Generic Accumulations

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

Abstract

Accumulations are recursive functions that keep intermediate results in additional parameters which are eventually used in later stages of the computation. We present a generic definition of accumulations obtained by the introduction of a new recursive operator on inductive types. We also show that the notion of downwards accumulation developed by Gibbons is subsumed by our notion of accumulation.

1 Introduction

Accumulations are recursive functions that keep intermediate results in additional parameters often called accumulators [19, 5, 15]. In functional programming, the notion of accumulation is usually associated with the so-called accumulation technique [8, 4, 17, 3], which transforms recursive definitions by the introduction of additional arguments over which intermediate results are computed. The accumulation technique is strongly connected with the familiar procedure of *generalization for induction* that arises in the field of theorem proving [7, 1, 26]: A proof by induction often fails because the property to be proved is too particular. Then it is necessary to modify/generalize the induction hypothesis before starting the proof. This situation often appears during *program verification*, for instance, when a given program is proved to satisfy its formal specification, a procedure that in general requires induction (see e.g. [17, 30]).

In this paper, we present a generic definition of accumulations that works uniformly for any inductive type. The kind of accumulations we have in mind are those that pass information down to the recursive calls. This paper follows up on an initial proposal presented in [28]. A drawback of the version of accumulation given in [28] is that it is too specific in the form it defines the modification of the accumulating parameters. This problem has been eliminated in the present version, which shows a higher degree of genericity in addition to being more elegant.

Generic accumulations have already been the subject of study of other works in the field of program calculation. Gibbons [15], for example, develops a generic definition of so-called downwards accumulations. These are functions that label every node of a tree with a function of its ancestors. We show that our notion of accumulation includes that of downwards accumulation as a particular case.

The remainder of the paper is organized as follows. Section 2 introduces the mathematical framework the paper is based on. In Section 3 we briefly review the definition of functions with constant parameters. This section serves as preamble and motivation for the definition of accumulations presented in Section 4. In fact, our notion of accumulation will be obtained by performing a slight modification to that of function with parameters. Section 5 is devoted to show that downwards accumulations are a particular case of accumulations. Section 6 concludes the paper giving some final remarks.

2 Mathematical Framework

Our approach to genericity is based on the category-theoretic modelling of types and programs. This representation, by now standard, turns out to be an appropriate framework for reasoning algebraically about programs and is the basis for current developments in generic programming (see e.g. [2, 18]). In this section we review the relevant concepts around the categorical approach to recursive types [23, 25, 21] and its application to program calculation [24, 27, 11, 22, 5].

The category-theoretic explanation of (recursive) types is based on the idea that types constitute objects of a category $\mathcal{C}$, programs are modelled by arrows of the category, and type constructors are functors on $\mathcal{C}$. In this setting, a datatype T is understood as a solution (a fixed point) of a type equation $X \cong FX$, for an appropriate endofunctor $F : \mathcal{C} \to \mathcal{C}$ that captures the shape (or signature) of the type.

2.1 Product and Sum

Throughout we shall assume that $\mathcal{C}$ is a category with finite products $(\times, 1)$ and finite coproducts $(+, 0)$, where $0/1$ denotes the initial/final object of $\mathcal{C}$. The leading example of such a category is **Set**, the category of sets and total functions.

The unique arrow from A to 1 is written $!_A : A \to 1$. Due to the isomorphism between an object A and $1 \times A$, the application of a function $f : 1 \times A \to B$ to the unique value of the type 1 and a value a of type A will be written as $f(a)$.

We write $\pi_1 : A \times B \to A$ and $\pi_2 : A \times B \to B$ to denote the product projections. The pairing of two arrows $f : C \to A$ and $g : C \to B$ is denoted by $\langle f, g \rangle : C \to A \times B$. Product associativity is denoted by $\alpha_{A,B,C} : A \times (B \times C) \to (A \times B) \times C$. The coproduct inclusions are written $\mathsf{inl} : A \to A + B$ and $\mathsf{inr} : B \to A + B$. For $f : A \to C$ and $g : B \to C$, case analysis is the unique morphism $[f, g] : A + B \to C$ such that $[f, g] \circ \mathsf{inl} = f$ and $[f, g] \circ \mathsf{inr} = g$. In pointwise notation we shall often write $[f, g](x)$ as:

$$\textbf{case } x \textbf{ of } \mathsf{inl}(a) \to f(a);\ \mathsf{inr}(b) \to g(b)$$

Product and coproduct can be made into bifunctors $\mathcal{C} \times \mathcal{C} \to \mathcal{C}$ by defining their action on arrows (see e.g. [5]). It is also straightforward to obtain their generalizations to n components.

Along the paper 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 and the category **Cpo** of complete partial orders (not necessarily having a bottom element) and continuous functions.

2.2 Polynomial functors

We consider datatypes with signatures given by so-called *polynomial functors*. The following is an inductive definition of this class of functors:

$$F ::= I \mid \underline{A}^n \mid \Pi_i^n \mid \times \mid + \mid F\langle F, \ldots, F \rangle$$

$I : \mathcal{C} \to \mathcal{C}$ stands for the identity functor. $\underline{A}^n : \mathcal{C}^n \to \mathcal{C}$ denotes the n-ary constant functor, which maps n-tuples of objects to the object A; when $n = 1$ we simply write $\underline{A}$. $\Pi_i^n : \mathcal{C}^n \to \mathcal{C}$ (with $n \geq 2$) denotes the i-th projection functor from a n-ary product category. $F\langle G_1, \ldots, G_n\rangle$ (or $F\langle G_i\rangle$ for short) denotes the composition of $F : \mathcal{C}^n \to \mathcal{C}$ with the functors $G_1, \ldots, G_n$ (all of the same arity); when $n = 1$ we omit brackets. It stands for the functor that maps $A \mapsto F(G_1 A, \ldots, G_n A)$. We write $F \dagger G$ for $\dagger\langle F, G\rangle$ when $\dagger \in \{\times, +\}$.

2.3 Inductive Types

Least fixpoints of (covariant) functors give rise to *inductive types*, which correspond to initial functor-algebras, a generalization of the usual notion of term algebras over a given signature.

Let $F : \mathcal{C} \to \mathcal{C}$ be a functor. An *F-algebra* is an arrow $h : FA \to A$, called the operation. The object A is called the carrier of the algebra. A morphism of algebras, or *F-homomorphism*, between $h : FA \to A$ and $k : FB \to B$ is an arrow $f : A \to B$ such that $f \circ h = k \circ Ff$. The category of F-algebras is formed by considering F-algebras as objects and F-homomorphisms as morphisms. The initial algebra, if it exists, gives the inductive type whose signature is captured by F. We shall denote the initial algebra by $in_F : F\mu F \to \mu F$. This arrow encodes the constructors of the inductive type and turns out to be an isomorphism.

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* [3] or *catamorphism* [27], 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 equation hold:

$$\mathsf{fold}_F(h) \circ in_F = h \circ F\, \mathsf{fold}_F(h)$$

Example 2.1 Consider a datatype for natural numbers,

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

Its signature is captured by the functor $N : \mathcal{C} \to \mathcal{C}$ such that $NA = 1 + A$ and $Nf = \mathsf{id}_1 + f$. Every N-algebra is a case analysis $[h_1, h_2] : 1 + A \to A$, with $h_1 : 1 \to A$ and $h_2 : A \to A$; in particular, the initial algebra $[\mathsf{zero}, \mathsf{succ}] : 1 + \mathsf{nat} \to \mathsf{nat}$ where $\mathsf{zero} : 1 \to \mathsf{nat}$ and $\mathsf{succ} : \mathsf{nat} \to \mathsf{nat}$. For each algebra $h = [h_1, h_2]$, fold is the unique arrow $f = \mathsf{fold}_N(h) : \mathsf{nat} \to A$ such that

$$f(\mathsf{zero}) = h_1 \qquad\qquad f(\mathsf{succ}(n)) = h_2(f(n))$$

□

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 2.2

(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

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

It corresponds to the standard foldr operator used in functional programming [3].

(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

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

□

From each parameterised datatype $DA = \mu F_A$, we can define a functor $D : \mathcal{C} \to \mathcal{C}$, called a *type functor* [5], by specifying its action on arrows $Df : DA \to DB$, for $f : A \to B$,

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

For instance, $\mathsf{list}(f) = \mathsf{fold}_{L_A}([\mathsf{nil}, \mathsf{cons} \circ (f \times \mathsf{id})])$, which corresponds to the usual map function on lists [3]:

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

The following are standard laws of fold.

Fold Identity

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

Fold Fusion

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

Acid Rain: Fold-Fold Fusion

$$\boldsymbol{T} \text{ transformer } \ \Rightarrow\ \mathsf{fold}_F(h) \circ \mathsf{fold}_G(\boldsymbol{T}(in_F)) = \mathsf{fold}_G(\boldsymbol{T}(h))$$

Map-Fold Fusion For $f : A \to B$ and $h : F_B\, C \to C$,

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

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. A *transformer* [12] is a mapping $\boldsymbol{T} : \forall A.\, (FA \to A) \to (GA \to A)$ from F-algebras to G-algebras that preserves homomorphisms, i.e., for $f : A \to B$, $h : FA \to A$ and $h' : FB \to B$,

$$f \circ h = h' \circ Ff \quad \Rightarrow \quad f \circ \boldsymbol{T}(h) = \boldsymbol{T}(h') \circ Gf$$

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.

2.4 Strong Functors

It might be the case that product not only distributes over coproduct, but also over an arbitrary functor. This property is what characterizes the so-called *strong functors*. In this paper, strong functors will play an essential role in the definition of recursive functions with additional parameters.

If $A = (A_1, \ldots, A_m)$ is an object of $\mathcal{C}^n$ and B an object of $\mathcal{C}$, then let us define $A \times B = (A_1 \times B, \ldots, A_m \times B)$. A functor $F : \mathcal{C}^m \to \mathcal{C}$ is called **strong** if it is equipped with a natural transformation

$$\tau^F_{A,X} : FA \times X \to F(A \times X)$$

called a *strength*, such that the following equations hold:

$$\begin{aligned} F\pi_1 \circ \tau^F_{A,X} &= \pi_1 \\ F\alpha_{A,X,Y} \circ \tau^F_{A,X\times Y} &= \tau^F_{A\times X,X} \circ (\tau^F_{A,X} \times \mathsf{id}_Y) \circ \alpha_{FA,X,Y} \end{aligned}$$

Under the assumption that the underlying category $\mathcal{C}$ is distributive, a strength for each polynomial functor F can be defined by induction on the structure of F.

$$\begin{aligned} \tau^I_{A,X} &= \mathsf{id}_{A\times X} & \tau^\times_{(A,B),X} &= \langle \pi_1 \times \mathsf{id}_X, \pi_2 \times \mathsf{id}_X \rangle \\ \tau^{\underline{C}^n}_{(A_1,\ldots,A_n),X} &= \pi_1 & \tau^+_{(A,B),X} &= d_{A,B,X} \\ \tau^{\Pi^n_i}_{(A_1,\ldots,A_n),X} &= \mathsf{id}_{A_i\times X} & \tau^{F\langle G_i\rangle}_{A,X} &= F(\tau^{G_1}_{A,X}, \ldots, \tau^{G_n}_{A,X}) \circ \tau^F_{(G_i A),X} \end{aligned}$$

Given two strong functors F and G, a natural transformation $\kappa_A : FA \to GA$ is said to be **strong** if it behaves consistently with respect to the strengths: $\kappa_{A\times X} \circ \tau^F_{A,X} = \tau^G_{A,X} \circ (\kappa_A \times \mathsf{id}_X)$.

3 Functions with Parameters

Some recursive functions require additional constant *parameters*, usually representing some context information, for their computation. As an example, consider the addition between natural numbers:

$$\mathsf{add}(\mathsf{zero}, n) = n \qquad \mathsf{add}(\mathsf{succ}(m), n) = \mathsf{succ}(\mathsf{add}(m, n)) \tag{1}$$

In this function the argument n remains unchanged throughout the recursion.

There are several ways in which a function of this kind can be defined. One is by using *currying*, something that is common practice in functional programming. A possible curried version is one that binds the constant parameters globally. That is, a definition of type $X \to [\mu F \to A]$, where X represents the type of parameters, μF the recursive datatype, and A the type of the result. In the case of addition it corresponds to:

$$\mathsf{add}(m) = \mathsf{fold}_N(\mathsf{const}(m), \mathsf{succ}) : \mathsf{nat} \to \mathsf{nat}$$

where $\mathsf{const}(m) : 1 \to \mathsf{nat}$ is a constant function returning m. An alternative curried definition is one that maps the recursive argument onto a function on the parameters: $\mu F \to [X \to A]$. In the case of addition,

$$\mathsf{add}(\mathsf{zero}) = \mathsf{id} \qquad\qquad \mathsf{add}(\mathsf{succ}(m)) = \mathsf{succ} \circ \mathsf{add}(m)$$

This version corresponds to a higher-order fold,

$$\mathsf{add} = \mathsf{fold}_N(h) : \mathsf{nat} \to [\mathsf{nat} \to \mathsf{nat}]$$

where $h = [h_1, h_2] : 1 + [\mathsf{nat} \to \mathsf{nat}] \to [\mathsf{nat} \to \mathsf{nat}]$ is given by $h_1(u) = \mathsf{id}_{\mathsf{nat}}$ and $h_2(f) = \mathsf{succ} \circ f$.

An alternative to currying consists of tupling the arguments: $\mu F \times X \to A$. In the case of addition, for instance, this means a definition of type $\mathsf{nat} \times \mathsf{nat} \to \mathsf{nat}$ in the style of (1). However, a problem with definitions of this kind is that they cannot be expressed in terms of fold, since it gives no way of defining functions with multiple arguments. For instance, we cannot write $\mathsf{add} = \mathsf{fold}_N(h) : \mathsf{nat} \times \mathsf{nat} \to \mathsf{nat}$, for some h. This problem can be overcome by defining another operator, called *pfold*, which represents a sort of *fold with parameters*. Categorically speaking, this alternative to currying makes reasoning and calculations much simpler, since in general product types are easier to handle than function spaces. On the other hand, there may be cases in which pfold is the only alternative available to define recursive functions with constant parameters, for example, in a traditional language without higher-order.

To define pfold it is sufficient to assume that the initial algebra is *strongly initial* [10] (or *initial with parameters*) giving rise to the so-called *strong datatypes* [9]. Next we briefly review the concept of strong initiality and the definition of pfold.

Let us fix an object X. We call an arrow of type $A \times X \to B$, for arbitrary A and B, an X*-action*. Composition of two X-actions $f : B \times X \to C$ and $g : A \times X \to B$ is given by $f \bullet g = f \circ \langle g, \pi_2 \rangle$. Let us define $\widetilde{f} = f \circ \pi_1 : A \times X \to B$, for $f : A \to B$.

Strong functors can be lifted to work on X-actions. Given a strong functor F, for each $f : A \times X \to B$, we define $\widetilde{F} f : FA \times X \to FB$ to be:

$$\widetilde{F} f \;=\; FA \times X \xrightarrow{\tau^F_{A,X}} F(A \times X) \xrightarrow{Ff} FB$$

Furthermore, it can be seen that $\widetilde{F}$ is a functor at the level of X-actions, that is, it preserves identities (given by $\pi_1 : A \times X \to A$) and compositions between X-actions (see e.g. [29] for details).

Definition 3.1 ([10]) Given a strong functor F, an initial F-algebra in_F is said to be **strongly initial** if, for each object X and X-action $h : FA \times X \to A$, there exists a unique X-action $f : \mu F \times X \to A$ that makes the following diagram commute

$$\begin{array}{ccc} F\mu F \times X & \xrightarrow{\langle \widetilde{F} f, \pi_2 \rangle} & FA \times X \\ {\scriptstyle in_F \times \mathsf{id}_X}\downarrow & & \downarrow{\scriptstyle h} \\ \mu F \times X & \xrightarrow[f]{} & A \end{array}$$

We call **pfold** [28, 29] the unique arrow that results from strong initiality and denote it by $\mathsf{pfold}_F(h) : \mu F \times X \to A$. □

Note the role played by the strength τ^F (as part of $\widetilde{F}$) in the definition of pfold: it distributes the value of the parameters to the recursive calls. The following proposition guarantees the existence of categories where strong initiality holds.

Proposition 3.2 ([10]) *If $\mathcal{C}$ is a cartesian closed category, then every initial algebra is strongly initial.*

This means that pfold can be defined in categories like **Set** and **Cpo**. This fact turns out to be a corollary of the following more general result.

Proposition 3.3 *Let $\mathcal{C}$ be a cartesian closed category. Then, for any natural transformation $\delta_Y : FY \times X \to F(Y \times X)$ (natural in Y) and X-action $h : FA \times X \to A$, there exists a unique $f : \mu F \times X \to A$ such that $f \circ (in_F \times \mathsf{id}_X) = h \circ \langle Ff \circ \delta, \pi_2 \rangle$.*

The proof of this proposition is completely similar to that of Proposition 3.2 (see e.g. [10, 20, 28]).

Example 3.4

(i) Natural numbers: for each $h = [h_1, h_2] \circ d : (1 + A) \times X \to A$, where $h_1 : 1 \times X \to A$ and $h_2 : A \times X \to A$, pfold is the unique arrow $f = \mathsf{pfold}_N(h) : \mathsf{nat} \times X \to A$ such that

$$f(\mathsf{zero}, x) = h_1(x) \qquad\qquad f(\mathsf{succ}(n), x) = h_2(f(n, x), x)$$

(ii) Lists: for each $h = [h_1, h_2] \circ d : (1 + A \times B) \times X \to B$, pfold is the unique arrow $f = \mathsf{pfold}_{L_A}(h) : A^* \times X \to B$ such that

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

(iii) Leaf-labelled binary trees: for each $h = [h_1, h_2] \circ d : (A + C \times C) \times X \to C$, pfold is the unique arrow $f = \mathsf{pfold}_{B_A}(h) : \mathsf{btree}(A) \times X \to C$ such that

$$\begin{aligned} f\ (\mathsf{leaf}(a), x) &= h_1(a, x) \\ f\ (\mathsf{join}(t, u), x) &= h_2(f(t, x), f(u, x), x) \end{aligned}$$

(iv) Binary trees with information in the nodes: for each $h = [h_1, h_2] \circ d : (1 + B \times A \times B) \times X \to B$, pfold is the unique arrow $f = \mathsf{pfold}_{T_A}(h) : \mathsf{tree}(A) \times X \to B$ such that:

$$\begin{aligned} f\ (\mathsf{empty}, x) &= h_1(x) \\ f\ (\mathsf{node}(t, a, u), x) &= h_2(f(t, x), a, f(u, x), x) \end{aligned}$$

□

The following are some laws for pfold.

Pfold Lifting

$$\widetilde{\mathsf{fold}_F(h)} = \mathsf{pfold}_F(\widetilde{h})$$

Pfold Identity

$$\mathsf{pfold}_F(\widetilde{in_F}) = \pi_1$$

Pfold Fusion

$$f \bullet h = h' \bullet \widetilde{F} f \;\Rightarrow\; f \bullet \mathsf{pfold}_F(h) = \mathsf{pfold}_F(h')$$

Pfold Pure Fusion

$$f \circ h = h' \circ (Ff \times \mathsf{id}) \;\Rightarrow\; f \circ \mathsf{pfold}_F(h) = \mathsf{pfold}_F(h')$$

Acid Rain: Pfold-Fold Fusion

$$\boldsymbol{T} \text{ transformer} \quad\Rightarrow\quad \mathsf{fold}_F(h) \circ \mathsf{pfold}_G(\boldsymbol{T}(in_F)) = \mathsf{pfold}_G(\boldsymbol{T}(h))$$

Fold-Pfold Fusion

$$\begin{array}{ll} & \kappa \text{ strongly natural} \\ \Rightarrow & \\ & \mathsf{pfold}_F(h) \circ (\mathsf{fold}_G(in_F \circ \kappa) \times \mathsf{id}) = \mathsf{pfold}_G(h \circ (\kappa \times \mathsf{id})) \end{array}$$

Map-Pfold Fusion For $f : A \to B$ and $DA = \mu F_A$,

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

Morph-PFold Fusion For every $f : X \to X'$,

$$\mathsf{pfold}_F(h) \circ (\mathsf{id} \times f) = \mathsf{pfold}_F(h \circ (\mathsf{id} \times f))$$

In the acid rain law $\boldsymbol{T}$ stands for a transformer of type $\forall A.\,(FA \to A) \to (GA \times X \to A)$. A proof and examples of the use of these laws can be found in [29].

4 Accumulations

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. [4, 19, 5, 15]). In this section we define a generic operator that permits us to represent structural recursive accumulations on inductive types. The operator is obtained by a small modification in the definition of pfold.

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

$$\mathsf{isums}(\mathsf{nil}, e) \;=\; \mathsf{wrap}(e) \tag{2}$$

$$\mathsf{isums}(\mathsf{cons}(n, \ell), e) \;=\; \mathsf{cons}(e, \mathsf{isums}(\ell, e + n)) \tag{3}$$

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

To define a function of this kind we have two alternatives. One is to define the function as a higher-order fold of type $\mu F \to [X \to A]$, where now X corresponds to the type of accumulators. That is the approach pursued in [19]. The other alternative consists of tupling the arguments, 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 (2) and (3). Like functions with (constant) parameters, accumulations defined in this manner cannot be written in terms of the standard fold operator. The reason is that fold lacks the possibility of representing functions with multiple arguments. Furthermore, accumulations cannot even be written as a pfold because the values of the accumulating parameters change during the computation.

The solution we adopt to express accumulations is similar to the one considered for functions with parameters. In fact, we introduce a new operator, called *afold*, which corresponds to a *fold with accumulators*. The definition of afold will be obtained by working with a modified version of strong initiality that reflects the possibility of performing alterations to the (accumulating) parameters.

In the sequel let us fix an object X that now will be regarded as the type of accumulators. Let us recall the diagram that defines pfold:

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

Existence and uniqueness of an arrow f fulfilling this diagram is what characterizes the notion of strong initiality. Accumulations have a similar structure to functions defined in this manner, the only difference being the fact that accumulations permit to alter the parameters in the recursive calls. We will then take advantage of this similarity to define afold. Since accumulators are modified when passed to the recursive calls, it is easy to see that such a modification has to take place within $\widetilde{F} f$, or more precisely, within the corresponding arrow $FA \times X \to F(A \times X)$. In the case of pfold, that arrow is given by the strength $\tau^F_{A,X}$, which simply distributes the value of the parameters to the recursive calls. In an accumulation, however, a modified value has to be distributed. This means that, to define an accumulation it is necessary to provide an arrow

$$\overline{\tau} : FA \times X \to F(A \times X)$$

that reflects this fact. 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 4.1 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

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

□

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 accumulation independent of 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. This is an immediate consequence of the naturality condition. The second condition coincides with one of the requirements for strengths. It 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 of each other. This is a consequence of the hypothesis about distributivity.

Given $\overline{\tau}$ satisfying Definition 4.1 we can define another extension of functor F that works on X-actions. For $f : A \times X \to B$, let us define $\overline{F}f : FA \times X \to FB$ to be:

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

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 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 4.2 ([29]) 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 $\mathsf{afold}_F(h, \overline{\tau}) : \mu F \times X \to A$. □

Like strong initiality, initiality with accumulators is guaranteed to exist in the presence of exponentials.

Proposition 4.3 *If $\mathcal{C}$ is a cartesian closed category, then every initial algebra is initial with accumulators.*

Like Proposition 3.2, this fact also follows directly from Proposition 3.3. Therefore, accumulations can be defined in categories like **Set** or **Cpo**.

Example 4.4

(i) 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 $\psi : X \to X$. For each $h = [h_1, h_2] \circ d : (1 + A) \times X \to A$, $f = \mathsf{afold}_N(h, \overline{\tau}) : \mathsf{nat} \times X \to A$ is such that:

$$f(\mathsf{zero}, x) = h_1(x) \qquad\qquad f(\mathsf{succ}(n), x) = h_2(f(n, \psi(x)), x)$$

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,

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

(ii) 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 $\psi : A \times X \to X$. For each $h = [h_1, h_2] \circ d : (1 + A \times C) \times X \to C$, $f = \mathsf{afold}_{L_A}(h, \overline{\tau}) : A^* \times X \to C$ is such that:

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

For example, the function isums can be defined by

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

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

(iii) 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). For each $h = [h_1, h_2] \circ d : (A + D \times D) \times X \to D$, $f = \mathsf{afold}_{B_A}(h, \overline{\tau}) : \mathsf{btree}(A) \times X \to D$ is such that:

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

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,

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

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

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

For example, the function $\mathsf{asums} : \mathsf{tree}(\mathsf{nat}) \to \mathsf{tree}(\mathsf{nat})$, 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,

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

$\square$

The following are some laws for afold.

Afold Lifting For any $\overline{\tau}$,

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

Afold Identity

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

Afold Pure Fusion

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

Acid Rain: Afold-Fold Fusion

$$\boldsymbol{T} \text{ transformer} \quad \Rightarrow \quad \mathsf{fold}_F(h) \circ \mathsf{afold}_G(\boldsymbol{T}(in_F), \overline{\tau}) = \mathsf{afold}_G(\boldsymbol{T}(h), \overline{\tau})$$

Fold-Afold Fusion For every natural transformation $\kappa : G \Rightarrow F$,

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

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

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

Morph-Afold Fusion For every $f : X \to X'$,

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

In the acid rain law, $\boldsymbol{T}$ stands for a transformer of type $\forall A.(FA \to A) \to (GA \times X \to A)$. Morph-afold fusion is particularly interesting because it relates two accumulations whose accumulating parameters have 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 [29].

Note 4.5 It is worth mentioning that the strong similarity between these laws and those for pfold is not accidental. In fact, in [29] it is shown that both pfold and afold are particular instances of so-called *comonadic fold*, a recursive operator based on comonads (which are algebraic structures dual to monads). The comonad that corresponds to the case of pfold and afold is the so-called *product comonad*, which has functor $WA = A \times X$. The laws for pfold and afold are then obtained by specialization from laws of comonadic fold. The (small) differences between the laws of pfold and afold is a consequence of different properties enjoyed by τ^F and $\overline{\tau}$.

Morph-afold fusion (as well as morph-pfold fusion) has another feature that makes it especially interesting. It is the fact that it states the composition of a recursive function with a comonad morphism. In fact, afold is composed with function $\mathsf{id} \times f$, which turns out to be a *comonad morphism* between the product comonads $WA = A \times X$ and $W'A = A \times X'$ (see [29] for details). Although it is a very simple case of comonad morphism, to the best of our knowledge this is the first time the concept of comonad morphism is used for program calculation purposes. $\square$

Example 4.6 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:

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

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: $\mathsf{max} \circ ((m+) \times (m+)) = (m+) \circ \mathsf{max}$. $\square$

Example 4.7 A typical example of accumulation is the linear-time version of reverse:

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

where

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

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

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

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 reduces to prove 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.

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

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 reduces 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 [19] that relates a fold with a higher-order fold.

Proposition 4.8 *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.

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

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

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

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

by initiality with accumulators we obtain that:

$$f \circ (\mathsf{fold}_F(h) \times \mathsf{id}_X) = \mathsf{afold}_F(k, \overline{\tau})$$

Therefore,

$$\mathsf{fold}_F(h)(t) = f(\mathsf{fold}_F(h)(t), e) = \mathsf{afold}_F(k, \overline{\tau})(t, e)$$

as desired. □

5 Downwards Accumulations

Downwards accumulations [13, 14, 15] are functions that label every node of a data structure with some function of its ancestors. Gibbons [15] presents a generic definition of downwards accumulation in terms of unfold (the operator dual to fold, see [16]) that uses an accumulating parameter to pass information downwards. In this section, we show that a generic definition of downwards accumulation can be given in terms of our notion of accumulation. More precisely, we show that a downwards accumulation can be written as an afold.

A downwards accumulation returns a data structure which is similar to that given as input but that has a label in every node. That makes necessary the introduction of a notion of *labelled variant* of a datatype.

Definition 5.1 ([6]) The **labelled variant** of a parameterized datatype DA induced by a bifunctor F is a datatype $D^{\mathcal{L}}A$ induced by a bifunctor G defined by $G(A,Y) = A \times F(1,Y)$. □

This means that $D^{\mathcal{L}}$ is defined by the type equation $D^{\mathcal{L}}A \cong A \times F(1, D^{\mathcal{L}}A)$. Note that each node of the labelled variant of a datatype carries one (and only one) label of type A.

Example 5.2

(i) In the case of lists, the functor G given by

$$\begin{array}{rcl} G(A,Y) & = & A \times L(1,Y) \\ & = & A \times (1 + 1 \times Y) \\ & \cong & A \times (1 + Y) \\ & \cong & A + A \times Y \end{array}$$

inducing a type of non-empty lists:

$$\mathsf{nelist}(A) = \mathsf{newrap}(A) \mid \mathsf{necons}(A \times \mathsf{nelist}(A))$$

(ii) In the case of leaf-labelled binary trees, the functor G is given by

$$\begin{array}{rcl} G(A,Y) & = & A \times B(1,Y) \\ & = & A \times (1 + Y \times Y) \\ & \cong & A + A \times Y \times Y \end{array}$$

inducing a type of so-called homogeneous binary trees [15]:

$$\mathsf{htree}(A) = \mathsf{hleaf}(A) \mid \mathsf{hnode}(A \times \mathsf{htree}(A) \times \mathsf{htree}(A))$$

(iii) In the case of binary trees with information in the nodes, the functor

$$\begin{array}{rcl} G(A,Y) & = & A \times T(1,Y) \\ & = & A \times (1 + Y \times 1 \times Y) \\ & \cong & A + A \times Y \times Y \end{array}$$

inducing the same type of homogeneous binary trees shown in the previous case.

□

Having defined the notion of labelled variant of a datatype, we are now ready to give a definition of downwards accumulation in terms of afold.

Definition 5.3 Given $\overline{\tau} : F_A C \times X \to F_A(C \times X)$, proper for accumulation, and $\rho : F(A,1) \times X \to B$, we define **downwards accumulation** to be the arrow $\mathsf{da}_{F_A}(\rho, \overline{\tau}) : DA \times X \to D^{\mathcal{L}}B$ given by

$$\mathsf{da}_{F_A}(\rho, \overline{\tau}) = \mathsf{afold}_{F_A}(in_{G_B} \circ \phi, \overline{\tau})$$

where

$$\begin{array}{rcl} \phi & = & F(A, D^{\mathcal{L}}B) \times X \\ & & \quad \searrow \langle F(\mathsf{id}_A, !) \times \mathsf{id}_X, \pi_1 \rangle \\ & & (F(A,1) \times X) \times F(A, D^{\mathcal{L}}B) \\ & & \quad \searrow \rho \times F(!, \mathsf{id}) \\ & & \underbrace{B \times F(1, D^{\mathcal{L}}B)}_{G(B, D^{\mathcal{L}}B)} \end{array}$$

□

Gibbons' version of downwards accumulation [15] carries a function of type $X \times F(A,1) \to B \times F(1,X)$ that takes an accumulator and the labels attached to a node of the input data structure and returns the label of type B that is attached to that node in the resulting data structure together with the value of the accumulators for the recursive calls. Our version, in contrast, splits up these actions into two functions

$$\overline{\tau} : F_A C \times X \to F_A(C \times X) \qquad\qquad \rho : F(A,1) \times X \to B$$

such that $\overline{\tau}$ computes the value of the accumulators for the recursive calls and $\rho : F(A,1) \times X \to B$ computes the label of type B to be attached to the node of the resulting data structure. Another difference we have with Gibbons' construction is that he needs to assume the existence of a polymorphic partial function

$$\mathsf{zip}_G : G(A,B) \times G(A',B') \to G(A \times A', B \times B')$$

that combines two values provided that they have the same shape. In our case that assumption is unnecessary.

Example 5.4 As a simple example of a generic function that can be defined in terms of downwards accumulation, Gibbons [15] presents a generic function $\mathsf{depths} : DA \to D^{\mathcal{L}}\,\mathsf{nat}$, which counts the number of ancestors of every node of a data structure. In terms of our version of downwards accumulation depths is defined by:

$$\mathsf{depths}_D(t) = \mathsf{dcount}_D(t, \mathsf{zero})$$

where

$$\begin{array}{rcl} \mathsf{dcount}_D & : & DA \times \mathsf{nat} \to D^{\mathcal{L}}\,\mathsf{nat} \\ \mathsf{dcount}_D & = & \mathsf{da}_{F_A}(\rho, \overline{\tau}) \end{array}$$

such that $\rho = \pi_2$ and $\overline{\tau} = \tau^{F_A} \circ (\mathsf{id} \times \mathsf{succ})$. Function dcount simply attaches the current value of the accumulator to the node and passes the accumulator's successor to the recursive calls. Let us see some instances:

(i) For lists, we have

$$\begin{aligned}\mathsf{dcount}(\mathsf{nil}, d) &= \mathsf{newrap}(d)\\ \mathsf{dcount}(\mathsf{cons}(a,\ell), d) &= \mathsf{necons}(d, \mathsf{dcount}(\ell, d+1))\end{aligned}$$

(ii) For leaf-labelled binary trees, we have

$$\begin{aligned}\mathsf{dcount}(\mathsf{leaf}(a), d) &= \mathsf{hleaf}(d)\\ \mathsf{dcount}(\mathsf{join}(t,u), d) &= \mathsf{hnode}(d, \mathsf{dcount}(t, d+1), \mathsf{dcount}(u, d+1))\end{aligned}$$

(ii) For binary trees with information in the nodes, we have

$$\begin{aligned}&\mathsf{dcount}(\mathsf{empty}, d)\\ &\qquad = \mathsf{hleaf}(d)\\ &\mathsf{dcount}(\mathsf{node}(t,a,u), d)\\ &\qquad = \mathsf{hnode}(d, \mathsf{dcount}(t, d+1), \mathsf{dcount}(u, d+1))\end{aligned}$$

□

Example 5.5

(i) If we modify function initsums (shown at the beginning of Section 4) so that it returns a value of type nonempty list instead of a value of type list (as it was declared), then we can define it in terms of a downwards accumulation: $\mathsf{initsums}(\ell) = \mathsf{isums}(\ell, \mathsf{zero})$, where

$$\begin{aligned}\mathsf{isums} &: \mathsf{nat}^* \times \mathsf{nat} \to \mathsf{nelist}(\mathsf{nat})\\ \mathsf{isums} &= \mathsf{da}_{L_{\mathsf{nat}}}(\pi_2, \overline{\tau}^{\mathsf{is}})\end{aligned}$$

The arrow $\overline{\tau}^{\mathsf{is}}$ coincides with the one used to define isums in Example 4.4.

(ii) In a similar manner, we can modify function asums (defined in Example 4.4) so that it now returns an homogeneous binary tree. This means that it will attach a label inclusive to an empty tree. This new version of asums can be defined in terms of a downwards accumulation: $\mathsf{asums}(t) = \mathsf{sdown}(t, \mathsf{zero})$, where

$$\begin{aligned}\mathsf{sdown} &: \mathsf{tree}(\mathsf{nat}) \times \mathsf{nat} \to \mathsf{hbtree}(\mathsf{nat})\\ \mathsf{sdown} &= \mathsf{da}_{T_{\mathsf{nat}}}(\pi_2, \overline{\tau}^{\mathsf{sd}})\end{aligned}$$

The arrow $\overline{\tau}^{\mathsf{sd}}$ coincides with the one used to define sdown in Example 4.4.

□

Now let us see some properties of downwards accumulations. Because downwards accumulations are a particular form of afold their properties are derived from those of afold. A proof of these laws can be found in [29].

DAcc-Fold Fusion

$$\mathsf{fold}_{G_B}(k) \circ \mathsf{da}_{F_A}(\rho, \overline{\tau}) = \mathsf{afold}_{F_A}(k \circ \phi, \overline{\tau})$$

DAcc-Map Fusion

$$D^{\mathcal{L}} f \circ \mathsf{da}_{F_A}(\rho, \overline{\tau}) = \mathsf{da}_{F_A}(f \circ \rho, \overline{\tau})$$

Map-DAcc Fusion For $f : A \to B$ and $DA = \mu F_A$,

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

Morph-DAcc Fusion For every $f : X \to X'$,

$$
\begin{aligned}
& F(\mathsf{id} \times f) \circ \overline{\tau} = \overline{\tau}' \circ (\mathsf{id} \times f) \\
\Rightarrow & \\
& \mathsf{da}_F(\rho, \overline{\tau}') \circ (\mathsf{id} \times f) = \mathsf{da}_F(\rho \circ (\mathsf{id} \times f), \overline{\tau})
\end{aligned}
$$

Except for dacc-map fusion, which is a corollary of dacc-fold fusion, the rest of the laws are specializations of laws for afold. This is a natural consequence of having defined downwards accumulation as an afold. For example, law dacc-fold fusion follows immediately from afold-fold fusion after observing that $in_{G_B} \circ \phi$ can be written as $\boldsymbol{T}(in_{G_B})$, being $\boldsymbol{T}(h) = h \circ \phi$ a transformer.

Example 5.6 The function $\mathsf{paths} : \mathsf{tree}(A) \to \mathsf{htree}(A^*)$, which labels each node with the list (in reverse order) of its ancestors, can be defined by:

$$\mathsf{paths} = \mathsf{pdown}(t, \mathsf{nil})$$

where

$$
\begin{aligned}
\mathsf{pdown} &: \mathsf{tree}(A) \times A^* \to \mathsf{htree}(A^*) \\
\mathsf{pdown} &= \mathsf{da}_{T_A}(\pi_2, \overline{\tau}^{\mathsf{pd}})
\end{aligned}
$$

and

$$
\begin{aligned}
\overline{\tau}^{\mathsf{pd}}(x, \ell) = \; & \textbf{case } x \textbf{ of} \\
& \quad \mathsf{inl}(u) \to \mathsf{inl}(u) \\
& \quad \mathsf{inr}(t, a, t') \to \mathsf{inr}((t, \mathsf{cons}(a, \ell)), a, (t', \mathsf{cons}(a, \ell)))
\end{aligned}
$$

That is,

$$
\begin{aligned}
\mathsf{pdown}(\mathsf{empty}, \ell) &= \mathsf{hleaf}(\ell) \\
\mathsf{pdown}(\mathsf{node}(t, a, t'), \ell) &= \mathsf{hnode}(\mathsf{pdown}(t, \mathsf{cons}(a, \ell)), \ell, \mathsf{pdown}(t', \mathsf{cons}(a, \ell)))
\end{aligned}
$$

Now, suppose we want to prove that:

$$\mathsf{asums} = \mathsf{htree}(\mathsf{sum}) \circ \mathsf{paths}$$

where $\mathsf{sum} = \mathsf{fold}([\mathsf{zero}, \mathsf{add}]) : \mathsf{nat}^* \to \mathsf{nat}$ is the function that adds the elements of a list of numbers. This reduces to prove that:

$$\mathsf{sdown}(t, \mathsf{zero}) = \mathsf{htree}(\mathsf{sum})(\mathsf{pdown}(t, \mathsf{nil}))$$

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

$$\mathsf{sdown} \circ (\mathsf{id} \times \mathsf{sum}) = \mathsf{htree}(\mathsf{sum}) \circ \mathsf{pdown}$$

By dacc-map fusion, we have that

$$\mathsf{htree}(\mathsf{sum}) \circ \mathsf{pdown} = \mathsf{da}_{T_{\mathsf{nat}}}(\mathsf{sum} \circ \pi_2, \overline{\tau}^{\mathsf{pd}})$$

whereas by morph-dacc fusion:

$$\mathsf{sdown} \circ (\mathsf{id} \times \mathsf{sum}) = \mathsf{da}_{T_{\mathsf{nat}}}(\pi_2 \circ (\mathsf{id} \times \mathsf{sum}), \overline{\tau}^{\mathsf{pd}})$$

Therefore, we have arrived at the same result in both sides, since $\pi_2 \circ (\mathsf{id} \times \mathsf{sum}) = \mathsf{sum} \circ \pi_2$. Morph-dacc fusion requires that

$$T_{\mathsf{nat}}(\mathsf{id} \times \mathsf{sum}) \circ \overline{\tau}^{\mathsf{pd}} = \overline{\tau}^{\mathsf{sd}} \circ (\mathsf{id} \times \mathsf{sum})$$

which can be verified by a simple calculation. □

6 Final Remarks

This work presented a generic definition of accumulations on inductive types. This was achieved by the introduction of a recursion operator on inductive types that encapsulates the action of accumulation in a natural transformation. The construction presented resolves some of the shortcomings of a previous version defined in [28]. Concretely, in [28] we attempted to specify in full detail the way in which accumulators are modified. As a consequence of being too specific, accumulations included useless cases, namely those corresponding to empty constructors. That problem has been overcome in the current version.

We also showed that the notion of downwards accumulation developed by Gibbons is a specific case of the notion of accumulation captured by the new operator. This demonstrates that downwards accumulations can also be defined by structural recursion.

Accumulations may also be defined in a domain-theoretic setting. In fact, every accumulation can be expressed as a hylomorphism [27]:

$$\mathsf{afold}_F(h, \overline{\tau}) = \mathsf{hylo}_{F \times \underline{X}}(h, \langle \overline{\tau}, \pi_2 \rangle \circ (out_F \times \mathsf{id}_X))$$

where $out_F : \mu F \to F \mu F$ is the inverse of in_F and for each $k : HA \to A$ and $g : B \to HB$, $\mathsf{hylo}_H(k, g) : B \to A$ is defined by

$$\mathsf{hylo}_H(k, g) = \mathsf{fix}(\phi) \quad \text{with} \quad \phi(f) = k \circ Hf \circ g$$

The following diagram makes the types explicit:

$$\begin{tikzcd}
F(\mu F \times X) \times X \arrow[r, "F\,\mathsf{afold}_F(h,\overline{\tau}) \times \mathsf{id}_X"] & FA \times X \arrow[dd, "h"] \\
F\mu F \times X \arrow[u, "\langle \overline{\tau}, \pi_2 \rangle"] & \\
\mu F \times X \arrow[u, "out_F \times \mathsf{id}_X"] \arrow[r, "\mathsf{afold}_F(h,\overline{\tau})"'] & A
\end{tikzcd}$$

The base functor of the hylomorphism is $GY = FY \times X$; it corresponds to the signature of a labelled variant of the input data structure (of signature F) which has a value of the accumulators attached to every node. A consequence of this representation is that the laws for afold (showed in Section 4) can be derived from standard laws for hylomorphism and by the addition of some strictness conditions in certain cases.

Every hylomorphism can be split into the composition of a fold after an unfold (see [27]): $\mathsf{hylo}_H(k, g) = \mathsf{fold}_H(k) \circ \mathsf{unfold}_H(g)$, where unfold is defined by

$$\mathsf{unfold}_H(g) = in_H \circ H\ \mathsf{unfold}_H(g) \circ g$$

In the case of afold, this means that

$$\mathsf{afold}_F(h, \overline{\tau}) \;=\; \mu F \times X \xrightarrow{\mathsf{unfold}_G(\langle \overline{\tau} \circ out_F, \pi_2 \rangle)} \mu G \xrightarrow{\mathsf{fold}_G(h)} A$$

This factorization makes the process of generation and consumption of the intermediate data structure (of signature G) explicit. In the generation phase, corresponding values of the accumulators are computed recursively (using $\overline{\tau}$) and attached to every node of the data structure. In the consumption phase, a result is computed from the intermediate data structure. To produce the result this process uses, in each step, the information contained in each node of the (original) data structure together with the attached values of the accumulators.

Our approach to accumulations avoids the use of higher-order folds. A consequence of this fact is that we can only represent accumulations that pass information strictly downwards. Clearly, there are other kind of accumulations that does not fit this scheme. For example, we cannot represent functions that use the extra parameter to provide a continuation for the result. A typical example is function $\mathsf{leaves} : \mathsf{btree}(A) \to [A^* \to A^*]$, which lists the leaves of a tree in the following manner:

$$\begin{aligned} \mathsf{leaves}\ (\mathsf{leaf}\ a)\ \ell &= \mathsf{cons}(a, \ell) \\ \mathsf{leaves}\ (\mathsf{join}(t, u))\ \ell &= \mathsf{leaves}\ t\ (\mathsf{leaves}\ u\ \ell) \end{aligned}$$

In this case the accumulator for the recursive call on the left subtree depends on the result of the recursive call on the right subtree, something not captured by our scheme. Accumulations of this kind can be written as a higher-order fold.

Acknowledgments

I would like to thank the anonymous referees for their detailed and helpful comments. Especial thanks go to Manuel Alcino Cunha for pointing out the representation of accumulations in terms of hylomorphisms. Diagrams were drawn using Paul Taylor's macros.

References

[1] R. Aubin. Mechanizing Structural Induction: Part I and II. *Theoretical Computer Science*, 9:329–362, 1979.

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

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

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

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

[6] R.S. Bird, P.F. Hoogendijk, and O. De Moor. Generic Programming with Relations and Functors. *Journal of Functional Programming*, 6(1):1–28, 1996.

[7] R.S. Boyer and J.S. Moore. *A Computational Logic.* Academic Press, 1979.

[8] R. M. Burstall and J. Darlington. A Transformation System for Developing Recursive Programs. *JACM*, 24(1):44–67, January 1977.

[9] R. Cockett and T. Fukushima. About Charity. Technical Report 92/480/18, University of Calgary, June 1992.

[10] R. Cockett and D. Spencer. Strong Categorical Datatypes I. In R.A.C. Seely, editor, *International Meeting on Category Theory 1991,* volume 13 of *Canadian Mathematical Society Conference Proceedings*, pages 141–169, 1991.

[11] M.M. Fokkinga. *Law and Order in Algorithmics.* PhD thesis, Universiteit Twente, The Netherlands, 1992.

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

[13] J. Gibbons. Upwards and Downwards Accumulations on Trees. In R.S. Bird, C.C. Morgan, and J.C P. Woodcock, editors, *Mathematics of Program Construction,* LNCS 669. Springer-Verlag, 1993.

[14] J. Gibbons. Polytypic Downwards Accumulations. In *Mathematics of Program Construction,* LNCS 1422. Springer-Verlag, 1998.

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

[16] J. Gibbons and G. Jones. The Under-Appreciated Unfold. In *Proc. 3rd. ACM SIGPLAN International Conference on Functional Programming.* ACM, September 1998.

[17] M.C. Henson. *Elements of Functional Programming.* Computer Science Texts. Blackwell Scientific Publications, 1987.

[18] 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.

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

[20] B. Jacobs. Parameters and Parameterization in Specification, using distributive categories. *Fundamenta Informaticae*, 24(3):209–250, 1995.

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

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

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

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

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

[26] Z. Manna and R. Waldinger. *The Deductive Foundations of Computer Programming.* Addison-Wesley, 1993.

[27] 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.

[28] 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.

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

[30] L.C. Paulson. *ML for the Working Programmer.* Cambridge University Press, Cambridge, UK, 1991.