generates an intermediate data structure. We want to obtain an efficient program $\mathsf{spex} = \mathsf{exch} \circ \mathsf{split}$. Since split is an accumulation we want to express it as an afold for ABlists.

4.1.2 Afold for ABlists

The signature of ABlists is captured by the functor $\mathcal{AB} = \underline{1} + I + I$. The initial algebra is given by $[\mathsf{nil}, \mathsf{A}, \mathsf{B}] \colon 1 + A + A \to A$. Let us call $d_3 \colon (A + B + C) \times X \to A \times X + B \times X + C \times X$ the natural transformation analogous to d for 3 addends. For each algebra $h = [h_1, h_2, h_3] \circ d_3$, afold is the unique arrow $f = \mathsf{afold}_{\mathcal{AB}}(h, \overline{\tau})$ such that

$$\begin{array}{lcl} f\ (\mathsf{nil}, x) & = & h_1(x) \\ f\ (\mathsf{A}\ u, x) & = & h_2(f(u, \psi(x)), x) \\ f\ (\mathsf{B}\ u, x) & = & h_3(f(u, \psi'(x)), x) \end{array}$$

for $\overline{\tau} = \pi_1 + \mathsf{id} \times \psi + \mathsf{id} \times \psi'$.

We can express split as an afold.

$$\begin{array}{lcl} \mathsf{split} & = & \mathsf{afold}_{\mathcal{AB}}(h, \overline{\tau}) \\ & & \text{where } \begin{array}{lcl} h_1 & = & \mathsf{id} \\ h_2 & = & \mathsf{A} \circ \pi_1 \\ h_3 & = & \pi_1 \\ \psi & = & \mathsf{id} \\ \psi' & = & \mathsf{B} \end{array} \end{array}$$

4.1.3 Attempting Pure Fusion

Back to our problem, we will apply the pure fusion law 3.15 to fuse split with exch and obtain spex. After simplifying the antecedent in 3.15 with corollary 3.7 we are left with the following equations:

$$\mathsf{exch} \circ h_1 = h'_1 \tag{4.1}$$

$$\mathsf{exch} \circ h_2 = h'_2 \circ (\mathsf{exch} \times \mathsf{id}) \tag{4.2}$$

$$\mathsf{exch} \circ h_3 = h'_3 \circ (\mathsf{exch} \times \mathsf{id}) \tag{4.3}$$

From 4.1, since $h_1 = \mathsf{id}$ we obtain $h'_1 = \mathsf{exch}$. From 4.2, we calculate

$$\begin{array}{cl} & \mathsf{exch} \circ h_2 \\ = & \quad \{\ \text{Definition of } h_2\ \} \\ & \mathsf{exch} \circ \mathsf{A} \circ \pi_1 \\ = & \quad \{\ \ \mathsf{exch.2}\ \ \} \\ & \mathsf{B} \circ \mathsf{exch} \circ \pi_1 \\ = & \quad \{\ \text{Products}\ \} \\ & \mathsf{B} \circ \pi_1 \circ (\mathsf{exch} \times \mathsf{id}) \\ = & \quad \{\ \ \text{Defining } h'_2 = \mathsf{B} \circ \pi_1\ \ \} \\ & h'_2 \circ (\mathsf{exch} \times \mathsf{id}) \end{array}$$

and obtain $h'_2 = \mathsf{B} \circ \pi_1$. Making an analogous calculation

$$
\begin{array}{ll}
= & \{ \text{ Definition of } h_3 \ \} \\
& \mathsf{exch} \circ \pi_1 \\
= & \{ \text{ Products } \} \\
& \pi_1 \circ (\mathsf{exch} \times \mathsf{id}) \\
= & \{ \text{ Defining } h'_3 = \pi_1 \ \} \\
& h'_3 \circ (\mathsf{exch} \times \mathsf{id})
\end{array}
$$

we obtain $h'_3 = \pi_1$.

We have obtained the accumulation:

$$
\begin{array}{rcl}
\mathsf{spex} & = & \mathsf{afold}_{AB}(h', \overline{\tau}) \\
& & \text{where } \begin{array}[t]{rcl} h'_1 & = & \mathsf{exch} \\ h'_2 & = & \mathsf{B} \circ \pi_1 \\ h'_3 & = & \pi_1 \\ \psi & = & \mathsf{id} \\ \psi' & = & \mathsf{B} \end{array}
\end{array}
$$

Inlining the accumulation gives as a result:

$$
\begin{array}{rcll}
\mathsf{spex}\ (\mathsf{nil}, x) & = & \mathsf{exch}\ x & (\mathsf{spex}.1) \\
\mathsf{spex}\ (\mathsf{A}\ \ell, x) & = & \mathsf{B}\ (\mathsf{spex}\ (\ell, x)) & (\mathsf{spex}.2) \\
\mathsf{spex}\ (\mathsf{B}\ \ell, x) & = & \mathsf{spex}\ (\ell, \mathsf{B}\ x) & (\mathsf{spex}.3)
\end{array}
$$

The fused function spex is more efficient than $\mathsf{exch} \circ \mathsf{split}$, but it is not optimal. While all the A's are being exchanged as the list is being splitted (eq. $(\mathsf{spex}.2)$), all the B's will be exchanged only when spex reaches the end of the list (eq. $(\mathsf{spex}.1)$ and $(\mathsf{spex}.3)$).

We can do better.

4.1.4 Helping fusion

The key observation is that in order to improve the fusion in this function we need to move the exch in $(\mathsf{spex}.1)$ into the accumulation function. We want to obtain h'' and $\overline{\tau}'$ such that

$$\mathsf{afold}_{AB}(h'', \overline{\tau}') = \mathsf{afold}_{AB}(h', \overline{\tau})$$

Looking at the algebraic laws provided by afold, we see that Morph-Afold Fusion (3.19) may be of help. For this h' it is easy to find an h'' such that

$$h' = h'' \circ (\mathsf{id} \times \mathsf{exch}).$$

since none of the recursive cases in the algebra use the accumulator. The above equation can be easily calculated separating it by cases:

$$
\begin{array}{rclcrcl}
\mathsf{exch} & = & h''_1 \circ \mathsf{exch} & \Leftarrow & h''_1 & = & \mathsf{id} \\
\mathsf{B} \circ \pi_1 & = & h''_2 \circ (\mathsf{id} \times \mathsf{exch}) & \Leftarrow & h''_2 & = & \mathsf{B} \circ \pi_1 \\
\pi_1 & = & h''_3 \circ (\mathsf{id} \times \mathsf{exch}) & \Leftarrow & h''_3 & = & \pi_1
\end{array}
$$

What remains is the calculation of $\overline{\tau}'$. The condition in Morph-Afold is:

$$AB\ (\mathsf{id} \times \mathsf{exch}) \circ \overline{\tau} = \overline{\tau}' \circ (\mathsf{id} \times \mathsf{exch}) \tag{4.4}$$

We calculate,

$$
\begin{array}{ll}
& \mathcal{AB}\ (\mathsf{id} \times \mathsf{exch}) \circ \overline{\tau} \\
= & \quad \{ \text{ Functor } \mathcal{AB}, \overline{\tau} \text{ definition } \} \\
& (\mathsf{id} + \mathsf{id} \times \mathsf{exch} + \mathsf{id} \times \mathsf{exch}) \circ (\pi_1 + \mathsf{id} \times \mathsf{id} + \mathsf{id} \times \mathsf{B}) \\
= & \quad \{ \text{ Coproducts } \} \\
& \pi_1 + \mathsf{id} \times (\mathsf{exch} \circ \mathsf{id}) + \mathsf{id} \times (\mathsf{exch} \circ \mathsf{B}) \\
= & \quad \{ \ \mathsf{exch.3} \ \} \\
& \pi_1 + \mathsf{id} \times \mathsf{exch} + \mathsf{id} \times (\mathsf{A} \circ \mathsf{exch}) \\
= & \quad \{ \text{ Products } \} \\
& \pi_1 \circ (\mathsf{id} \times \mathsf{exch}) + \mathsf{id} \circ (\mathsf{id} \times \mathsf{exch}) + (\mathsf{id} \times \mathsf{A}) \circ (\mathsf{id} \times \mathsf{exch}) \\
= & \quad \{ \text{ Coproducts } \} \\
& (\pi_1 + \mathsf{id} + \mathsf{id} \times \mathsf{A}) \circ d_3 \circ (\mathsf{id} \times \mathsf{exch}) \\
= & \quad \{ \text{ Defining } \overline{\tau}' = (\pi_1 + \mathsf{id} + \mathsf{id} \times \mathsf{A}) \circ d_3 \ \} \\
& \overline{\tau}' \circ (\mathsf{id} \times \mathsf{exch})
\end{array}
$$

We have obtained the accumulation

$$
\begin{array}{lll}
\mathsf{spex}' & = & \mathsf{afold}_{\mathcal{AB}}(h'', \overline{\tau}') \\
& & \text{where } \begin{array}{lll} h''_1 & = & \mathsf{id} \\ h''_2 & = & \mathsf{B} \circ \pi_1 \\ h''_3 & = & \pi_1 \\ \psi & = & \mathsf{id} \\ \psi' & = & \mathsf{A} \end{array}
\end{array}
$$

Inlining spex', we obtain:

$$
\begin{array}{lll}
\mathsf{spex}'\ (\mathsf{nil}, x) & = & x \\
\mathsf{spex}'\ (\mathsf{A}\ \ell, x) & = & \mathsf{B}\ (\mathsf{spex}\ (\ell, x)) \\
\mathsf{spex}'\ (\mathsf{B}\ \ell, x) & = & \mathsf{spex}\ (\ell, \mathsf{A}\ x)
\end{array}
$$

where we can observe that spex' is optimal in the sense described before.

We have obtained a function spex' such that

$$\mathsf{spex}' \circ (\mathsf{id} \times \mathsf{exch}) = \mathsf{spex}.$$

Now we calculate from the definition of main

$$
\begin{array}{ll}
& \mathsf{main}\ t \\
= & \quad \{ \text{ Definition of } \mathsf{main}, \text{ Afold Pure Fusion } \} \\
& \mathsf{spex}\ (t, \mathsf{nil}) \\
= & \quad \{ \text{ Morh-Afold Fusion (3.19) } \} \\
& (\mathsf{spex}' \circ (\mathsf{id} \times \mathsf{exch}))\ (t, \mathsf{nil}) \\
= & \quad \{ \text{ (exch.1) } \} \\
& \mathsf{spex}'\ (t, \mathsf{nil})
\end{array}
$$

The final program

$$\mathsf{main}\ t = \mathsf{spex}'(t, \mathsf{nil})$$

does not generate any intermediate structures.

4.2 Foldl

Another example of the kind of functions where pure fusion does not give a satisfactory result is the well-known operator on lists foldl. In this section we will derive an effective fusion law for foldl by simple calculation.

4.2.1 Foldl as an accumulation

The usual definition of the foldl recursion operator in functional languages is:

$$\begin{array}{lcl} \mathsf{foldl}\ (f,e)\ \mathsf{nil} & = & e \\ \mathsf{foldl}\ (f,e)\ (\mathsf{cons}(a,\ell)) & = & \mathsf{foldl}\ (f, f\ (a,e))\ \ell \end{array}$$

We can express this operator as an instance of the afold operator for lists:

$$\begin{array}{lcl} \mathsf{foldl}\ (f,e)\ \ell & = & \mathsf{afold}_L(h,\overline{\tau})\ (\ell,e) \\ & & \text{where } \begin{array}{lcl} h_1 & = & \mathsf{id} \\ h_2 & = & \pi_2 \circ \pi_1 \\ \psi & = & f \end{array} \end{array}$$

where $h = [h_1, h_2] \circ d$, and $\overline{\tau} = (\pi_1 + \phi) \circ d$, for $\phi((a,b),x) = (a,(b,\psi(a,x)))$.

If we have a composition $g \circ \mathsf{foldl}(f,e)$, and apply Afold Pure Fusion (3.15), we obtain the following undesirable result

$$\begin{array}{lcl} (g \circ \mathsf{foldl}\ (f,e))\ \ell & = & \mathsf{afold}_L(h',\overline{\tau})\ (\ell,e) \\ & & \text{where } \begin{array}{lcl} h'_1 & = & g \\ h'_2 & = & \pi_2 \circ \pi_1 \\ \psi & = & f \end{array} \end{array}$$

where, as in the previous example, g will only be applied when the whole input list is consumed.

Again we can solve this by the application of the Morph-Afold Fusion law. After some calculations we obtain:

$$\begin{array}{lcl} (g \circ \mathsf{foldl}\ (f,e))\ \ell & = & \mathsf{afold}_L(h',\overline{\tau}')\ (\ell,g\ e) \\ & & \text{where } \begin{array}{lcl} h'_1 & = & \mathsf{id} \\ h'_2 & = & \pi_2 \circ \pi_1 \end{array} \end{array}$$

which is a foldl. Here $\overline{\tau}' = (\pi_1 + \phi') \circ d$, with $\phi'((a,b),x) = (a,(b,\psi'(a,x)))$. The sanity condition on Morph-Afold Fusion for this case is

$$g \circ \psi = \psi' \circ (\mathsf{id} \times g)$$

4.2.2 Fusion law for foldl

From these results we can derive the following fusion law for foldl.

Proposition 4.1 (Foldl Fusion)

$$
\begin{array}{ll}
 & g \circ f = h \circ (\mathsf{id} \times g) \\
\wedge & \\
 & g\ e = e' \\
\Rightarrow & \\
 & g \circ \mathsf{foldl}\ (f, e) = \mathsf{foldl}\ (h, e')
\end{array}
$$

Chapter 5

Extending *afold*

Afold, as it was defined in the previous chapter, only allows us to express accumulations where the structure of recursion follows exactly the structure of the input datatype. In this section we define an extension to *afold* that is more flexible in the kind of structural recursive accumulations on inductive types that it can express. This extended operator is obtained by relaxing the requirements on accumulator arrows.

Consider the following example:

$$\begin{array}{rcl} \mathsf{subs}\,(\mathsf{nil},\mathsf{y}) & = & (\mathsf{wrap}\circ\mathsf{wrap})\,\mathsf{y} \\ \mathsf{subs}\,(\mathsf{cons}(\mathsf{x},\ell),\mathsf{y}) & = & \mathsf{subs}(\ell,\mathsf{y})\ +\!\!+\ \mathsf{list}\,(\mathsf{y}:)\ \mathsf{subs}(\ell,x) \end{array}$$

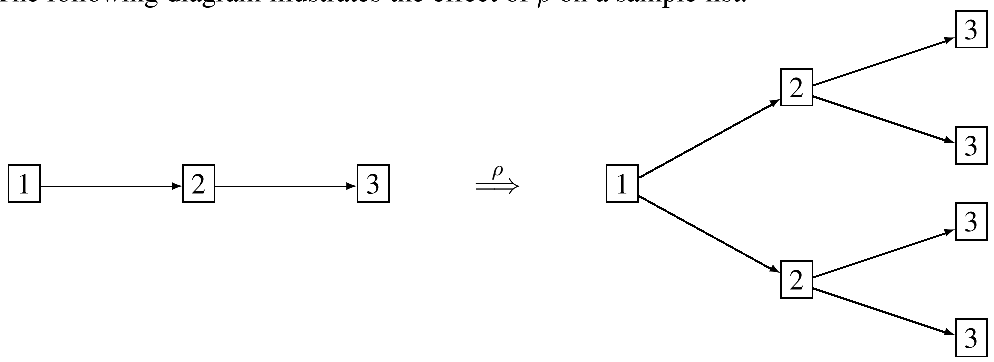
We can see that the input structure is a list. The expression for an afold on lists is:

$$\begin{array}{lcl} f\ (\mathsf{nil},x) & = & h_1(x) \\ f\ (\mathsf{cons}(a,\ell),x) & = & h_2(a,f(\ell,\psi(a,x)),x) \end{array}$$

Now it should be clear that subs cannot be defined using our previous formulation of afold. Being the input structure a list we can only have one recursive call —exactly the same number as the number of recursive instances in the list datatype. However subs has two recursive calls with different accumulation functions. Nevertheless, if we could transform the input into a binary tree using a natural transformation ρ, we would be able to define subs as the composition of ρ with an afold for binary trees. Fortunately, there exists such transformation.

$$\begin{array}{lcl} \rho\,(\mathsf{nil}) & = & \mathsf{empty} \\ \rho\,(\mathsf{cons}(x,xs)) & = & \mathsf{node}(\rho(xs),x,\rho(xs)) \end{array}$$

The following diagram illustrates the effect of ρ on a sample list.



5.1 The extended *afold* operator

We extend afold in order to accomodate the transformation we have just mentioned into the operator. To make the extension, we will relax the shape and data preservation condition on the accumulator arrow $\overline{\tau}$.

Definition 5.1 An arrow $\tau_\sigma : FA \times X \to G(A \times X)$ is said to be **proper for accumulation** if there exists an arrow $\overline{\tau} : GA \times X \to G(A \times X)$ which conforms to definition 3.1 and a natural transformation $\sigma : F \Rightarrow G$ such that $\tau_\sigma = \overline{\tau} \circ (\sigma \times \mathsf{id}_X)$. □

Since both $\overline{\tau}$ and σ are natural in A, we have that τ_σ is also natural in A, so the naturality condition holds. Nevertheless, from the type of τ_σ it should be obvious that it does not preserve shape and data. Given τ_σ satisfying Definition 5.1 we can define another extension of functor G that works on X-actions.

Definition 5.2 For $f : A \times X \to B$, let us define $G_\sigma f : FA \times X \to GB$ to be:

$$G_\sigma f \;=\; FA \times X \xrightarrow{\sigma_A \times id_X} GA \times X \xrightarrow{\overline{\tau}_A} G(A \times X) \xrightarrow{Gf} GB$$

This means that $G_\sigma f = Gf \circ \tau_\sigma$. In terms of our previous functor extension, we have $G_\sigma f = \overline{G} \circ (\sigma \times \mathsf{id}_X)$. □

Definition 5.3 An initial algebra in_F is said to be **initial with accumulators** if for each object X, $\tau_\sigma : FA \times X \to G(A \times X)$ proper for accumulation, and $h : GA \times X \to A$, there exists a unique $f : \mu F \times X \to A$ that makes the following diagram commute:

$$\begin{tikzcd}
F\mu F \times X \arrow[r, "\langle G_\sigma f, \pi_2 \rangle"] \arrow[d, "in_F \times \mathsf{id}_X"'] & GA \times X \arrow[d, "h"] \\
\mu F \times X \arrow[r, "f"'] & A
\end{tikzcd}$$

We call **afold** the unique arrow that results from initiality with accumulators and we denote it by

$$\mathsf{afold}_{F,G}(h, \tau_\sigma) : \mu F \times X \to A.$$

□

Like our previous definition of initiality with accumulators, the extended definition is also guaranteed to exist in the presence of exponentials.

Proposition 5.4 *If $\mathcal{C}$ is a cartesian closed category, then every initial algebra is initial with accumulators.*

Therefore, our extended accumulations can be defined in categories like **Set** or **Cpo**.

Proposition 5.5 *This definition of* afold *is an extension of the previous one.*
Take $\tau_{\mathsf{id}} = \overline{\tau} \circ (\mathsf{id} \times \mathsf{id}_X) = \overline{\tau}$.

$$\mathsf{afold}_F(h, \overline{\tau}) = \mathsf{afold}_{F,F}(h, \tau_{\mathsf{id}})$$

Therefore our previous afold is a particular case of the extended one. We will continue using the previous notation with only one functor in the subscript to refer to this particular case.

Most functors that we deal with in practice are sums. The following proposition shows us how to simplify certain equations into simpler ones that only take into account one addend at a time.

Proposition 5.6 *Let* $G = G_1 + G_2$ *be a composite functor,* $h = [h_1, h_2] \circ d$*,* $G_\sigma f = Gf \circ (\overline{\tau}_1 + \overline{\tau}_2) \circ d \circ (\sigma \times id_X)$*, where* $\sigma = \sigma_1 + \sigma_2$*, and* $k = [k_1, k_2] \circ d$*. Then*

$$f \circ h = k \circ \langle G_\sigma f, \pi_2 \rangle \quad \Leftrightarrow \quad \begin{cases} f \circ h_1 = k_1 \circ \langle G_{1\sigma_1} f, \pi_2 \rangle \\ \\ f \circ h_2 = k_2 \circ \langle G_{2\sigma_2} f, \pi_2 \rangle \end{cases}$$

Corollary 5.7 *Let* $G = G_1 + G_2$ *be a composite functor,* $h = [h_1, h_2]$*,* $G_\sigma f = Gf \circ (\overline{\tau}_1 + \overline{\tau}_2) \circ d \circ (\sigma \times id_X)$*, where* $\sigma = \sigma_1 + \sigma_2$*, and* $k = [k_1, k_2] \circ d$*. Then*

$$f \circ (h \times \mathsf{id}_X) = k \circ \langle G_\sigma f, \pi_2 \rangle \quad \Leftrightarrow \quad \begin{cases} f \circ (h_1 \times id_X) = k_1 \circ \langle G_{1\sigma_1} f, \pi_2 \rangle \\ \\ f \circ (h_2 \times id_X) = k_2 \circ \langle G_{2\sigma_2} f, \pi_2 \rangle \end{cases}$$

Corollary 5.8 *Let* $G = G_1 + G_2$ *be a composite functor,* $h = [h_1, h_2] \circ d$*, and* $k = [k_1, k_2] \circ d$*. Then*

$$f \circ h = k \circ (Gf \times \mathsf{id}) \quad \Leftrightarrow \quad \begin{cases} f \circ h_1 = k_1 \circ (G_1 f \times \mathsf{id}) \\ \\ f \circ h_2 = k_2 \circ (G_2 f \times \mathsf{id}) \end{cases}$$

Even though proposition 5.6 and its corollaries 5.7 and 5.8 were formulated and proved for binary sums, they could be easily extended to n-ary sums. The proofs of all the propositions in this section and its corollaries can be found in appendix B.

Here are some examples:

(i) **Natural numbers**

Let $\sigma\colon 1 + A \to 1 + A \times A$, be natural in A and $\tau_A = (\pi_1 + \phi) \circ d$, where $\phi\colon (A \times A) \times X \to (A \times X) \times (A \times X)$ is natural in A and preserves shape and data.

Therefore, $\phi = \langle \pi_1 \times \psi, \pi_2 \times \psi' \rangle$, for some $\psi, \psi'\colon X \to X$.

Let $h = [h_1, h_2] \circ d\colon (1 + B \times B) \times X \to B$, $H = 1 + I \times I$ and $f = \mathsf{afold}_{N,H}(h, \tau_\sigma)$.

By definition 5.3, f is such that:

$$f \circ (in_N \times \mathsf{id}_X) = h \circ \langle H_\sigma f, \pi_2 \rangle$$

Now we can use corollary 5.7 to obtain:

$$\begin{aligned} f \circ (\mathsf{zero} \times \mathsf{id}_X) &= h_1 \circ \langle 1_{\sigma_1} f, \pi_2 \rangle \\ f \circ (\mathsf{succ} \times \mathsf{id}_X) &= h_2 \circ \langle (I \times I)_{\sigma_2} f, \pi_2 \rangle \end{aligned}$$

In pointwise notation,

$$\begin{array}{lcl} f\ (\mathsf{zero}, x) & = & h_1(x) \\ f\ (\mathsf{succ}(n), x) & = & h_2\ (f(n, \psi(x)), f(n, \psi'(x)), x) \end{array}$$

For example, binomial coefficients can be defined using the addition law

$$\binom{n+1}{m} = \binom{n}{m-1} + \binom{n}{m}$$

and the two base cases

$$\binom{0}{0} = 1 \qquad \binom{0}{1} = 0$$

which can be expressed as

$$\mathsf{comb} = \mathsf{afold}_{N,H}(h, \tau_\sigma)$$

where $h_1 = [\mathsf{succ} \circ \mathsf{zero}, \mathsf{zero}] \circ \pi_2$, $h_2(c_1, c_2, x) = c_1 + c_2$, $\psi = \mathsf{pred}$ and $\psi' = \mathsf{id}$. That is,

$$\begin{array}{lcl} \mathsf{comb}\ (\mathsf{zero}, \mathsf{inl}()) & = & \mathsf{succ} \circ \mathsf{zero} \\ \mathsf{comb}\ (\mathsf{zero}, \mathsf{inr}(y)) & = & \mathsf{zero} \\ \mathsf{comb}\ (\mathsf{succ}(n), x) & = & \mathsf{comb}\ (n, \mathsf{pred}\ x) + \mathsf{comb}\ (n, x). \end{array}$$

(ii) **Lists**

Let $\sigma\colon 1 + A \times B \to 1 + B \times A \times B = \mathsf{id} + \gamma$ with $\gamma(a, b) = (b, a, b)$.

Let $\tau_B = (\pi_1 + \phi) \circ d$, where $\phi\colon (B \times A \times B) \times X \to (B \times X) \times A \times (B \times X)$ is given by $\phi((b, a, b'), x) = ((b, \psi(a, x)), a, (b', \psi'(a, x)))$, for some $\psi, \psi'\colon A \times X \to X$.

Let $h = [h_1, h_2] \circ d\colon (1 + C \times A \times C) \times X \to C$ and $f = \mathsf{afold}_{L_A, T_A}(h, \tau_\sigma)$.

By definition 5.3, f is such that:

$$f \circ (in_L \times \mathsf{id}_X) = h \circ \langle T_\sigma f, \pi_2 \rangle$$

Now we can use corollary 5.7 to obtain:

$$\begin{array}{l} f \circ (\mathsf{nil} \times \mathsf{id}_X) = h_1 \circ \langle 1_{\sigma_1} f, \pi_2 \rangle \\ f \circ (\mathsf{cons} \times \mathsf{id}_X) = h_2 \circ \langle (I \times \underline{A} \times I)_{\sigma_2} f, \pi_2 \rangle \end{array}$$

In pointwise notation,

$$\begin{array}{lcl} f\ (\mathsf{nil}, x) & = & h_1(x) \\ f\ (\mathsf{cons}(a, \ell), x) & = & h_2\ (f(\ell, \psi(a, x)), a, f(\ell, \psi'(a, x)), x) \end{array}$$

5.2 Laws for the extended *afold*

We are now going to show some laws about afold. The proof of these theorems can be found in appendix B. Most of these theorems are the extended counterpart of the afold laws stated in chapter 3.

In the sequel we take $\tau_\sigma = \overline{\tau} \circ (\sigma \times \mathsf{id}_X)$ and $\tau'_\sigma = \overline{\tau}' \circ (\sigma \times \mathsf{id}_X)$

Theorem 5.9 (Afold Factorization)
Let $\overline{\tau}$ be proper for accumulation and $\sigma : F \Rightarrow G$, then

$$\mathsf{afold}_{F,G}(h, \tau_\sigma) = \mathsf{afold}_G(h, \overline{\tau}) \circ (\mathsf{fold}_F(in_G \circ \sigma) \times id_X)$$

where $\tau_\sigma = \overline{\tau} \circ (\sigma \times id_X)$.

Afold Factorization tell us that an extended afold can be factorized in the composition of a fold and an afold.

Theorem 5.10 (Afold Transformation Shift)
For every natural transformation $\kappa : F \Rightarrow G$, $\sigma : F \Rightarrow F$,

$$\begin{array}{ll} & \kappa \circ \tau_\sigma = \tau'_\sigma \circ (\kappa \times \mathsf{id}) \\ \Rightarrow & \\ & \mathsf{afold}_{F,G}(h, \tau'_{\sigma\circ\kappa}) = \mathsf{afold}_{F,F}(h \circ (\kappa \times \mathsf{id}), \tau_\sigma) \end{array}$$

Afold Transformation Shift tell us that under certain conditions we can move a natural transformation from the accumulation function to the algebra.

Theorem 5.11
For any $\overline{\tau}$,

$$\mathsf{fold}_F(h) \circ \pi_1 = \mathsf{afold}_{F,F}(h \circ \pi_1, \overline{\tau})$$

Theorem 5.12 (Afold Identity)

$$\mathsf{afold}_{F,F}(in_F \circ \pi_1, \overline{\tau}) = \pi_1$$

Theorem 5.13 (Afold Pure Fusion)

$$f \circ h = h' \circ (Gf \times \mathsf{id}) \quad \Rightarrow \quad f \circ \mathsf{afold}_{F,G}(h, \tau_\sigma) = \mathsf{afold}_{F,G}(h', \tau_\sigma)$$

To simplify the condition on Afold Pure Fusion the corollary 5.8 might come in handy.

Theorem 5.14 (Acid Rain: Afold-Fold Fusion)

$$\begin{array}{ll} & T : \forall A.\, (HA \to A) \to (GA \times X \to A) \\ \Rightarrow & \\ & \mathsf{fold}_H(h) \circ \mathsf{afold}_{F,G}(\boldsymbol{T}(in_H), \tau_\sigma) = \mathsf{afold}_{F,G}(\boldsymbol{T}(h), \tau_\sigma) \end{array}$$

Theorem 5.15 (Fold-Afold Transformation Fusion)
For every natural transformation $\kappa : H \Rightarrow F$,

$$\mathsf{afold}_{F,G}(h, \tau'_\sigma) \circ (\mathsf{fold}_H(in_F \circ \kappa) \times \mathsf{id}) = \mathsf{afold}_{H,G}(h, \tau_{\sigma\circ\kappa})$$

Corollary 5.16 (Fold-Afold Fusion)
If $\kappa : G \Rightarrow F$ and $\sigma : F \Rightarrow F$,

$$\begin{array}{ll} & \kappa \circ \tau_\sigma = \tau'_\sigma \circ (\kappa \times \mathsf{id}) \\ \Rightarrow & \\ & \mathsf{afold}_{F,F}(h, \tau'_\sigma) \circ (\mathsf{fold}_G(in_F \circ \kappa) \times \mathsf{id}) = \mathsf{afold}_{F,F}(h \circ (\kappa \times \mathsf{id}), \tau_\sigma) \end{array}$$

When fusing a fold with an afold we may choose between the above theorem and its corollary, depending on what we want to do. Theorem 5.15 fuses the fold into the accumulation function while corollary 5.16 fuses it into the algebra.

Theorem 5.17 (Map-Afold Fusion)
For $f : A \to B$ and $DA = \mu F_A$,

$$\begin{array}{ll} & G(f, \mathsf{id}) \circ \overline{\tau} = \overline{\tau}' \circ (G(f, \mathsf{id}) \times \mathsf{id}) \\ \Rightarrow & \\ & \mathsf{afold}_{F_B,G_B}(h, \tau'_\sigma) \circ (Df \times \mathsf{id}) = \mathsf{afold}_{F_A,G_A}(h \circ (G(f, \mathsf{id}) \times \mathsf{id}), \tau_\sigma) \end{array}$$

Theorem 5.18 (Morph-Afold Fusion)
For every $f : X \to X'$,

$$\begin{array}{ll} & G(\mathsf{id} \times f) \circ \overline{\tau} = \overline{\tau}' \circ (\mathsf{id} \times f) \\ \Rightarrow & \\ & \mathsf{afold}_{F,G}(h, \tau'_\sigma) \circ (\mathsf{id} \times f) = \mathsf{afold}_{F,G}(h \circ (\mathsf{id} \times f), \tau_\sigma) \end{array}$$

The following law allows us to calculate an accumulation from a fold.

Proposition 5.19
Let $f : A \times X \to A$ be a function with right identity e, i.e. $f(a, e) = a$, for every a. Then,

$$f \circ (h \times \mathsf{id}_X) = k \circ \langle G_\sigma f, \pi_2 \rangle \quad \Rightarrow \quad \mathsf{fold}_F(h)(t) = \mathsf{afold}_{F,G}(k, \tau_\sigma)(t, e)$$

where $G_\sigma f = Gf \circ \tau_\sigma$, for τ_σ proper for accumulation.

Corollary 5.20
Let $f : A \times X \to A$ be a function with right identity e, i.e. $f(a, e) = a$, for every a. Let $G = G_1 + G_2$ and $F = F_1 + F_2$ be composite functors, $h = [h_1, h_2]$, $G_\sigma f = Gf \circ (\overline{\tau}_1 + \overline{\tau}_2) \circ d \circ (\sigma \times id_X)$, where $\sigma = \sigma_1 + \sigma_2$, and $k = [k_1, k_2] \circ d$. Then, for every $in_F = [c_1, c_2] \colon F\mu F \to \mu F$

$$\left.\begin{array}{rcl} f \circ (h_1 \times id_X) & = & k_1 \circ \langle G_{1\sigma_1} f, \pi_2 \rangle \\ \\ f \circ (h_2 \times id_X) & = & k_2 \circ \langle G_{2\sigma_2} f, \pi_2 \rangle \end{array}\right\} \Rightarrow \left\{\begin{array}{rcl} \mathsf{fold}_F(h) \circ c_1 & = & \mathsf{afold}_{F,G}(k, \tau_\sigma) \circ \langle c_1, \underline{e} \rangle \\ \\ \mathsf{fold}_F(h) \circ c_2 & = & \mathsf{afold}_{F,G}(k, \tau_\sigma) \circ \langle c_2, \underline{e} \rangle \end{array}\right.$$

Chapter 6

Case Study

In this chapter we will apply the results of chapter 5 to calculate an efficient program for the Path Sequence Problem [Bir84, HIT96], starting with a simple specification of the problem.

6.1 Specification

The problem is to determine the length of the longest subsequence of a given sequence of vertices that forms a connected path in a given directed graph G. For simplicity we suppose that G is presented through a predicate arc so that $\mathsf{arc}\ a\ b$ is $true$ just in the case that (a, b) is an arc of G from vertex a to vertex b.

As illustration, consider the graph of Figure 6.1 and the sequence $x = CABDACDEBE$. The length of the longest path sequence is 5, corresponding to $CDABE$ and $ABCBE$.

The specification of the problem is:

$$\mathsf{llp} \ = \ \mathsf{maximum} \circ \mathsf{list}\,(\mathsf{length}) \circ (\mathsf{filter\ path}) \circ \mathsf{subs}$$

where path is a predicate that is true if the given sequence is a path in the graph.

$$\begin{array}{lcll}
\mathsf{path}\ (\mathsf{nil}) & = & \mathsf{true} & (\mathsf{path}.1)\\
\mathsf{path}\ (\mathsf{cons}(x, \mathsf{nil}) & = & \mathsf{true} & (\mathsf{path}.2)\\
\mathsf{path}\ (\mathsf{cons}(x_1, \mathsf{cons}(x_2, xs))) & = & \mathsf{arc}\ x_1\ x_2\ \wedge\ \mathsf{path}(\mathsf{cons}(x_2, xs)) & (\mathsf{path}.3)
\end{array}$$

and subs is a function that generates all the subsequences of a given sequence.

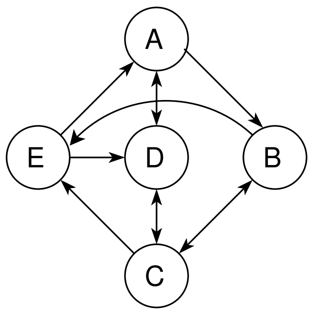


Figure 6.1: An example graph

$$\begin{aligned}
\mathsf{subs}\,(\mathsf{nil}) &= \mathsf{wrap} \circ \mathsf{nil} && (\mathsf{subs}.1)\\
\mathsf{subs}\,(\mathsf{cons}(x, xs)) &= \mathsf{subs}\ xs\ +\!\!+\ \mathsf{list}\,(x{:})\ (\mathsf{subs}\ xs) && (\mathsf{subs}.2)
\end{aligned}$$

where $(x{:})$ denotes the function that puts x at the head of a given list.

This means that the length of the longest path sequence is defined to be the maximum of the lengths of all subsequences of the input that satisfy the path predicate.

6.2 Program Derivation

The specification just given does describe an algorithm to solve the problem, though not an efficient one. It requires us to generate the set of all subsequences of the input, of which there are 2^n if the length of the input is n, test each one for the path property, compute the length of each subsequence that passes the test, and finally extract the maximum. Clearly, the algorithm is exponential in the length of the given sequence.

We will calculate an efficient algorithm from this specification by fusing all the parts. We will start deriving an accumulation for subs in order to be able to fuse it with filter path. Then, we will manipulate the accumulation using the afold theorems.

6.2.1 An accumulation for subs

To derive an accumulation for subs we express it as a fold:

$$\begin{aligned}
\mathsf{subs} \quad = \quad & \mathsf{fold}_L([h_1, h_2])\\
& \text{where } h_1 = wrap \circ \mathsf{nil}\\
& \qquad\quad\, h_2(x, p) = p +\!\!+ \mathsf{list}\,(x{:})\ p.
\end{aligned}$$

Now we can use Proposition 5.19. To obtain an afold we need to find f, k and G_σ such that $f \circ (h \times \mathsf{id}_X) = k \circ \langle G_\sigma f, \pi_2\rangle$. Applying corollary 5.7, we simplify this condition into the following equations:

$$f \circ (h_1 \times id_X) \quad = \quad k_1 \circ \langle G_1 f \circ \tau_{\sigma_1}, \pi_2\rangle \tag{6.1}$$

$$f \circ (h_2 \times id_X) \quad = \quad k_2 \circ \langle G_2 f \circ \tau_{\sigma_2}, \pi_2\rangle \tag{6.2}$$

Now we have to think where and how we want to accumulate. We express this with the following invariant.

$$\mathsf{asubs}\ (xs, y) = \mathsf{list}\,(y{:})\ (\mathsf{subs}\ xs)$$

where asubs is the accumulative version of subs.

Looking at the equations 6.1 and 6.2 and the invariant suggests that f be $f(r, y) = \mathsf{list}\,(y{:})\ r$. Proposition 5.19 requires us to have a right identity e for f, but f has no such right identity. We will solve this problem by lifting f to f_e, where for any $h : A \times B \to A$, and $d_r : A \times (B + C) \to A \times B + A \times C$, the natural transformation that distributes to the right, we have that $h_e : A \times (1 + B) \to A = (\pi_1 + h) \circ d_r$, effectively creating a *virtual* right identity. An analogous lifting can be used to obtain a left identity for a function $g : B \times A \to A$; we will use the same notation for both liftings. The reader should be able to infer from the type of the function being lifted and the context which one is meant.

The lifted invariant is

$$\mathsf{asubs}_e\ (xs, y) = \mathsf{list}\,(y{:}_e)\ (\mathsf{subs\ xs})$$

and the lifted equations now are

$$\begin{aligned} f_e \circ (h_1 \times id_X) &= k_1 \circ \langle G_1 f_e \circ \tau_{\sigma_1}, \pi_2 \rangle && (6.3)\\ f_e \circ (h_2 \times id_X) &= k_2 \circ \langle G_2 f_e \circ \tau_{\sigma_2}, \pi_2 \rangle. && (6.4) \end{aligned}$$

The lifted version of f, i.e. f_e, does have a right identity $\mathsf{inl}()$.

$$f_e(r, y) = \mathsf{list}\,(y :_e)\, r$$

We resume the derivation using the lifted equations. In equation 6.3, we assume $\tau_{\sigma_1} = \pi_1$ and $G_1 = \underline{\mathbf{1}}$, and obtain:

$$(\mathsf{list}\,(y :_e) \circ (\mathsf{wrap} \circ \mathsf{nil}))\; () = k_1(y) \quad \Rightarrow \quad k_1 = [\mathsf{wrap} \circ \mathsf{nil}, \mathsf{wrap} \circ \mathsf{wrap}]$$

Analyzing the LHS of equation 6.4,

$$\begin{aligned} & f\ (h_2(x, p), y) \\ = & \qquad \{\ \text{Definition of } f \text{ and } h_2\ \} \\ & \mathsf{list}\,(y :_e)\ (p \mathbin{+\!\!+} \mathsf{list}\,(x :)\ p) \\ = & \qquad \{\ \text{Naturality of } \mathbin{+\!\!+}\ \} \\ & \mathsf{list}\,(y :_e)\ p \mathbin{+\!\!+} \mathsf{list}\,(y :_e)\ (\mathsf{list}\,(x :)\ p) \\ = & \qquad \{\ \text{Definition of } f\ \} \\ & f_e(p, y) \mathbin{+\!\!+} \mathsf{list}\,(y :_e)\ (f_e(p, \mathsf{inr}\ x)) \end{aligned}$$

we can observe that there are two occurrences of the recursive parameter with two different values of the accumulating parameter. This suggests that σ is a natural transformation of type $\underline{A} \times I \Rightarrow I \times \underline{A} \times I$. This means that $G_2 = I \times \underline{A} \times I$, and $\overline{\tau}((c, a, c'), x) = ((c, \psi(a, x)), a, (c', \psi'(a, x)))$. We take $\sigma(x, p) = (p, x, p)$. After expanding these definitions in equation 6.4, we have:

$$f_e(p, y) \mathbin{+\!\!+} \mathsf{list}\,(y :_e)\ (f_e(p, \mathsf{inr}\ x)) = k_2(f_e(p, \psi(x, y)), x, f_e(p, \psi'(x, y)), y)$$

Taking $\psi = \pi_2$, $\psi' = \mathsf{inr} \circ \pi_1$, we obtain

$$\begin{aligned} & f_e(p, y) \mathbin{+\!\!+} \mathsf{list}\,(y :_e)\ (f_e(p, \mathsf{inr}\ x)) = k_2\ (f_e(p, y), x, f_e(p, \mathsf{inr}\ x), y) \\ \Leftarrow & \qquad \{\ \text{Generalising } f_e(p, y) \text{ to } p_y \text{ and } f_e(p, \mathsf{inr}\ x) \text{ to } p_x\ \} \\ & p_y \mathbin{+\!\!+} \mathsf{list}\,(y :_e)\ p_x = k_2\ (p_y, x, p_x, y) \end{aligned}$$

Now that we have found k_1 and k_2, proposition 5.19 tells us that the accumulation we want is

$$\begin{aligned} \mathsf{asubs}_e\ xs \ =\ & \mathsf{afold}_{L,T}([k_1, k_2] \circ d, \tau_\sigma)\ (xs, e) \\ & \quad \mathit{where}\ k_1 && = \mathsf{wrap} \circ \mathsf{wrap}_e \\ & \qquad\quad k_2((p_y, x, p_x), y) && = p_y \mathbin{+\!\!+} \mathsf{list}\,(y :_e)\ p_x \\ & \qquad\quad \tau_\sigma(x, p, y) && = ((p, y), x, (p, \mathsf{inr}\ x), y) \end{aligned}$$

Inlining the above function gives as a result:

$$\begin{aligned} \mathsf{asubs}_e\ (\mathsf{nil}, \mathsf{inl}()) &= (\mathsf{wrap} \circ \mathsf{nil})\ () \\ \mathsf{asubs}_e\ (\mathsf{nil}, \mathsf{inr}(z)) &= (\mathsf{wrap} \circ \mathsf{wrap})\ z \\ \mathsf{asubs}_e\ (\mathsf{cons}\,(x, xs), \mathsf{inl}()) &= \mathsf{subs}\,(xs, \mathsf{inl}()) \mathbin{+\!\!+} \mathsf{subs}\,(xs, \mathsf{inr}\ x) \\ \mathsf{asubs}_e\ (\mathsf{cons}\,(x, xs), \mathsf{inr}(z)) &= \mathsf{subs}\,(xs, \mathsf{inr}(z)) \mathbin{+\!\!+} \mathsf{list}\,(z :)\ (\mathsf{subs}\,(xs, \mathsf{inr}\ x) \end{aligned}$$

6.2.2 Fusing $(\mathsf{filter\ path}) \circ \mathsf{asubs}_e$

We will now use Theorem 5.13 to fuse $\mathsf{filter\ path}$ with asubs_e. According to Theorem 5.13, we have to find k' such that $f \circ k = k' \circ (T\ f \times id)$ where $f = \mathsf{filter\ path}$. Putting corollary 5.8 into use gives us the following equations:

$$(\mathsf{filter\ path}) \circ k_1 \quad = \quad k_1' \tag{6.5}$$
$$(\mathsf{filter\ path}) \circ k_2 \quad = \quad k_2' \circ ((\mathsf{filter\ path} \times id \times \mathsf{filter\ path}) \times id_X) \tag{6.6}$$

In equation 6.5, since $k_1 = \mathsf{wrap} \circ \mathsf{wrap}_e$ and path is true for singletons and empty lists, we have that $k_1' = k_1$.

Next, we will derive k_2'. Let's recall that $k_2((p_y, x, p_x), y) = p_y \ +\!\!+\ \mathsf{list}\,(y :_e)\ p_x$. The LHS of equation 6.6 is:

$$
\begin{array}{ll}
& ((\mathsf{filter\ path}) \circ k_2)\ ((p_y, x, p_x), y) \\
= & \quad \{\ \text{Definition of } k_2\ \} \\
& (\mathsf{filter\ path})\ (p_y \ +\!\!+\ \mathsf{list}\,(y :_e)\ p_x) \\
= & \quad \{\ \text{Proposition A.2}\ \} \\
& (\mathsf{filter\ path})\ p_y \ +\!\!+\ (\mathsf{filter\ path})\ (\mathsf{list}\,(y :_e)\ p_x)
\end{array}
$$

We would like to express $(\mathsf{filter\ path})\ (\mathsf{list}\,(y :_e)\ p_x)$ —the expression to the right of the append operation— in terms of $\mathsf{filter\ path}\ p_x$.

$$
\begin{array}{ll}
& (\mathsf{filter\ path}) \circ (\mathsf{list}\,(y :_e)) \\
= & \quad \{\ \text{Proposition A.1}\ \} \\
& \mathsf{list}\,(y :_e)\ (\mathsf{filter}\ (\mathsf{path} \circ (y :_e)) \\
= & \quad \{\ \text{Property 6.7, see figure 6.2}\ \} \\
& \mathsf{list}\,(y :_e) \circ \mathsf{filter}\ (\wedge \circ \langle \mathsf{arc}'\ y, \mathsf{path} \rangle) \\
= & \quad \{\ \text{Proposition A.3}\ \} \\
& \mathsf{list}\,(y :_e) \circ \mathsf{filter}\ (\mathsf{arc}'\ y) \circ (\mathsf{filter\ path})
\end{array}
$$

We continue this derivation using pointwise notation.

$$
\begin{array}{ll}
& (\mathsf{list}\,(y :_e) \circ \mathsf{filter}\ (\mathsf{arc}'\ y))\ (\mathsf{filter\ path}\ p_x) \\
= & \quad \{\ p_x = \mathsf{list}\,(x :)\ p_x'\ \} \\
& (\mathsf{list}\,(y :_e) \circ \mathsf{filter}\ (\mathsf{arc}'\ y))\ (\mathsf{filter\ path}\ (\mathsf{list}\,(x :)\ p_x')) \\
= & \quad \{\ \text{Proposition A.1}\ \} \\
& (\mathsf{list}\,(y :_e) \circ \mathsf{filter}\ (\mathsf{arc}'\ y) \circ \mathsf{list}\,(x :))\ (\mathsf{filter}\ (\mathsf{path} \circ (x :))\ p_x') \\
= & \quad \{\ \text{Proposition A.1, Type Functor}\ \} \\
& (\mathsf{list}\,(y :_e x :) \circ \mathsf{filter}\ (\mathsf{arc}'\ y \circ (x :)))\ (\mathsf{filter}\ (\mathsf{path} \circ (x :))\ p_x')
\end{array}
$$

Looking at the second equation of arc', we observe that

$$\mathsf{arc}'\ y \circ (x :) = \lambda l.\mathbf{case}\ y\ \mathbf{of}\ \mathsf{inl}() \to \mathsf{true};\ \mathsf{inr}(z) \to \mathsf{arc}\ z\ x$$

To make the notation lighter and the calculations easier we are now going to consider the two cases of y separately.

We need a new predicate arc' such that the following property holds:

$$(\mathsf{path} \circ (y :_e))\ p = (\mathsf{arc}'\ y\ p) \wedge \mathsf{path}\ p \tag{6.7}$$

We calculate:

$$\begin{array}{ll} & \mathsf{true} \\ = & \quad \{ \text{ path.1 and path.2 } \} \\ & (\mathsf{path} \circ (y :_e))\ \mathsf{nil} \\ = & \quad \{ \text{ Property 6.7 } \} \\ & (\mathsf{arc}'\ y\ \mathsf{nil}) \wedge \mathsf{path}\ \mathsf{nil} \\ = & \quad \{ \text{ path.1 } \} \\ & \mathsf{arc}'\ y\ \mathsf{nil} \end{array}$$

$$\begin{array}{ll} & (\mathsf{arc}'\ y\ x) \wedge \mathsf{path}\ \mathsf{cons}(x, xs) \\ = & \quad \{ \text{ Property 6.7 } \} \\ & (\mathsf{path} \circ (y :_e))\ \mathsf{cons}(x, xs) \\ = & \quad \{ \text{ path.3 and Definition of } (-)_e \} \\ & \mathbf{case}\ y\ \mathbf{of} \\ & \quad \mathsf{inl}() \ \rightarrow \mathsf{path}\ \mathsf{cons}(x, xs) \\ & \quad \mathsf{inr}(z) \rightarrow \mathsf{arc}\ z\ x \wedge \mathsf{path}\ \mathsf{cons}(x, xs) \end{array}$$

Hence, the predicate arc' we are after is:

$$\begin{array}{lcll} \mathsf{arc}'\ y\ \mathsf{nil} & = & \mathsf{true} & (\mathsf{arc}'.1) \\ \mathsf{arc}'\ y\ \mathsf{cons}(x, xs) & = & \mathbf{case}\ y\ \mathbf{of} & (\mathsf{arc}'.2) \\ & & \quad \mathsf{inl}() \ \rightarrow \mathsf{true} & \\ & & \quad \mathsf{inr}(z) \rightarrow \mathsf{arc}\ z\ x & \end{array}$$

Figure 6.2: Derivation of arc'

Case $y = \mathsf{inl}()$

$$\begin{array}{ll} & (\mathsf{list}\,(y :_e x :) \circ \mathsf{filter}\ (\mathsf{arc}'\ y \circ (x :)))\ (\mathsf{filter}\ (\mathsf{path} \circ (x :))\ p'_x) \\ = & \quad \{ \text{ Definition of } (-)_e \text{ lifting } \} \\ & (\mathsf{list}\,(x :) \circ \mathsf{filter}\ (\lambda l.\mathsf{true}))\ (\mathsf{filter}\ (\mathsf{path} \circ (x :))\ p'_x) \\ = & \quad \{ \text{ Corollary A.5 } \} \\ & \mathsf{list}\,(x :)\ (\mathsf{filter}\ (\mathsf{path} \circ (x :))\ p'_x) \\ = & \quad \{ \text{ Proposition A.1 } \} \\ & \mathsf{filter}\ \mathsf{path}\ (\mathsf{list}\,(x :)\ p'_x) \\ = & \quad \{ \ p_x = \mathsf{list}\,(x :)\ p'_x \ \} \\ & \mathsf{filter}\ \mathsf{path}\ p_x \end{array}$$

Case $y = \mathsf{inr}(z)$

$$\begin{array}{ll} & (\mathsf{list}\,(y :_e x :) \circ \mathsf{filter}\ (\mathsf{arc}'\ y \circ (x :)))\ (\mathsf{filter}\ (\mathsf{path} \circ (x :))\ p'_x) \\ = & \quad \{ \text{ Definition of } (-)_e \text{ lifting } \} \\ & (\mathsf{list}\,(y : x :) \circ \mathsf{filter}\ (\lambda l.\mathsf{arc}\ z\ x))\ (\mathsf{filter}\ (\mathsf{path} \circ (x :))\ p'_x) \\ = & \quad \{ \text{ Proposition A.4 } \} \\ & \mathsf{list}\,(y : x :)\ (\,\mathbf{if}\ \mathsf{arc}\ z\ x\ \mathbf{then}\ \mathsf{filter}\ (\mathsf{path} \circ (x :))\ p'_x \\ & \qquad\qquad\qquad \mathbf{else}\ \mathsf{nil}\ ()) \end{array}$$

$$
\begin{array}{ll}
= & \quad \{ \text{ Conditional } \} \\
& \mathbf{if}\ \mathsf{arc}\ z\ x\ \mathbf{then}\ \mathsf{list}\,(y\!:\!x\!:)\ (\mathsf{filter}\ (\mathsf{path} \circ (x\!:))\ p'_x) \\
& \qquad\qquad \mathbf{else}\ \mathsf{nil}\ () \\
= & \quad \{ \text{ Functors, proposition A.1 } \} \\
& \mathbf{if}\ \mathsf{arc}\ z\ x\ \mathbf{then}\ (\mathsf{list}\,(y\!:) \circ \mathsf{filter}\ \mathsf{path} \circ \mathsf{list}\,(x\!:))\ p'_x \\
& \qquad\qquad \mathbf{else}\ \mathsf{nil}\ () \\
= & \quad \{ \ p_x = \mathsf{list}\,(x\!:)\ p'_x \ \} \\
& \mathbf{if}\ \mathsf{arc}\ z\ x\ \mathbf{then}\ \mathsf{list}\,(y\!:)\ (\mathsf{filter}\ \mathsf{path}\ p_x) \\
& \qquad\qquad \mathbf{else}\ \mathsf{nil}\ ()
\end{array}
$$

Putting both cases together,

$$
(\mathsf{filter}\ \mathsf{path})\ (\mathsf{list}\,(y\!:_e)\ p_x) = \begin{array}[t]{l}
\mathbf{case}\ y\ \mathbf{of} \\
\quad \mathsf{inl}() \ \to \mathsf{filter}\ \mathsf{path}\ p_x \\
\quad \mathsf{inr}(z) \to \ \mathbf{if}\ \mathsf{arc}\ z\ x\ \mathbf{then}\ \mathsf{list}\,(y\!:)\ (\mathsf{filter}\ \mathsf{path})\ p_x) \\
\qquad\qquad\qquad\qquad \mathbf{else}\ \mathsf{nil}\ ()
\end{array}
$$

Returning to the main derivation,

$$
\begin{array}{ll}
& ((\mathsf{filter}\ \mathsf{path}) \circ k_2)\ ((p_y, x, p_x), y) \\
= & \quad \{ \text{ Previous calculations } \} \\
& (\mathsf{filter}\ \mathsf{path})\ p_y \ +\!\!+ \ \begin{array}[t]{l}
\mathbf{case}\ y\ \mathbf{of} \\
\quad \mathsf{inl}() \ \to \mathsf{filter}\ \mathsf{path}\ p_x \\
\quad \mathsf{inr}(z) \to \ \mathbf{if}\ \mathsf{arc}\ z\ x\ \mathbf{then}\ \mathsf{list}\,(y\!:)\ (\mathsf{filter}\ \mathsf{path}\ p_x) \\
\qquad\qquad\qquad\qquad \mathbf{else}\ \mathsf{nil}\ ()
\end{array}
\end{array}
$$

By equation 6.6

$$
\begin{array}{ll}
& k'_2((\mathsf{filter}\ \mathsf{path}\ p_x, x, \mathsf{filter}\ \mathsf{path}\ p_y), y) = \begin{array}[t]{l}
\mathsf{filter}\ \mathsf{path}\ p_y \ +\!\!+ \\
\ \mathbf{case}\ y\ \mathbf{of} \\
\quad \mathsf{inl}() \ \to \mathsf{filter}\ \mathsf{path}\ p_x \\
\quad \mathsf{inr}(z) \to \ \mathbf{if}\ \mathsf{arc}\ z\ x \\
\qquad\qquad\qquad \mathbf{then}\ \ \mathsf{list}\,(y\!:)\ (\mathsf{filter}\ \mathsf{path}\ p_x) \\
\qquad\qquad\qquad \mathbf{else}\ \ \ \mathsf{nil}\ ()
\end{array} \\
\Leftarrow & \quad \{ \text{ Generalising } (\mathsf{filter}\ \mathsf{path}\ p_x) \text{ to } q_x \text{ and } (\mathsf{filter}\ \mathsf{path}\ p_y) \text{ to } q_y \ \} \\
& k'_2((q_y, x, q_x), y) = q_y +\!\!+ \ \begin{array}[t]{l}
\mathbf{case}\ y\ \mathbf{of} \\
\quad \mathsf{inl}() \ \to q_x \\
\quad \mathsf{inr}(z) \to \ \mathbf{if}\ \mathsf{arc}\ z\ x\ \mathbf{then}\ \mathsf{list}\,(z\!:)\ q_x \\
\qquad\qquad\qquad\qquad \mathbf{else}\ \mathsf{nil}\ ()
\end{array} \\
= & \quad \{ \text{ Coproducts } \} \\
& k'_2((q_y, x, q_x), y) = \ \begin{array}[t]{l}
\mathbf{case}\ y\ \mathbf{of} \\
\quad \mathsf{inl}() \ \to q_y +\!\!+ q_x \\
\quad \mathsf{inr}(z) \to q_y +\!\!+ \mathbf{if}\ \mathsf{arc}\ z\ x\ \mathbf{then}\ \mathsf{list}\,(z\!:)\ q_x \\
\qquad\qquad\qquad\qquad \mathbf{else}\ \mathsf{nil}\ ()
\end{array}
\end{array}
$$

We have obtained k'_1 and k'_2. So, the fusion of $\mathsf{filter}\ \mathsf{path}$ and asubs_e, function fps, is

$$
\begin{array}{lll}
\mathsf{fps}\ xs & = & \mathsf{afold}_{L,T}([k_1', k_2'] \circ d, \tau_\sigma)\ (xs, \mathsf{inl}()) \\
& & \text{where } k_1'(y) \quad = \quad \mathbf{case}\ y\ \mathbf{of} \\
& & \qquad \mathsf{inl}() \to (\mathsf{wrap} \circ \mathsf{nil})\ () \\
& & \qquad \mathsf{inr}(z) \to (\mathsf{wrap} \circ \mathsf{wrap})\ z \\
& & k_2'((q_y, x, q_x), y) \quad = \quad \mathbf{case}\ y\ \mathbf{of} \\
& & \qquad \mathsf{inl}() \to q_y \mathbin{+\!\!+} q_x \\
& & \qquad \mathsf{inr}(z) \to q_y \mathbin{+\!\!+} \mathbf{if}\ \mathsf{arc}\ z\ x\ \mathbf{then}\ \mathsf{list}\,(z:)\ q_x \\
& & \qquad\qquad \mathbf{else}\ \mathsf{nil}\ () \\
& & \tau_\sigma(x, xs, y) \quad = \quad ((xs, y), x, (xs, \mathsf{inr}\ x), y)
\end{array}
$$

Inlining the above function gives the following result:

$$
\begin{array}{lll}
\mathsf{fps}(\mathsf{nil}, \mathsf{inl}()) & = & (\mathsf{wrap} \circ \mathsf{nil})\ () \\
\mathsf{fps}(\mathsf{nil}, \mathsf{inr}(z)) & = & (\mathsf{wrap} \circ \mathsf{wrap})\ z \\
\mathsf{fps}(\mathsf{cons}(x, xs), \mathsf{inl}()) & = & \mathsf{fps}\ (xs, \mathsf{inl}()) \mathbin{+\!\!+} \mathsf{fps}\ (xs, \mathsf{inr}\ x) \\
\mathsf{fps}(\mathsf{cons}(x, xs), \mathsf{inr}(z)) & = & \mathsf{fps}\ (xs, \mathsf{inr}(z)) \mathbin{+\!\!+} \mathbf{if}\ \mathsf{arc}\ z\ x\ \mathbf{then}\ \mathsf{list}\,(z:)\ (\mathsf{fps}\ (xs, \mathsf{inr}\ x)) \\
& & \qquad \mathbf{else}\ \mathsf{nil}\ ()
\end{array}
$$

6.2.3 Fusing $(\mathsf{maximum} \circ \mathsf{list}\ (\mathsf{length})) \circ \mathsf{fps}$

So far, we have obtained fps, which is the fusion of $\mathsf{filter}\ \mathsf{path}$ and asubs_e. Now we will fuse $(\mathsf{maximum} \circ \mathsf{list}\,(\mathsf{length})) \circ \mathsf{fps}$ to obtain our final result, function llp. Here length is the function that gives the length of a list:

$$
\begin{array}{lll}
\mathsf{length}\ \mathsf{nil} & = & 0 \qquad (\mathsf{length}.1) \\
\mathsf{length}\ (\mathsf{cons}(x, xs)) & = & 1 + \mathsf{length}\ xs \qquad (\mathsf{length}.2)
\end{array}
$$

and $\mathsf{maximum}$ gives the maximum of a list of positive integers.

$$
\begin{array}{lll}
\mathsf{maximum}\ \mathsf{nil} & = & 0 \qquad (\mathsf{maximum}.1) \\
\mathsf{maximum}\ (\mathsf{cons}(x, xs)) & = & \mathsf{max}\ (x, \mathsf{maximum}\ xs) \qquad (\mathsf{maximum}.2)
\end{array}
$$

where max gives the maximum of a pair of integers.

Let us call $mll = \mathsf{maximum} \circ \mathsf{list}\,(\mathsf{length})$.

According to the Afold Pure Fusion (5.13), we have to find k'' such that $mll \circ k' = k'' \circ (T mll \times \mathsf{id})$. Applying corollary 5.8 to this equation we obtain:

$$mll \circ k_1' = k_1'' \tag{6.8}$$

$$mll \circ k_2' = k_2'' \circ ((mll \times \mathsf{id} \times mll) \times \mathsf{id}_X) \tag{6.9}$$

From equation 6.8, after a few calculations we obtain $k_1''\ (y) = \begin{array}{l} \mathbf{case}\ y\ \mathbf{of} \\ \quad \mathsf{inl}() \to 0 \\ \quad \mathsf{inr}(z) \to 1 \end{array}$.

Calculating from the LHS of equation 6.9,

$$
\begin{array}{ll}
 & (mll \circ k_2')\ ((q_y, x, q_x), y) \\
= & \quad \{\ \text{Definition of } k_2'\ \} \\
 & mll\ (\ \textbf{case}\ y\ \textbf{of} \\
 & \qquad \mathsf{inl}() \ \rightarrow q_y +\!\!+ q_x \\
 & \qquad \mathsf{inr}(z) \rightarrow q_y +\!\!+ \textbf{if}\ \mathsf{arc}\ z\ x\ \textbf{then}\ \mathsf{list}\ (z:)\ q_x \\
 & \qquad\qquad\qquad\qquad \textbf{else}\ \mathsf{nil}\ ()) \\
= & \quad \{\ \text{Coproducts}\ \} \\
 & \textbf{case}\ y\ \textbf{of} \\
 & \quad \mathsf{inl}() \ \rightarrow mll\ (q_y +\!\!+ q_x) \\
 & \quad \mathsf{inr}(z) \rightarrow mll\ (q_y +\!\!+ \textbf{if}\ \mathsf{arc}\ z\ x\ \textbf{then}\ \mathsf{list}\ (z:)\ q_x \\
 & \qquad\qquad\qquad\qquad \textbf{else}\ \mathsf{nil}\ ()) \\
= & \quad \{\ \text{Naturality of } +\!\!+ \text{, proposition A.6}\ \} \\
 & \textbf{case}\ y\ \textbf{of} \\
 & \quad \mathsf{inl}() \ \rightarrow \mathsf{max}\ (mll\ q_y, mll\ q_x) \\
 & \quad \mathsf{inr}(z) \rightarrow \mathsf{max}\ (mll\ q_y, mll\ (\textbf{if}\ \mathsf{arc}\ z\ x\ \textbf{then}\ \mathsf{list}\ (z:)\ q_x \\
 & \qquad\qquad\qquad\qquad\qquad \textbf{else}\ \mathsf{nil}\ ())) \\
= & \quad \{\ \text{Conditional}\ \} \\
 & \textbf{case}\ y\ \textbf{of} \\
 & \quad \mathsf{inl}() \ \rightarrow \mathsf{max}\ (mll\ q_y, mll\ q_x) \\
 & \quad \mathsf{inr}(z) \rightarrow \mathsf{max}\ (mll\ q_y,\ \textbf{if}\ \mathsf{arc}\ z\ x\ \textbf{then}\ mll\ (\mathsf{list}\ (z:)\ q_x) \\
 & \qquad\qquad\qquad\qquad\qquad \textbf{else}\ mll\ (\mathsf{nil}\ ())) \\
= & \quad \{\ mll\ (\mathsf{list}\ (z:)\ q_x) = 1 + mll\ q_x, mll\ (\mathsf{nil}\ ()) = 0 \text{ — proof obligations}\ \} \\
 & \textbf{case}\ y\ \textbf{of} \\
 & \quad \mathsf{inl}() \ \rightarrow \mathsf{max}\ (mll\ q_y, mll\ q_x) \\
 & \quad \mathsf{inr}(z) \rightarrow \mathsf{max}\ (mll\ q_y,\ \textbf{if}\ \mathsf{arc}\ z\ x\ \textbf{then}\ 1 + mll\ q_x \\
 & \qquad\qquad\qquad\qquad\qquad \textbf{else}\ 0)
\end{array}
$$

The first proof obligations is

$$
\begin{array}{ll}
 & mll\ (\mathsf{list}\ (z:)\ q_x) \\
= & \quad \{\ \text{Definition of } mll\ \} \\
 & (\mathsf{maximum} \circ \mathsf{list}\ (\mathsf{length}) \circ \mathsf{list}\ (z:))\ q_x \\
= & \quad \{\ \text{Functors, proposition A.7}\ \} \\
 & (\mathsf{maximum} \circ \mathsf{list}\ (1+) \circ \mathsf{list}\ (\mathsf{length}))\ q_x \\
= & \quad \{\ \text{Proposition A.8}\ \} \\
 & ((1+) \circ \mathsf{maximum} \circ \mathsf{list}\ (\mathsf{length}))\ q_x \\
= & \quad \{\ \text{Definition of } mll\ \} \\
 & 1 + mll\ q_x
\end{array}
$$

And the second one is

$$
\begin{aligned}
& \mathit{mll}\ (\mathsf{nil}\ ()) \\
= & \qquad \{ \ \text{Definition of } \mathit{mll} \ \} \\
& (\mathsf{maximum} \circ \mathsf{list}\,(\mathsf{length}))\ (\mathsf{nil}\ ()) \\
= & \qquad \{ \ \text{Type Functor} \ \} \\
& \mathsf{maximum}\ (\mathsf{nil}\ ()) \\
= & \qquad \{ \ \mathsf{maximum.1} \ \} \\
& 0
\end{aligned}
$$

We have obtained

$$
\begin{aligned}
& k_2''\ ((\mathit{mll}\ q_y, x, \mathit{mll}\ q_x), y) = \ \textbf{case}\ y\ \textbf{of} \\
& \qquad \mathsf{inl}() \ \to \mathsf{max}\ (\mathit{mll}\ q_y, \mathit{mll}\ q_x) \\
& \qquad \mathsf{inr}(z) \to \mathsf{max}\ (\mathit{mll}\ q_y,\ \textbf{if}\ \mathsf{arc}\ z\ x\ \textbf{then}\ 1 + \mathit{mll}\ q_x\ \textbf{else}\ 0) \\
\Leftarrow & \qquad \{ \ \text{Generalising } \mathit{mll}\ q_x \text{ to } r_x \text{ and } \mathit{mll}\ q_y \text{ to } r_y \ \} \\
& k_2''\ ((r_y, x, r_x), y) = \ \textbf{case}\ y\ \textbf{of} \\
& \qquad \mathsf{inl}() \ \to \mathsf{max}\ (r_y, r_x) \\
& \qquad \mathsf{inr}(z) \to \mathsf{max}\ (r_y,\ \textbf{if}\ \mathsf{arc}\ z\ x\ \textbf{then}\ 1 + r_x\ \textbf{else}\ 0)
\end{aligned}
$$

The result of the fusion of llp, fllp is

$$
\begin{aligned}
\mathsf{fllp}\ xs \ = \ & \mathsf{afold}_{L,T}([k_1'', k_2''] \circ d, \tau_\sigma)\ (xs, e) \\
\text{where}\ & k_1''\ (y) = \ \textbf{case}\ y\ \textbf{of} \\
& \qquad \mathsf{inl}() \ \to 0 \\
& \qquad \mathsf{inr}(z) \to 1 \\
& k_2''\ ((r_y, x, r_x), y) = \ \textbf{case}\ y\ \textbf{of} \\
& \qquad \mathsf{inl}() \ \to \mathsf{max}\ (r_y, r_x) \\
& \qquad \mathsf{inr}(z) \to \mathsf{max}\ (r_y,\ \textbf{if}\ \mathsf{arc}\ z\ x\ \textbf{then}\ 1 + r_x\ \textbf{else}\ 0) \\
& \tau_\sigma(x, l, y) = ((l, y), x, (l, \mathsf{inr}\ x), y)
\end{aligned}
$$

Inlining the above function gives as a result:

$$
\begin{aligned}
\mathsf{fllp}\ (\mathsf{nil}, \mathsf{inl}()) &= 0 \\
\mathsf{fllp}\ (\mathsf{nil}, \mathsf{inr}(z)) &= 1 \\
\mathsf{fllp}\ (\mathsf{cons}(x, xs), \mathsf{inl}()) &= \mathsf{max}\ (\mathsf{fllp}\ (xs, \mathsf{inl}()),\ \mathsf{fllp}\ (xs, \mathsf{inr}\ x)) \\
\mathsf{fllp}\ (\mathsf{cons}(x, xs), \mathsf{inr}(z)) &= \mathsf{max}\ (\mathsf{fllp}\ (xs, \mathsf{inr}(z)),\ \textbf{if}\ \mathsf{arc}\ z\ x\ \textbf{then}\ 1 + \mathsf{fllp}\ (xs, \mathsf{inr}\ x)\ \textbf{else}\ 0)
\end{aligned}
$$

Our final program is

$$\mathsf{llp}'\ xs = \mathsf{flpp}\ (xs, \mathsf{inl}())$$

6.3 Summary

We have started from a simple but inefficient specification of the path sequence problem and by means of program calculation —and a heavy use of fusion laws— we have obtained an efficient program. We had to derive an accumulation for subs since the fusion with $\mathsf{filter\ path}$ could not be performed otherwise.

Chapter 7

Conclusions

In this thesis several aspects of the problem of calculating programs in the presence of accumulators were considered. The motivation for this interest is that accumulations are in widespread use in functional programs but they are difficult to reason with when using standard recursion operators since they require the use of currying and higher order.

The standard category-theoretical modelling of types and programs was used as foundation. The main reason for choosing this representation is its ability to abstract from the details of specific datatypes and to serve as a streamlined proof framework. A presentation of this model was made in chapter 2, which introduced the concepts and tools that would be needed in later chapters. As such, it is by no means intended to be exhaustive; other sources of information are [BdM97, BJJM99, Fok92, JR97].

In chapter 3 the afold generic recursive operator on inductive datatypes is presented. [Par01] introduced afold, along with a collection of algebraic laws. A simpler presentation of this operator and its laws can be found in [Par02], on which our presentation is based. In these previous works only polynomial datatypes were considered. One of the contributions of this thesis is the study of the structure of afold in the presence of regular datatypes, showed by the instance of afold for rose trees. The rest of the contributions are located from chapter 4 onwards.

The structure of some accumulations, in particular tail recursive accumulations, yield as a result that fusion is not completely effective, as it does not eliminate all the intermediate structures. In chapter 4 a technique was introduced that improves the fusion of such accumulations. This technique is illustrated by two examples. This first example was taken from [Voi03], where the problem of sub-optimal fusions was pointed out. The second example is the classic function foldl, typical example of a tail recursive function. Although its fusion law is already known [Bir98], here it is derived in calculational form from its expression as an accumulation.

The afold operator, as it was defined, was unable to express certain kinds of functions where accumulations have more than one recursive call in each subterm, with different accumulator values. In chapter 5 an extension to afold that copes with this limitation is proposed. This extension has proved to be conservative: laws for the extended operator are similar to those of the original afold. Additionally, it has been found that this extended operator can be factorised in the composition of a fold whose algebra is a natural transformation, with the original afold. This fact proved to be extremely useful when proving the extended operator laws, as can be seen in the simplicity of the proofs in Appendix B.

Finally, in chapter 6, we present a case study for the path sequence problem. This problem was originally solved with the use of accumulators in [Bir84], and was taken up again in [HIT96]. In this work, a program is derived from an specification of the problem using the extended operator and its associated laws introduced in chapter 5.

Appendix A

Simple Properties

In this appendix we list simple propositions that were used in the case study. We will not provide proofs of these propositions, but references will be provided when known.

Proposition A.1 [BdM97, Tho99] $\mathsf{list}\,(f) \circ \mathsf{filter}\ (h \circ f) = \mathsf{filter}\ \mathsf{h} \circ \mathsf{list}\,(f)$.

Proposition A.2 $(\mathsf{filter}\ p)\ (xs +\!\!+ ys) = \mathsf{filter}\ p\ xs +\!\!+ \mathsf{filter}\ p\ ys$

Proposition A.3 [Bir98] $\mathsf{filter}\ p\ xs = (\mathsf{filter}\ f \circ \mathsf{filter}\ g)\ xs$
where $p\ x = f\ x \wedge g\ x$

Proposition A.4 x *does not occur free in* $b \Rightarrow \mathsf{filter}\ (\lambda x.b)\ xs = \ \mathbf{if}\ b\ \mathbf{then}\ xs\ \mathbf{else}\ \mathsf{nil}()$

Corollary A.5 $\mathsf{filter}\ (\lambda x.\mathsf{true})\ xs = xs$

Proposition A.6 $\mathsf{maximum}\ (xs +\!\!+ ys) = \mathsf{max}\ (\mathsf{maximum}\ xs, \mathsf{maximum}\ ys)$

Proposition A.7 $\mathsf{length} \circ (z:) = 1 + \mathsf{length}$

Proposition A.8 $\mathsf{maximum} \circ \mathsf{list}\,(1+) = (1+) \circ \mathsf{maximum}$

Appendix B

Proofs

Proposition 5.4
If $\mathcal{C}$ is a cartesian closed category, then every initial algebra is initial with accumulators.

Proof. Let in_F be initial. Consider an X-action $h\colon FA \times X \to A$. With it construct the F-algebra

$$k = \mathsf{curry}(\overline{k})\colon F[X \to A] \to [X \to A]$$

where

$$\overline{k} = h \circ \langle G_\sigma \mathsf{apply}, \pi_2 \rangle \colon F[X \to A] \times X \to A$$

Now consider the following composite diagram:

$$\begin{tikzcd}
F\mu F \times \mathsf{id}_X \arrow[r, "F\mathsf{fold}_F(k) \times \mathsf{id}_X"] \arrow[d, "in_F \times \mathsf{id}_X"'] \arrow[dr, phantom, "(I)"] & F[X \to A] \times X \arrow[r, "\langle G_\sigma \mathsf{apply}, \pi_2 \rangle"] \arrow[d, "k \times \mathsf{id}_X"'] \arrow[dr, phantom, "(II)"] & GA \times X \arrow[d, "h"] \\
\mu F \times \mathsf{id}_X \arrow[r, "\mathsf{fold}_F(k) \times \mathsf{id}_X"'] & {[X \to A]} \arrow[r, "\mathsf{apply}"'] & A
\end{tikzcd}$$

(I) commutes by definition of fold, whereas (II) commutes by the universal property of the exponential. i.e.

$$\mathsf{apply} \circ \mathsf{curry}(\overline{k} \times \mathsf{id}_X) = \overline{k}$$

Therefore, the outer rectangle commutes. By the bijection between the curried and uncurried version of an arrow, we have that there is a unique $f\colon \mu F \times X \to A$ such that $\mathsf{apply} \circ (\mathsf{fold}_F(k) \times \mathsf{id}_X) = f$. Since

$$\begin{aligned}
& \langle G_\sigma \mathsf{apply}, \pi_2 \rangle \circ (F\mathsf{fold}_F(k) \times \mathsf{id}_X) \\
= \quad & \quad \{ \text{ Products } \} \\
& \langle G_\sigma \mathsf{apply} \circ (F\mathsf{fold}_F(k) \times \mathsf{id}_X), \pi_2 \rangle \\
= \quad & \quad \{ \text{ Definition 5.2 } \} \\
& \langle G \mathsf{apply} \circ \tau_\sigma \circ (F\mathsf{fold}_F(k) \times \mathsf{id}_X), \pi_2 \rangle \\
= \quad & \quad \{ \text{ Natural transformation } \tau_\sigma, \text{ Functors } \} \\
& \langle G_\sigma (\mathsf{apply} \circ (\mathsf{fold}_F(k) \times \mathsf{id}_X)), \pi_2 \rangle
\end{aligned}$$

it follows that f is the unique arrow such that

$$f \circ (in_F \times \mathsf{id}_X) = h \circ \langle G_\sigma f, \pi_2 \rangle$$

and therefore in_F is initial with accumulators. $\square$

In the sequel we take $\tau_\sigma = \overline{\tau} \circ (\sigma \times \mathsf{id}_X)$ and $\tau'_\sigma = \overline{\tau}' \circ (\sigma \times \mathsf{id}_X)$

Theorem B.1 (Lemma) *Let $G = G_1 + G_2$, $\tau_\sigma = (\overline{\tau}_1 + \overline{\tau}_2) \circ d \circ (\sigma \times \mathsf{id}_X)$, where $\sigma = \sigma_1 + \sigma_2$, then*

$$d \circ \langle G_\sigma f, \pi_2 \rangle = (\langle G_{1\sigma_1} f, \pi_2 \rangle + \langle G_{2\sigma_2} f, \pi_2 \rangle) \circ d$$

Proof.

$$\begin{array}{ll}
 & d \circ \langle G_\sigma f, \pi_2 \rangle \\
= & \quad \{ \text{ Definition of } G\text{, definition 5.2 } \} \\
 & d \circ \langle (G_1 + G_2) f \circ (\overline{\tau}_1 + \overline{\tau}_2) \circ d \circ (\sigma \times \mathsf{id}_X), \pi_2 \rangle \\
= & \quad \{ \text{ Naturality of } d \ \} \\
 & d \circ \langle (G_1 f + G_2 f) \circ (\overline{\tau}_1 + \overline{\tau}_2) \circ (\sigma_1 \times \mathsf{id}_X + \sigma_2 \times \mathsf{id}_X) \circ d, \pi_2 \rangle \\
= & \quad \{ \text{ Definition 5.2 } \} \\
 & d \circ \langle (G_{1\sigma_1} f + G_{2\sigma_2} f) \circ d, \pi_2 \rangle \\
= & \quad \{ \text{ Proof obligation } \} \\
 & (\langle G_{1\sigma_1} f, \pi_2 \rangle + \langle G_{2\sigma_2} f, \pi_2 \rangle) \circ d
\end{array}$$

The proof obligation is $d \circ \langle (h + k) \circ d, \pi_2 \rangle = (\langle h, \pi_2 \rangle + \langle k, \pi_2 \rangle) \circ d$. Since d is an isomorphism, this is the same as proving

$$\langle (h + k) \circ d, \pi_2 \rangle \circ d^{-1} = d^{-1} \circ (\langle h, \pi_2 \rangle + \langle k, \pi_2 \rangle)$$

The proof goes like this

$$\begin{array}{ll}
 & \langle (h + k) \circ d, \pi_2 \rangle \circ d^{-1} \\
= & \quad \{ \text{ Products } \} \\
 & \langle (h + k), [\pi_2, \pi_2] \rangle \\
= & \quad \{ \text{ Coproducts } \} \\
 & \langle [\mathsf{inl} \circ h, \mathsf{inr} \circ k], [\pi_2, \pi_2] \rangle \\
= & \quad \{ \text{ Exchange Law } \} \\
 & [\langle \mathsf{inl} \circ h, \pi_2 \rangle, \langle \mathsf{inr} \circ k, \pi_2 \rangle] \\
= & \quad \{ \text{ Products } \} \\
 & [(\mathsf{inl} \times \mathsf{id}) \circ \langle h, \pi_2 \rangle, (\mathsf{inr} \times \mathsf{id}) \circ \langle k, \pi_2 \rangle] \\
= & \quad \{ \text{ Coproducts } \} \\
 & [\mathsf{inl} \times \mathsf{id}, \mathsf{inr} \times \mathsf{id}] \circ (\langle h, \pi_2 \rangle + \langle k, \pi_2 \rangle) \\
= & \quad \{ \text{ Definition of } d^{-1} \ \} \\
 & d^{-1} \circ (\langle h, \pi_2 \rangle + \langle k, \pi_2 \rangle)
\end{array}$$

□

Proposition 5.6

Let $G = G_1 + G_2$ be a composite functor, $h = [h_1, h_2] \circ d$, $G_\sigma f = Gf \circ (\overline{\tau}_1 + \overline{\tau}_2) \circ d \circ (\sigma \times id_X)$, where $\sigma = \sigma_1 + \sigma_2$, and $k = [k_1, k_2] \circ d$. Then

$$f \circ h = k \circ \langle G_\sigma f, \pi_2 \rangle \quad \Leftrightarrow \quad \begin{cases} f \circ h_1 = k_1 \circ \langle G_{1\sigma_1} f, \pi_2 \rangle \\ f \circ h_2 = k_2 \circ \langle G_{2\sigma_2} f, \pi_2 \rangle \end{cases}$$

Proof. Consider $k = [k_1, k_2] \circ d : GA \times X \to A$, with $k_1 : G_1 A \to A$ and $k_2 : G_2 A \to A$. Then,

$$\begin{aligned}
& f \circ h = k \circ \langle G_\sigma f, \pi_2 \rangle \\
\equiv & \quad \{ \ h = [h_1, h_2] \circ d, k = [k_1, k_2] \circ d \ \} \\
& f \circ [h_1, h_2] \circ d = [k_1, k_2] \circ d \circ \langle G_\sigma f, \pi_2 \rangle \\
\equiv & \quad \{ \ \text{Lemma B.1} \ \} \\
& f \circ [h_1, h_2] \circ d = [k_1, k_2] \circ (\langle G_{1\sigma_1} f, \pi_2 \rangle + \langle G_{2\sigma_2} f, \pi_2 \rangle) \circ d \\
\equiv & \quad \{ \ \text{Post-composing by } d^{-1} \ \} \\
& f \circ [h_1, h_2] = [k_1, k_2] \circ (\langle G_{1\sigma_1} f, \pi_2 \rangle + \langle G_{2\sigma_2} f, \pi_2 \rangle) \\
\equiv & \quad \{ \ \text{Coproducts} \ \} \\
& [f \circ h_1, f \circ h_2] = [k_1 \circ \langle G_{1\sigma_1} f, \pi_2 \rangle, k_2 \circ \langle G_{2\sigma_2} f, \pi_2 \rangle]
\end{aligned}$$

By case analysis, we have the desired result. □

Corolary 5.7

Let $G = G_1 + G_2$ be a composite functor, $h' = [h'_1, h'_2]$, $G_\sigma f = Gf \circ (\overline{\tau}_1 + \overline{\tau}_2) \circ d \circ (\sigma \times id_X)$, where $\sigma = \sigma_1 + \sigma_2$, and $k = [k_1, k_2] \circ d$. Then

$$f \circ (h' \times \mathsf{id}_X) = k \circ \langle G_\sigma f, \pi_2 \rangle \quad \Leftrightarrow \quad \begin{cases} f \circ (h'_1 \times id_X) = k_1 \circ \langle G_{1\sigma_1} f, \pi_2 \rangle \\ f \circ (h'_2 \times id_X) = k_2 \circ \langle G_{2\sigma_2} f, \pi_2 \rangle \end{cases}$$

Proof. Take proposition 5.6, and let's consider this h in particular:

$$h = [h'_1 \times \mathsf{id}, h'_2 \times \mathsf{id}] \circ d$$

$$\begin{aligned}
& h \\
= & \quad \{ \ \text{Definition of } h \ \} \\
& [h'_1 \times \mathsf{id}, h'_2 \times \mathsf{id}] \circ d \\
= & \quad \{ \ [g \times f, h \times f] \circ d = [g, h] \times f \text{ (see example 2.27)} \ \} \\
& [h'_1, h'_2] \times id
\end{aligned}$$

□

Corolary 5.8
Let $G = G_1 + G_2$ be a composite functor, $h = [h_1, h_2] \circ d$, and $k = [k_1, k_2] \circ d$. Then

$$f \circ h = k \circ (Gf \times \mathsf{id}) \quad \Leftrightarrow \quad \begin{cases} f \circ h_1 = k_1 \circ (G_1 f \times \mathsf{id}) \\ \\ f \circ h_2 = k_2 \circ (G_2 f \times \mathsf{id}) \end{cases}$$

Proof. Take proposition 5.6, also take $\sigma = id$ and $\overline{\tau}_1 = \pi_1, \overline{\tau}_2 = \pi_1$.

$$\begin{array}{ll} & \langle G_\sigma f, \pi_2 \rangle \\ = & \quad \{ \text{ Definition of } G_\sigma \ \} \\ & \langle Gf \circ ((\pi_1 + \pi_1) \circ d), \pi_2 \rangle \\ = & \quad \{ \ (\pi_1 + \pi_1) \circ d = \pi_1 \text{ (see example 2.26) } \} \\ & \langle Gf \circ \pi_1, \pi_2 \rangle \\ = & \quad \{ \text{ Products } \} \\ & (Gf \times \mathsf{id}) \circ \langle \pi_1, \pi_2 \rangle \\ = & \quad \{ \text{ Product identity } \} \\ & Gf \times \mathsf{id} \end{array}$$

The proofs that $\langle G_{1\sigma_1} f, \pi_2 \rangle = G_1 f \times \mathsf{id}$ and $\langle G_{2\sigma_2} f, \pi_2 \rangle = G_2 f \times \mathsf{id}$ are analogous. □

Theorem 5.9 (Afold Factorization)
Let $\overline{\tau}$ be proper for accumulation and $\sigma : F \Rightarrow G$, then

$$\mathsf{afold}_{F,G}(h, \tau_\sigma) = \mathsf{afold}_G(h, \overline{\tau}) \circ (\mathsf{fold}_F(in_G \circ \sigma) \times id_X)$$

where $\tau_\sigma = \overline{\tau} \circ (\sigma \times id_X)$.

Proof. Let us consider the following diagram:

$$\begin{tikzcd}
F\mu F \times X \arrow[r, "F\mathsf{fold}_F(in_G \circ \sigma) \times \mathsf{id}_X"] \arrow[dd, "in_F \times \mathsf{id}_X"] \arrow[dr, phantom, "(I)"] & F\mu G \times X \arrow[d, "\sigma \times \mathsf{id}_X"'] \arrow[dr, "\langle G\mathsf{afold}_G(h, \overline{\tau}) \circ \tau_\sigma, \pi_2 \rangle"] & \\
 & G\mu G \times X \arrow[r, "\langle \overline{G}\mathsf{afold}_G(h, \overline{\tau}), \pi_2 \rangle"'] \arrow[d, "in_G \times X"'] \arrow[dr, phantom, "(II)"] & GA \times X \arrow[d, "h"'] \\
\mu F \times X \arrow[r, "\mathsf{fold}_F(in_G \circ \sigma) \times \mathsf{id}_X"] \arrow[rr, bend right, "\mathsf{afold}_{F,G}(h, \tau_\sigma)"'] & \mu G \times X \arrow[r, "\mathsf{afold}_G(h, \overline{\tau})"] & A
\end{tikzcd}$$

The triangle in the diagram commutes:

$$\begin{tikzcd}
F\mu F \times X \arrow[r, "F\mathsf{fold}_F(in_F \circ \sigma) \times \mathsf{id}_X"] \arrow[d, "\tau_{\sigma,\mu F}"'] & F\mu G \times X \arrow[d, "\tau_{\sigma,\mu G}"] \\
G(\mu F \times X) \arrow[r, "G(\mathsf{fold}_F(in_F \circ \sigma) \times \mathsf{id}_X)"'] & G(\mu G \times X)
\end{tikzcd}$$

Figure B.1: Using the naturality of τ_σ

$$\begin{array}{cl}
 & \langle \overline{G}\mathsf{afold}_G(h,\overline{\tau}), \pi_2\rangle \circ (\sigma \times \mathsf{id}_X) \\
= & \quad \{ \text{ Definition of } \overline{G} \ \} \\
 & \langle G\mathsf{afold}_G(h,\overline{\tau}) \circ \overline{\tau}, \pi_2\rangle \circ (\sigma \times \mathsf{id}_X) \\
= & \quad \{ \text{ Products } \} \\
 & \langle G\mathsf{afold}_G(h,\overline{\tau}) \circ \overline{\tau} \circ (\sigma \times \mathsf{id}_X), \pi_2\rangle \\
= & \quad \{ \ \tau_\sigma = \overline{\tau} \circ (\sigma \times id_X) \ \} \\
 & \langle G\mathsf{afold}_G(h,\overline{\tau}) \circ \tau_\sigma, \pi_2\rangle
\end{array}$$

(I) and (II) commute by definition of fold (2.1) and afold (3.3) respectively.
Since

$$\begin{array}{cl}
 & \langle G\mathsf{afold}_G(h,\overline{\tau}) \circ \tau_\sigma, \pi_2\rangle \circ (F\mathsf{fold}_F(in_F \circ \sigma) \times \mathsf{id}_X) \\
= & \quad \{ \text{ Products } \} \\
 & \langle G\mathsf{afold}_G(h,\overline{\tau}) \circ \tau_\sigma \circ (F\mathsf{fold}_F(in_F \circ \sigma) \times \mathsf{id}_X), \pi_2\rangle \\
= & \quad \{ \ \tau_\sigma \text{ is a natural transformation (see Figure B.1) } \} \\
 & \langle G\mathsf{afold}_G(h,\overline{\tau}) \circ G(\mathsf{fold}_F(in_F \circ \sigma) \times \mathsf{id}_X) \circ \tau_\sigma, \pi_2\rangle \\
= & \quad \{ \text{ Functors } \} \\
 & \langle G(\mathsf{afold}_G(h,\overline{\tau}) \circ (\mathsf{fold}_F(in_F \circ \sigma) \times \mathsf{id}_X)) \circ \tau_\sigma, \pi_2\rangle
\end{array}$$

we have that the following diagram commutes.

$$\begin{tikzcd}
F\mu F \times X \arrow[r, "\langle G_\sigma(\mathsf{afold}_G(h,\overline{\tau}) \circ (\mathsf{fold}_F(in_F \circ \sigma) \times \mathsf{id}_X)), \pi_2\rangle"] \arrow[d, "in_F \times \mathsf{id}_X"'] & GA \times X \arrow[d, "h"] \\
\mu F \times X \arrow[r, "\mathsf{afold}_G(h,\overline{\tau}) \circ (\mathsf{fold}_F(in_F \circ \sigma) \times \mathsf{id}_X)"] & A
\end{tikzcd}$$

By initiality with accumulators (5.3), the theorem is proved.

□

Theorem 5.10 (Afold Transformation Shift)

For every natural transformation $\kappa : F \Rightarrow G$, $\sigma : F \Rightarrow F$,

$$\begin{array}{l} \quad \kappa \circ \tau_\sigma = \tau'_\sigma \circ (\kappa \times \mathsf{id}) \\ \Rightarrow \\ \quad \mathsf{afold}_{F,G}(h, \tau'_{\sigma\circ\kappa}) = \mathsf{afold}_{F,F}(h \circ (\kappa \times \mathsf{id}), \tau_\sigma) \end{array}$$

Proof. The diagram for the afold on the left hand side is

$$\begin{tikzcd} F\mu F \times X \arrow[r, "\langle G_{\sigma\circ\kappa} f, \pi_2\rangle"] \arrow[d, "in_F \times \mathsf{id}_X"'] & GA \times X \arrow[d, "h"] \\ \mu F \times X \arrow[r, "f"'] & A \end{tikzcd}$$

If we take the arrow on the top of the diagram, and make some calculations,

$$\begin{array}{ll} & \langle G_{\sigma\circ\kappa} f, \pi_2\rangle \\ = & \quad \{ \text{ Definition of } G_\sigma \ \} \\ & \langle Gf \circ \overline{\tau}' \circ (\sigma \circ \kappa \times \mathsf{id}), \pi_2\rangle \\ = & \quad \{ \text{ Products } \} \\ & \langle Gf \circ \tau'_\sigma \circ (\kappa \times \mathsf{id}), \pi_2\rangle \\ = & \quad \{ \text{ Hypothesis } \} \\ & \langle Gf \circ \kappa \circ \tau_\sigma, \pi_2\rangle \\ = & \quad \{ \text{ Natural Transformation } \kappa \ \} \\ & \langle \kappa \circ Ff \circ \tau_\sigma, \pi_2\rangle \\ = & \quad \{ \text{ Products } \} \\ & (\kappa \times \mathsf{id}) \circ \langle Ff \circ \tau_\sigma, \pi_2\rangle \end{array}$$

we have the following diagram:

$$\begin{tikzcd} F\mu F \times X \arrow[r, "\langle F_\sigma f, \pi_2\rangle"] \arrow[dd, "in_F \times \mathsf{id}_X"'] & FA \times X \arrow[d, "(\kappa \times \mathsf{id}_X)"] \\ & GA \times X \arrow[d, "h"] \\ \mu F \times X \arrow[r, "f"'] & A \end{tikzcd}$$

$\square$

Theorem 5.11
For any $\overline{\tau}$,

$$\mathsf{fold}_F(h) \circ \pi_1 = \mathsf{afold}_{F,F}(h \circ \pi_1, \overline{\tau})$$

Proof.

$$\begin{array}{ll} & \mathsf{fold}_F(h) \circ \pi_1 \\ = & \quad \{ \text{ Afold Lifting (3.13) } \} \\ & \mathsf{afold}_F(h \circ \pi_1, \overline{\tau}) \\ = & \quad \{ \text{ Proposition 5.5 } \} \\ & \mathsf{afold}_{F,F}(h \circ \pi_1, \overline{\tau}) \end{array}$$

□

Theorem 5.12 (Afold Identity)

$$\mathsf{afold}_{F,F}(in_F \circ \pi_1, \overline{\tau}) = \pi_1$$

Proof.

$$\begin{array}{ll} & \mathsf{afold}_{F,F}(in_F \circ \pi_1, \overline{\tau}) \\ = & \quad \{ \text{ Proposition 5.5 } \} \\ & \mathsf{afold}_F(in_F \circ \pi_1, \overline{\tau}) \\ = & \quad \{ \text{ Afold Identity (3.14) } \} \\ & \pi_1 \end{array}$$

□

Theorem 5.13 (Afold Pure Fusion)

$$f \circ h = h' \circ (Gf \times \mathsf{id}) \quad \Rightarrow \quad f \circ \mathsf{afold}_{F,G}(h, \tau_\sigma) = \mathsf{afold}_{F,G}(h', \tau_\sigma)$$

Proof.

$$\begin{array}{ll} & f \circ \mathsf{afold}_{F,G}(h, \tau_\sigma) \\ = & \quad \{ \text{ Afold Factorization (5.9) } \} \\ & f \circ \mathsf{afold}_G(h, \overline{\tau}) \circ (\mathsf{fold}_F(in_G \circ \sigma) \times \mathsf{id}) \\ = & \quad \{ \text{ Afold Pure Fusion (3.15) } \} \\ & \mathsf{afold}_G(h', \overline{\tau}) \circ (\mathsf{fold}_F(in_G \circ \sigma) \times \mathsf{id}) \\ = & \quad \{ \text{ Afold Factorization (5.9) } \} \\ & \mathsf{afold}_{F,G}(h', \tau_\sigma) \end{array}$$

□

Theorem 5.14 (Acid Rain: Afold-Fold)

$$\begin{array}{ll} & T : \forall A.\, (HA \to A) \to (GA \times X \to A) \\ \Rightarrow & \\ & \mathsf{fold}_H(h) \circ \mathsf{afold}_{F,G}(\boldsymbol{T}(in_H), \tau_\sigma) = \mathsf{afold}_{F,G}(\boldsymbol{T}(h), \tau_\sigma) \end{array}$$

Proof.

$$
\begin{aligned}
& \mathsf{fold}_H(h) \circ \mathsf{afold}_{F,G}(\boldsymbol{T}(in_H), \tau_\sigma) \\
= & \quad \{ \text{ Afold Factorization (5.9) } \} \\
& \mathsf{fold}_H(h) \circ \mathsf{afold}_G(\boldsymbol{T}(in_H), \overline{\tau}) \circ (\mathsf{fold}_F(in_G \circ \sigma) \times \mathsf{id}) \\
= & \quad \{ \text{ Acid Rain: Afold-Fold Fusion (3.16) } \} \\
& \mathsf{afold}_G(\boldsymbol{T}(h), \overline{\tau}) \circ (\mathsf{fold}_F(in_G \circ \sigma) \times \mathsf{id}) \\
= & \quad \{ \text{ Afold Factorization (5.9) } \} \\
& \mathsf{afold}_{F,G}(\boldsymbol{T}(h), \tau_\sigma)
\end{aligned}
$$

□

Theorem 5.15 (Fold-Afold Transformation Fusion)
For every natural transformation $\kappa : H \Rightarrow F$,

$$\mathsf{afold}_{F,G}(h, \tau'_\sigma) \circ (\mathsf{fold}_H(in_F \circ \kappa) \times \mathsf{id}) = \mathsf{afold}_{H,G}(h, \tau_{\sigma\circ\kappa})$$

Proof.

$$
\begin{aligned}
& \mathsf{afold}_{F,G}(h, \tau'_\sigma) \circ (\mathsf{fold}_H(in_F \circ \kappa) \times \mathsf{id}) \\
= & \quad \{ \text{ Afold Factorization (5.9) } \} \\
& \mathsf{afold}_G(h, \overline{\tau}) \circ (\mathsf{fold}_F(in_G \circ \sigma) \times \mathsf{id}) \circ (\mathsf{fold}_H(in_F \circ \kappa) \times \mathsf{id}) \\
= & \quad \{ \text{ Products } \} \\
& \mathsf{afold}_G(h, \overline{\tau}) \circ (\mathsf{fold}_F(in_G \circ \sigma) \circ \mathsf{fold}_H(in_F \circ \kappa)) \times \mathsf{id} \\
= & \quad \{ \text{ Acid Rain: Fold-Fold Fusion (2.43) } \} \\
& \mathsf{afold}_G(h, \overline{\tau}) \circ (\mathsf{fold}_H(in_G \circ \sigma \circ \kappa) \times \mathsf{id}) \\
= & \quad \{ \text{ Afold Factorization (5.9) } \} \\
& \mathsf{afold}_{H,G}(h, \tau_{\sigma\circ\kappa})
\end{aligned}
$$

□

Corollary 5.16 (Fold-Afold Fusion)
If $\kappa : G \Rightarrow F$ and $\sigma : F \Rightarrow F$,

$$
\begin{aligned}
& \kappa \circ \tau_\sigma = \tau'_\sigma \circ (\kappa \times \mathsf{id}) \\
\Rightarrow & \\
& \mathsf{afold}_{F,F}(h, \tau'_\sigma) \circ (\mathsf{fold}_G(in_F \circ \kappa) \times \mathsf{id}) = \mathsf{afold}_{F,F}(h \circ (\kappa \times \mathsf{id}), \tau_\sigma)
\end{aligned}
$$

Proof.

$$
\begin{aligned}
& \mathsf{afold}_{F,F}(h, \tau'_\sigma) \circ (\mathsf{fold}_G(in_F \circ \kappa) \times \mathsf{id}) \\
= & \quad \{ \text{ Fold-Afold Transformation Fusion (5.15) } \} \\
& \mathsf{afold}_{G,F}(h, \tau_{\sigma\circ\kappa}) \\
= & \quad \{ \text{ Afold Transformation Shift (5.10) } \} \\
& \mathsf{afold}_{F,F}(h \circ (\kappa \times \mathsf{id}), \tau_\sigma)
\end{aligned}
$$

□

Theorem 5.17 (Map-Afold Fusion)
For $f : A \to B$ and $DA = \mu F_A$,

$$G(f, \mathsf{id}) \circ \overline{\tau} = \overline{\tau}' \circ (G(f, \mathsf{id}) \times \mathsf{id})$$
$$\Rightarrow$$
$$\mathsf{afold}_{F_B,G_B}(h, \tau'_\sigma) \circ (Df \times \mathsf{id}) = \mathsf{afold}_{F_A,G_A}(h \circ (G(f, \mathsf{id}) \times \mathsf{id}), \tau_\sigma)$$

Proof.

$$
\begin{array}{ll}
& \mathsf{afold}_{F_B,G_B}(h, \tau'_\sigma) \circ (Df \times \mathsf{id}) \\
= & \quad \{ \text{ Afold Factorization (5.9) } \} \\
& \mathsf{afold}_G(h, \overline{\tau}') \circ (\mathsf{fold}_{F_B}(in_G \circ \sigma) \times \mathsf{id}) \circ (Df \times \mathsf{id}) \\
= & \quad \{ \text{ Products } \} \\
& \mathsf{afold}_G(h, \overline{\tau}') \circ (\mathsf{fold}_{F_B}(in_G \circ \sigma) \circ Df \times \mathsf{id}) \\
= & \quad \{ \text{ Map-Fold Fusion (2.45) } \} \\
& \mathsf{afold}_G(h, \overline{\tau}') \circ (\mathsf{fold}_{F_A}(in_G \circ \sigma \circ F(f, \mathsf{id})) \times \mathsf{id}) \\
= & \quad \{ \text{ Afold Factorization (5.9), } F(f, \mathsf{id}) \text{ is natural on its } 2^{nd} \text{ argument } \} \\
& \mathsf{afold}_{F_A,G_A}(h, \overline{\tau}' \circ (\sigma \circ F(f, \mathsf{id}) \times \mathsf{id})) \\
= & \quad \{ \text{ Natural Transformation } \sigma \ \} \\
& \mathsf{afold}_{F_A,G_A}(h, \overline{\tau}' \circ (G(f, \mathsf{id}) \circ \sigma \times \mathsf{id})) \\
= & \quad \{ \text{ Hypothesis } \} \\
& \mathsf{afold}_{F_A,G_A}(h, G(f, \mathsf{id}) \circ \tau_\sigma) \\
= & \quad \{ \text{ Functors } \} \\
& \mathsf{afold}_{F_A,G_A}(h \circ (G(f, \mathsf{id}) \times \mathsf{id}), \tau_\sigma)
\end{array}
$$

Where in the last step, we use the fact that

$$\langle G(\mathsf{id}, g) \circ G(f, \mathsf{id}) \circ \tau_\sigma, \pi_2 \rangle = (G(f, \mathsf{id}) \times \mathsf{id}) \circ \langle G(\mathsf{id}, g) \circ \tau_\sigma, \pi_2 \rangle$$

□

Theorem 5.18 (Morph-Afold Fusion)
For every $f : X \to X'$,

$$G(\mathsf{id} \times f) \circ \overline{\tau} = \overline{\tau}' \circ (\mathsf{id} \times f)$$
$$\Rightarrow$$
$$\mathsf{afold}_{F,G}(h, \tau'_\sigma) \circ (\mathsf{id} \times f) = \mathsf{afold}_{F,G}(h \circ (\mathsf{id} \times f), \tau_\sigma)$$

Proof.

$$
\begin{aligned}
& \mathsf{afold}_{F,G}(h, \tau'_\sigma) \circ (\mathsf{id} \times f) \\
= \quad & \qquad \{ \text{ Afold Factorization (5.9) } \} \\
& \mathsf{afold}_G(h, \overline{\tau}') \circ (\mathsf{fold}_F(in_G \circ \sigma) \times \mathsf{id}) \circ (\mathsf{id} \times f) \\
= \quad & \qquad \{ \text{ Products } \} \\
& \mathsf{afold}_G(h, \overline{\tau}') \circ (\mathsf{id} \times f) \circ (\mathsf{fold}_F(in_G \circ \sigma) \times \mathsf{id}) \\
= \quad & \qquad \{ \text{ Morph-Afold Fusion (3.19) } \} \\
& \mathsf{afold}_G(h \circ (\mathsf{id} \times f), \overline{\tau}) \circ (\mathsf{fold}_F(in_G \circ \sigma) \times \mathsf{id}) \\
= \quad & \qquad \{ \text{ Afold Factorization (5.9) } \} \\
& \mathsf{afold}_{F,G}(h \circ (\mathsf{id} \times f), \tau'_\sigma)
\end{aligned}
$$

□

Proposition 5.19

Let $f : A \times X \to A$ be a function with right identity e, i.e. $f(a, e) = a$, for every a. Then,

$$f \circ (h \times \mathsf{id}_X) = k \circ \langle G_\sigma f, \pi_2 \rangle \quad \Rightarrow \quad \mathsf{fold}_F(h)(t) = \mathsf{afold}_{F,G}(k, \tau_\sigma)(t, e)$$

where $G_\sigma f = Gf \circ \tau_\sigma$, for τ_σ proper for accumulation.

Proof. First, let us consider the following composite diagram:

$$
\begin{tikzcd}
F\mu F \times X \arrow[r, "F\,\mathsf{fold}_F(h) \times \mathsf{id}_X"] \arrow[d, "in_F \times \mathsf{id}_X"'] \arrow[dr, phantom, "(I)"] & FA \times X \arrow[r, "\langle G_\sigma f, \pi_2 \rangle"] \arrow[d, "h \times \mathsf{id}_X"] \arrow[dr, phantom, "(II)"] & GA \times X \arrow[d, "k"] \\
\mu F \times X \arrow[r, "\mathsf{fold}_F(h) \times \mathsf{id}_X"'] & A \times X \arrow[r, "f"'] & A
\end{tikzcd}
$$

(I) commutes by definition of fold, while (II) commutes by hypothesis. Since,

$$\langle G_\sigma f, \pi_2 \rangle \circ (F\,\mathsf{fold}_F(h) \times \mathsf{id}_X) = \langle G_\sigma(f \circ (\mathsf{fold}_F(h) \times \mathsf{id}_X)), \pi_2 \rangle$$

by initiality with accumulators we obtain that:

$$f \circ (\mathsf{fold}_F(h) \times \mathsf{id}_X) = \mathsf{afold}_{F,G}(k, \tau_\sigma)$$

Therefore,

$$\mathsf{fold}_F(h)(t) = f(\mathsf{fold}_F(h)(t), e) = \mathsf{afold}_{F,G}(k, \tau_\sigma)(t, e)$$

as desired. □

Corollary 5.20

Let $f : A \times X \to A$ be a function with right identity e, i.e. $f(a, e) = a$, for every a. Let $G = G_1 + G_2$ and $F = F_1 + F_2$ be composite functors, $h = [h_1, h_2]$, $G_\sigma f = Gf \circ (\overline{\tau}_1 + \overline{\tau}_2) \circ d \circ (\sigma \times id_X)$, where $\sigma = \sigma_1 + \sigma_2$, and $k = [k_1, k_2] \circ d$. Then, for every $in_F = [c_1, c_2] \colon F\mu F \to \mu F$

$$
\left.
\begin{aligned}
f \circ (h_1 \times id_X) &= k_1 \circ \langle G_{1\sigma_1} f, \pi_2 \rangle \\
f \circ (h_2 \times id_X) &= k_2 \circ \langle G_{2\sigma_2} f, \pi_2 \rangle
\end{aligned}
\right\} \Rightarrow
\left\{
\begin{aligned}
\mathsf{fold}_F(h) \circ c_1 &= \mathsf{afold}_{F,G}(k, \tau_\sigma) \circ \langle c_1, \underline{e} \rangle \\
\mathsf{fold}_F(h) \circ c_2 &= \mathsf{afold}_{F,G}(k, \tau_\sigma) \circ \langle c_2, \underline{e} \rangle
\end{aligned}
\right.
$$

Proof. This corollary is simply the application of proposition 5.7 to proposition 5.19. □

Bibliography

[AL91] A. Asperti and G. Longo. *Categories, Types and Structures: An Introduction to Category Theory for the Working Computer Scientist*. Foundations of Computing. MIT Press, Cambridge, Massachusetts, 1991.

[BdM97] R.S. Bird and O. de Moor. *Algebra of Programming*. Prentice Hall, UK, 1997.

[Bir84] R.S. Bird. The Promotion and Accumulation Strategies in Transformational Programming. *ACM Transactions on Programming Languages and Systems*, 6(4), October 1984.

[Bir98] R. Bird. *Introduction to Functional Programming using Haskell*. Prentice-Hall, UK, 2nd edition, 1998.

[BJJM99] R. Backhouse, P. Jansson, J. Jeuring, and L. Meertens. Generic Programming - An Introduction -. In *Advanced Functional Programming,* LNCS 1608. Springer-Verlag, 1999.

[BW99] M. Barr and C. Wells. *Category Theory for Computing Science*. Les Publications CRM, Montréal, 3rd edition, 1999.

[CDPR98] Loïc Correnson, Etienne Duris, Didier Parigot, and Gilles Roussel. How to deforest in accumulative parameters? Technical report, INRIA Rocquencourt, 1998.

[Cor99] Loïc Correnson. Equational Semantics. In D. Parigot and M. Mernik, editors, *Second Workshop on Attribute Grammars and their Applications, WAGA'99*, pages 205–222, Amsterdam, The Netherlands, 1999. INRIA Rocquencourt.

[Fok92] Maarten M. Fokkinga. *Law and Order in Algorithmics*. PhD thesis, University of Twente, 7500 AE Enschede, Netherlands, February 1992.

[Fok96] M.M. Fokkinga. Datatype Laws without Signatures. *Mathematical Structures in Computer Science*, 6:1–32, 1996.

[Gib00] J. Gibbons. Generic Downwards Accumulations. *Science of Computer Programming*, 37(1–3):37–65, 2000.

[Hin99] R. Hinze. Polytypic Programming with Ease. In *4th Fuji International Symposium on Functional and Logic Programming (FLOPS'99), Tsukuba, Japan.*, Lecture Notes in Computer Science Vol. 1722, pages 21–36. Springer-Verlag, 1999.

[HIT96] Z. Hu, H. Iwasaki, and M. Takeichi. Calculating Accumulations. Technical Report METR 96-03, Faculty of Engineering, University of Tokyo, March 1996.

[Hug89] J. Hughes. Why Functional Programming Matters. *Computer Journal*, 32(2):98–107, 1989.

[Hut99] Graham Hutton. A Tutorial on the Universality and Expressiveness of Fold. *Journal of Functional Programming*, 9(4):355–372, July 1999.

[Jeu93] J. Jeuring. *Theories for Algorithm Calculation*. PhD thesis, Utrecht University, 1993.

[JR97] B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *Bulletin of EATCS*, 62:222–259, 1997.

[Lan71] S. Mac Lane. *Categories for the Working Mathematician*. Springer Verlag, 1971.

[LS81] D.J. Lehmann and M.B. Smith. Algebraic specification of data types. *Mathematical Systems Theory*, 14:97–139, 1981.

[MA86] E.G. Manes and M.A. Arbib. *Algebraic Approaches to Program Semantics*. Texts and Monographs in Computer Science. Springer-Verlag, 1986.

[Mal90] G. Malcolm. Data Structures and Program Transformation. *Science of Computer Programming*, 14:255–279, 1990.

[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.

[Par00] A. Pardo. Towards Merging Recursion and Comonads. In *Workshop on Generic Programming*, Ponte de Lima, Portugal, July 2000. Technical Report UU-CS-2000-19, Utrecht University.

[Par01] A. Pardo. *A Calculational Approach to Recursive Programs with Effects*. PhD thesis, Technische Universität Darmstadt, October 2001.

[Par02] A. Pardo. Generic Accumulations. In *IFIP WG2.1 Working Conference on Generic Programming*, Dagstuhl, Germany, July 2002.

[Pie91] B.C. Pierce. *Basic Category Theory for Computer Scientists*. Foundations of Computing. MIT Press, Cambridge, Massachusetts, 1991.

[Rey83] J.C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing'83*, 1983.

[Tho99] S. Thompson. *Haskell: The Craft of Functional Programming*. Addison-Wesley, 1999.

[VK04] Janis Voigtländer and Armin Kühnemann. Composition of functions with accumulating parameters. *Journal of Functional Programming*, 14:317–363, 2004.

[Voi03] Janis Voigtländer. Elimination of intermediate results in functional programs. Colloquium at TU München, November 2003.

[Wad89] P. Wadler. Theorems for free! In *The 4th International Conference on Functional Programming Languages and Computer Architecture (FPCA '89)*, pages 347–359, London, September 1989. Imperial College, ACM Press.

[Wad90] P. Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73(2):231–248, 1990.