“Explosive” Programming Controlled by Calculation

J.N. Oliveira

Dep. Informática, Universidade do Minho,
Campus de Gualtar, 4700 Braga, Portugal
Tel: 351+53+604470
EMAIL: jno@di.uminho.pt

Abstract. In the design of a functional library in the area of data-mining several algorithmic patterns have been identified which call for generic programming. Some of these have to do with flattening functions which arise in a particular group of hierarchical systems.
In this paper we describe our efforts to make such functionalities generic. We start by a generic inductive construction of the intended class of hierarchical types. We conclude by relating the structure of the relevant base-functors with the algebraic structure which is required by the generic flattening functionality, in particular concerning its “deforestation” towards a linearly complex implementation.
The instances we provide as examples include the widely known *bill of materials* “explode” operation.

1 Introduction

The definition of a function

$$f : B \longrightarrow A \tag{1}$$

can be regarded as a kind of “contract”: function f is committed to produce an A-value provided it is supplied with a B-value.

Such “functional contracts” can be of two kinds: (a) f intentionally loses information, because B is found too detailed and one wants to capture only the A-aspect of B-values — so, A is an abstraction of B (f is non-injective); (b) f faithfully converts data from the B-format to the A-format — so, f is injective and, in the limit, the two formats are the same (f is the identity).

Case (a) above is perhaps more interesting than (b) and supports the following aphorism about a facet of *functional programming*: it is the *art* [1] of transforming or losing information in a controlled and precise way. That is, the art of constructing the exact observation of data which fits in a particular context or requirement.

At the heart of this “data mining” discipline one finds many situations in which knowledge is extracted from a complex data structure and accumulated via a binary operator which provides for the intended abstraction (*e.g.* summing up the elements of a list). The algebraic structure which accommodates this kind of operation is the

[1] Which computer scientists wish to convert — and are converting — into *science*.

ubiquitous *monoid*. And, in fact, functional data-mining is nothing but a series of clever, highly scrutinized monoidal reductions of a complex input structure.

In the design of a CAMILA [2] functional library containing kernel data-mining operators we have met situations in which slightly more elaborate reduction algebras are required which resemble (but are less sophisticated than) *vector spaces* — something to be expected from the "metric" nature of data-mining.

This paper identifies a generic (*polytypic*) class of functions which provide for such metric reductions wherever the observed datatype is a recursive data structure which embodies a notion of hierarchy, captured by an appropriate class of *base functor* constructions. Instances of the provided functional abstraction are given which put together algorithms as far apart as, for instance, the *bill of materials* "explode" operation and the UNIX `tar` command. The paper includes a calculation which ports these results from functional to imperative programming media.

2 Context

Everybody is familiar with the concept of a function since the school desk. The functional intuition traverses mathematics from end to end because it has a solid semantics rooted on a clear-cut mathematical structure — the category **Fun** (also called **Set**) of "all" sets and set-theoretical functions.

Functional programming has a tradition of absorbing fresh results from theoretical computer science, algebra and category theory. One of the most significant advances over the last decade has been the so-called *functorial* approach to datatypes which originated mainly from [13], was popularized by [12] and reached the textbook format in [6]. A comfortable basis for explaining *polymorphism* [23], the "datatypes as functors" moto has proved beneficial at a higher level of abstraction, giving birth to *polytypism* [11].

Polymorphism and polytypism are steps of the same ladder, that of *generic programming* [5]. The main target of this fast evolving discipline is to raise the level of abstraction of the programming discourse in a way such that seemingly disparate programming techniques, algorithms *etc.* are unified into idealized, kernel programming notions.

Besides polymorphism and polytypism, *generic programming* is "polymediatic" in the sense that the same generic result crosses the boundaries of different programming media, or paradigms, simply by changing the category in which the result is interpreted. The `repmin` derivation of [7], for instance, is shown to yield a functional program if interpreted in **Fun** or to yield the "corresponding" logic program if interpreted in **Rel**, a generalization of **Fun** to set-theoretical relations which has received increasing attention within the mathematics of program construction community [2].

3 Motivation

In this paper we wish to contribute to the "datatypes as functors" trend by identifying and exploiting a particular way of building complex functional data-structures (*induc-*

[2] Cf. *e.g.* [4, 6]. See [16] for other categories worth a visit "beyond **Fun**".

tive datatypes) out of existing ones. To be more specific, we will invest in the structure of the *base functor* which underlies the definition of an inductive datatype.

Recall the typical Haskell-like definition of $cons$-lists:

```
data List a = nil | cons(a, List a)        (2)
```

According to the sandard semantics of inductive datatypes, this definition declares `List a` as a solution to domain equation

$$x = 1 + a \times x \tag{3}$$

which can be abbreviated to $x = \mathsf{F}\, x$ by introducing functor $\mathsf{F}\, x \stackrel{\text{def}}{=} 1 + a \times x$. For F to be properly defined, a should be a constant or fixed datatype, *e.g.* $a = I\!N$, the set of natural numbers. But, for polymorphism we wish (2) to express $cons$-lists of any type a; therefore, a binary functor $\mathsf{B}(a, b) \stackrel{\text{def}}{=} 1 + a \times b$ should be used instead, called the *base functor* of the definition, whereby (3) rewrites to $x = \mathsf{B}(a, x)$.

In general, the definition of an inductive, n-ary parametric datatype will be an equation of the form

$$X = \mathsf{B}(A_1, \ldots, A_n, X) \tag{4}$$

where base-functor B is of arity $n + 1$ and A_1 to A_n are type parameters.

It is clear that in B resides the "essence" of the datatype, that is to say, the *pattern* of recursion which determines its expressive power. A constructive theory of inductive types should invest in structurally building more and more elaborate patterns of recursion out of pre-existing ones. In the above terminology, this would mean a discipline for *scaling up* base-functors.

This paper investigates a particular scale-up manœuvre which enables us to construct *hierarchical* extensions to pre-existing types. The purpose of this work, which still rather experimental, is to find a generic way extending the functionality of the original type up to its hierarchical outcome.

With no loss of generality, we will focus our attention to binary or ternary base-functors, that is, to datatypes of the form

$$\mathsf{T}\, A = \mathsf{B}(A, \mathsf{T}\, A)$$

or

$$\mathsf{T}(A, B) = \mathsf{B}(A, B, \mathsf{T}(A, B))$$

One of our notation simplifications includes the use of the equality symbol in places where (as above) the isomorphism symbol "$\cong$" would be more correct. We will prefer $A^\star$ to $List\ A$ (2) and, concerning functional expression infix operator precedence, we will assume $\circ$ (function composition) to bind closer than $\times$ (products) and this to bind closer than $+$ (coproducts).

4 Illustration

In the personal computer age everybody has become acquainted with the standard file-system structure of *e.g.* UNIX or WINDOWS, which is made of directories (folders) which are in turn made of sub-directories, and so on. Such a structure, which can be visualized as a tree, is a form of hierarchical knowledge representation.

But, who still remembers the CPM file system, or the CDS ISIS file system by Intel, back to the 1970s? It was just a *flat* structure mapping file names to file attributes. So, it was clearly non-hierarchical. How do we express the kind of "improvement" on file system structuring which happened in the meantime?

Let I be a primitive datatype of file names (identifiers) and $File$ be the datatype which describes files (*e.g.* contents, attributes, *etc.*). A flat file system will be described by a finite partial function from I to $File$,

$$I \rightharpoonup File \tag{5}$$

(For A, B two given datatypes, read "$A \rightharpoonup B$" as the datatype of *finite partial functions* from A to B, that is, of relations $\sigma \subseteq B \times A$ such that $\sigma \circ \sigma^\circ$ is a subset of identity id_B; in other words, the As are *keys* which uniquely identify the Bs [3].)

In a hierarchical file system we have to upgrade (5) to something like

$$I \rightharpoonup (File + Directory)$$

where $Directory$ is again a $I \rightharpoonup (File + Directory)$ and so on. So we obtain a recursive datatype of shape

$$X = I \rightharpoonup (File + X) \tag{6}$$

To "measure" the improvement from to (5) to (6) we go parametric on $File$ and specify the flat version as functor $\mathsf{F}\,A = I \rightharpoonup A$. Then (6) becomes equation $X = \mathsf{F}(File + X)$ and, turning $File$ a parameter again, we obtain type

$$\mathsf{H}\,A = \mu X.\mathsf{F}(A + X)$$

In short, the generic pattern of the improvement is as follows: the hierarchical extension of a predefined (type) functor $\mathsf{F}\,A$ is the type functor $\mathsf{H}\,A$ which is obtained as solution to equation $X = \mathsf{F}(A+X)$. In other words, the base functor B of $\mathsf{H}\,A$ "reuses" the original functor F, *i.e.*, $\mathsf{B}(A, X) \stackrel{\mathrm{def}}{=} \mathsf{F}(A + X)$.

This hierarchical pattern is very common in practice. For instance, from the $cons$-list functor $A^\star$ we build the *generalized*-list functor which stems from equation

$$X = (A + X)^\star \tag{7}$$

that is, the Haskell datatype

```
data GList a = [ a | GList a ]
```

[3] In [6], partial functions are called *simple* arrows or *imps*.

One may wonder about the hierarchical extension of the identity functor $\mathsf{F}\,A = A$, which is easily shown to be (isomorphic to) $\mathsf{H}\,A = A \times I\!N$ (a natural number is added to every $a \in A$ specifying how deep a happens to occur in the hierarchy determined by $X = A + X$), or of every constant functor $\mathsf{F}\,A = K$, which degenerates in itself since, in this case, $\mathsf{F}(A + X) = K$.

Moving on to more interesting examples, we get hierarchical (nested) sets out of the powerset functor,

$$X = \mathcal{P}\,(A + X)$$

or abstract hierarchical type

$$X = (A + X) \rightharpoonup B \tag{8}$$

(for some constant type B), obtained as a companion of the hierarchical file system pattern by freezing parameter B in $A \rightharpoonup B$ instead of freezing A (which was turned to constant I in the file system example). The functorial behaviour of (8) requires some care because $A \rightharpoonup B$ is contravariant on A. But this is a very expressive datatype on top of which we will model, in the sequel, the *bill of materials* problem and its "part explosion" functionality (and thus the "explosive" qualifier in the title of the paper).

In the remainder of the paper we will be interested in specifying and calculating functions which browse hierarchical structures such as illustrated above, and extract information which will be used elsewhere. Such functions are naturally described as *hylomorphisms* [22]. In the section which follows we will review some concepts, definitions and notations which will be adopted throughout the paper [4]. Readers familiar with [22] or with textbook [6] may choose to skip it with no loss of continuity.

5 An overview of "types as functors"

Recall the declaration of an arbitrary function f given by expression (1). In many situations we know that B happens to be the least fixpoint $\mu\mathsf{F}$ of some given equation $X = \mathsf{F}\,X$. Intuitively, f is expected to be recursive.

Wherever it exists, $\mu\mathsf{F}$ is the carrier of the initial F-algebra $(\mu\mathsf{F},\ \mu\mathsf{F} \xleftarrow{\iota_F} \mathsf{F}\,\mu\mathsf{F})$ which, in short, we will identify just by writing ι_F. So it is natural to express f as an F-*catamorphism* (or "generic fold") of some F-algebra α on target type A:

$$\begin{tikzcd} \mu\mathsf{F} \arrow[d, "f = (\!|\alpha|\!)_\mathsf{F}"'] & \mathsf{F}\,\mu\mathsf{F} \arrow[l, "\iota_\mathsf{F}"'] \arrow[d, "\mathsf{F}\,(\!|\alpha|\!)_\mathsf{F}"] \\ A & \mathsf{F}\,A \arrow[l, "\alpha"] \end{tikzcd}$$

Because ι_F is initial, $f = (\!|\alpha|\!)_\mathsf{F}$ is the unique F-homomorphism from ι_F to α. Initiality provides catamorphisms with the expected (universal) properties, *e.g.* fusion

$$f \circ (\!|\alpha|\!)_\mathsf{F} = (\!|\beta|\!)_\mathsf{F} \quad \text{if} \quad f \circ \alpha = \beta \circ \mathsf{F}\,f \tag{9}$$

[4] Reference [20] contains a detailed account of all this terminology.

reflection,

$$(\!|\iota_{\mathsf{F}}|\!)_{\mathsf{F}} = id_{\mu\mathsf{F}} \tag{10}$$

and so on. Intuitively, $(\!|\alpha|\!)_{\mathsf{F}}$ captures the abstract notion of a $\mu\mathsf{F}$ "browser", "parser", or better: of the abstract F-induced recursion schema. Note that laws such as (9) and (10) express themselves independently of F. So we are talking about *higher-order* polymorphism, — that is, about *polytypism* [11].

In the same way we identified the source type B above with $\mu\mathsf{F}$, for some F, we may happen to identify the target type A with $\mu\mathsf{G}$, for some G:

$$\begin{tikzcd} \mu\mathsf{F} \arrow[d, "f"'] & \mathsf{F}\,\mu\mathsf{F} \arrow[l, "\iota_{\mathsf{F}}"'] \arrow[d, "\mathsf{F}\,f"] \\ \mu\mathsf{G} & \mathsf{F}\,\mu\mathsf{G} \arrow[l, "\alpha"] \end{tikzcd} \tag{11}$$

Now, it would be "unnatural" to ignore the initial G-algebra ι_{G}, which is the standard constructor of values of type $\mu\mathsf{G}$.

Clearly, f analyses input data according to the F-recursive pattern and synthesizes output data according to the G-recursive pattern — it behaves like a "protocol" between such patterns of recursion.

How can such a protocol be captured in this setting? An F to G natural transformation (and its *theorem for free!* [23]) seems the most "natural" device for this purpose. One is tempted to somehow "paste" ι_{G} into diagram (11),

$$\begin{tikzcd} \mu\mathsf{F} \arrow[d, "f"'] & & \mathsf{F}\,\mu\mathsf{F} \arrow[ll, "\iota_{\mathsf{F}}"'] \arrow[d, "\mathsf{F}\,f"] \\ \mu\mathsf{G} & \mathsf{G}\,\mu\mathsf{G} \arrow[l, "\iota_{\mathsf{G}}"] & \mathsf{F}\,\mu\mathsf{G} \arrow[l, "\ldots"] \end{tikzcd}$$

leaving it open how to fill the "..." arrow. First, let us illustrate this diagram with a typical "protocol" function, that which should enable us to compute the length of a finite sequence:

$$\begin{tikzcd} C^{\star} \arrow[d, "length"'] & & 1 + C \times C^{\star} \arrow[ll, "{[\,nil, cons\,]}"'] \arrow[d, "1+C\times length"] \\ \mathbb{N}_0 & 1 + \mathbb{N}_0 \arrow[l, "{[\,\underline{0}, suc\,]}"] & 1 + C \times \mathbb{N}_0 \arrow[l, "1+\pi_2"] \end{tikzcd}$$

where $nil = \underline{[\,]}$ (given a constant c, $\underline{c}$ means the "everywhere c" polymorphic function $\lambda x.c$ [20]), $cons$ is the usual operator and suc is the successor function.

In this example, the "..." arrow was filled with natural transformation $1 + \pi_2$,

$$\begin{tikzcd} A \arrow[d, "f"'] & 1 + C \times A \arrow[r, "(1+\pi_2)_A"] \arrow[d, "1+C\times f"'] & 1 + A \arrow[d, "1+f"] \\ B & 1 + C \times B \arrow[r, "(1+\pi_2)_B"'] & 1 + B \end{tikzcd}$$

which provides for the required recursion pattern “protocol”. So, the diagram can be enriched thus,

$$
\begin{tikzcd}
C^\star \arrow[d, "length"'] & 1 + C^\star \arrow[d, "1+length"'] & 1 + C \times C^\star \arrow[l, "1+\pi_2"] \arrow[ll, bend right, "{[\,nil, cons\,]}"'] \arrow[d, "1+C\times length"] \\
I\!N_0 & 1 + I\!N_0 \arrow[l, "{[\,\underline{0}, suc\,]}"] & 1 + C \times I\!N_0 \arrow[l, "1+\pi_2"]
\end{tikzcd}
$$

immediately delivering properties

$$
\begin{aligned}
length \circ nil &= \underline{0} \\
length \circ cons &= succ \circ length \circ \pi_2
\end{aligned}
$$

whatever path is followed up in the right-hand square of the diagram.

A computable definition of $length$ will pop out by closing the left-hand side square of the diagram with coalgebra $C^\star \xrightarrow{\gamma} 1 + C^\star$, which is fully determined by composing $(1 + \pi_2)$ with the inverse of $[\,nil, cons\,]$, *i.e.* the standard “destructor” of lists,

$$
\begin{aligned}
\omega &= [\,nil, cons\,]^{-1} \\
&= (! + \langle head, tail\rangle) \circ =_{[\,]}?
\end{aligned}
\tag{12}
$$

where ! is the unique arrow from $C^\star$ to 1, predicate $=_{[\,]}$ denotes equality test $\lambda l.(l = [\,])$ and $=_{[\,]}?$ is an instance of a guard $p?$ [6]. Then

$$
\begin{aligned}
\gamma &= (1 + \pi_2) \circ \omega \\
&= (! + tail) \circ =_{[\,]}?
\end{aligned}
$$

and, therefore,

$$
length \stackrel{\text{def}}{=} [\,\underline{0}, suc \circ length \circ tail\,] \circ =_{[\,]}?
$$

What have been the basic building blocks of this specification? *Input* F-coalgebra γ + *output* G-algebra $[\,\underline{0}, suc\,]$ + natural *transformation* $1 + \pi_2 : \mathsf{F} \longrightarrow \mathsf{G}$. These are precisely the components of a so-called *hylomorphism* triplet, as presented in [22]: given two $\mathbf{Set}$ (endo)functors F, G, an F-coalgebra $(B,\ B \xrightarrow{\beta} \mathsf{F}\,B\ )$, a G-algebra $(A,\ A \xleftarrow{\alpha} \mathsf{G}\,A\ )$ and a natural transformation $\nu : \mathsf{F} \longrightarrow \mathsf{G}$, we abbreviate by hylomorphism triplet $[\![\alpha, \nu, \beta]\!]_{\mathsf{F},\mathsf{G}}$, the morphism from B to A defined by the least fixpoint of equation

$f = \alpha \circ \nu_A \circ (\mathsf{F}\,f) \circ \beta$ *cf.* diagram

$$
\begin{tikzcd}
B \arrow[d, "f"'] \arrow[rr, bend left, "\beta"] & \mathsf{G}\,B \arrow[d, "\mathsf{G}\,f"'] & \mathsf{F}\,B \arrow[l, "\nu_B"] \arrow[d, "\mathsf{F}\,f"] \\
A & \mathsf{G}\,A \arrow[l, "\alpha"] & \mathsf{F}\,A \arrow[l, "\nu_A"]
\end{tikzcd}
$$

It is not always the case that we have ν as a natural transformation. So, in general, hylomorphisms are simply defined as pairs $[\![\gamma, \beta]\!]$ of an algebra γ and a coalgebra β of the same functor, *e.g.* $\gamma = \alpha \circ \nu$ concerning triplet $[\![\alpha, \nu, \beta]\!]_{\mathsf{F},\mathsf{G}}$. One of the advantages of reasoning in terms of triplets is the naive *deforestation* [22] which arises from ν's *theorem for free!*. This is meaningful wherever F and G are polynomial and the degree of G (*e.g.* linear lists) is stricly smaller than that of F (*e.g.* binary trees). Note that a catamorphism is a special case of a hylomorphism, for β the inverse of ι_{F}.

Many useful programming schemata arise from the hylomorphism construct even where no initial algebras are involved. For instance, take co-algebra

$$(A \rightharpoonup B) \xrightarrow{\beta} 1 + (A \times B) \times (A \rightharpoonup B)$$

of base functor $\mathsf{B}(A, B, X) = 1 + (A \times B) \times X$, defined by

$$\beta = (! + get) \circ =_{\perp}? \tag{13}$$

where $\perp$ denotes the totally undefined partial function and get is defined as follows:

$$\begin{array}{l} get\,\sigma \stackrel{\mathrm{def}}{=} let\ a \in dom\,\sigma \\ \qquad\qquad in\ \ ((a, \sigma\, a), \sigma \setminus \{a\}) \end{array}$$

Here $dom\,\sigma$ denotes the domain of definition of σ and $\sigma \setminus \{a\}$ denotes σ "domain-subtracted" by a [5]. Arrow β can be recognized as the standard "parser" for finite partial functions, and $[\![\gamma, \beta]\!]$, for some algebra $\gamma = [\ \gamma_1 , \gamma_2\ ]$, as the generic finite partial function "processor",

$$f\,\sigma \stackrel{\mathrm{def}}{=} \begin{cases} \sigma = \perp & \Rightarrow \gamma_1 \\ ((a, b), \sigma') = get\,\sigma \Rightarrow \gamma_2((a, b), f\,\sigma') \end{cases}$$

which we may re-write as follows:

$$\begin{array}{l} f\,\sigma \stackrel{\mathrm{def}}{=} if\ \ \sigma = \perp \\ \qquad\ \ then\ \ \gamma_1 \\ \qquad\ \ else\ \ let\ ((a, b), \sigma') = get\,\sigma \\ \qquad\qquad\quad in\ \ \gamma_2((a, b), f\,\sigma') \end{array} \tag{14}$$

Coalgebras ω (12) and β (13) will be relevant in the sequel. See [22, 21] for more about hylomorphism theory and [17, 20] for the application of all this to data refinement.

6 A study of (hierarchical) flattening

Recall the generalized-list datatype which satisfies domain equation (7), that is, type functor $\mathsf{H}\,A = (A + \mathsf{H}\,A)^{\star}$ which is the hierarchical extension of the $cons$-list functor

---

[5] Operators such as dom, $\setminus$ and others to come are typical of specification languages where the mapping (= finite partial function) datatype is primitive, *cf. e.g.* CAMILA [2], VDM-SL [8] *etc.*. In Haskell, this datatype is easily implemented in terms of lists of pairs [15].

Also note that, strictly speaking, get is a relation and one should switch to the broader category $\mathbf{Rel}$ of [6].

$\mathsf{F}\,A \stackrel{\mathrm{def}}{=} A^\star = 1 + A \times A^\star$. In the following $\mathsf{H}\,A$-instantiation of the polytypic function $flatten$ [10],

$$
\begin{aligned}
flatten &: \mathsf{H}\,A \longrightarrow A^\star \\
flatten\, l &\stackrel{\mathrm{def}}{=} \mathop{+\!\!+}_{x \leftarrow l} \left\{ \begin{array}{l} (x = i_1\,a) \Rightarrow [a] \\ (x = i_2\,l') \Rightarrow flatten\, l' \end{array} \right.
\end{aligned}
\tag{15}
$$

i_1 and i_2 are coproduct injections $A \xrightarrow{i_1} A + B \xleftarrow{i_2} B$ (*cf.* inl and inr in [6]) and notation $\mathop{+\!\!+}_{x \leftarrow l} \ldots$ indicates the iteration of binary list-concatenation ($x \mathbin{+\!\!+} y$) to a sequence of lists, that is,

$$
\begin{aligned}
\mathop{+\!\!+}\, [\,] &= [\,] \\
\mathop{+\!\!+}\, cons(x, l) &= x \mathbin{+\!\!+} (\mathop{+\!\!+}\, l)
\end{aligned}
\tag{16}
$$

hold. Combining (15) with (16) we obtain

$$
\begin{aligned}
flatten\,[\,] &= [\,] \\
flatten\,(cons(x,l)) &= \left\{ \begin{array}{l} (x = i_1\,a) \Rightarrow [a] \\ (x = i_2\,l') \Rightarrow flatten\, l' \end{array} \right. \mathbin{+\!\!+} flatten\, l
\end{aligned}
$$

as pointwise version of equation

$$flatten \circ [\, nil, cons\,] = [\, nil, \mathbin{+\!\!+} \circ ([\, wrap, flatten\,] \times flatten)\,]$$

where $wrap\, a = [a]$ and $nil = \underline{[\,]}$, as earlier on.

This equation is a simplification of the one which arises from commutative diagram

$$
\begin{tikzcd}
\mathsf{H}\,A \arrow[d, "flatten"] & (A + \mathsf{H}\,A)^\star \arrow[l, "\delta"'] & 1 + (A + \mathsf{H}\,A) \times (A + \mathsf{H}\,A)^\star \arrow[l, "{[\, nil, cons\,]}"'] & 1 + (A + \mathsf{H}\,A) \times \mathsf{H}\,A \arrow[l, "1 + id \times \delta^{-1}"'] \arrow[d, "1 + (A + flatten) \times flatten"'] \\
A^\star & 1 + A^\star \times A^\star \arrow[l, "{[\, nil, \mathbin{+\!\!+}\,]}"] & & 1 + (A + A^\star) \times A^\star \arrow[ll, "1 + [\, wrap, id\,] \times id"]
\end{tikzcd}
$$

where fixpoint isomorphism δ can be regarded as the identity, since we have not introduced any abstract syntax for $\mathsf{H}\,A$. Assuming this simplification, from this diagram we can express $flatten$ as either catamorphism $(\![\, [\, nil, \mathbin{+\!\!+} \circ ([\, wrap, id\,] \times id)\,]\,]\!)$ or as hylomorphism $[\![\, [\, nil, \mathbin{+\!\!+}\,] \circ \nu, \omega\,]\!]$,

$$
\begin{tikzcd}
\mathsf{H}\,A \arrow[rr, "\omega"] \arrow[d, "f"'] & & 1 + (A + \mathsf{H}\,A) \times \mathsf{H}\,A \arrow[d, "1 + (A + f) \times f"] \\
A^\star & 1 + A^\star \times A^\star \arrow[l, "{[\, nil, \mathbin{+\!\!+}\,]}"] & 1 + (A + A^\star) \times A^\star \arrow[l, "\nu"]
\end{tikzcd}
$$

where $\omega = [\, nil, cons \,]^{-1}$ is given by (12) and $\nu = 1 + [\, wrap, id \,] \times id$.

This $\mathsf{H}\,A$-instantiation of polytypic $flatten$ is relevant, for our purposes, in two ways. First, it works in the opposite direction on the hierarchical enrichment of $A^\star$ into $\mathsf{H}\,A$, as a kind of attempt to convert hierarchical lists into flat ones, obviously losing something (the sequence-nesting effect of $\mathsf{H}\,A$) but retaining something else (the As which can be found as leafs):

$$\mathsf{H}\,A \underset{lift = (i_1)^\star \text{ (\textit{i.e.}, extend, expand)}}{\overset{flatten \text{ (\textit{i.e.}, compress, reduce)}}{\rightleftarrows}} A^\star$$

Second, it is an instance of the generic functional pattern which captures monoidal reductions, which are very typical "data-mining" operations. A monoidal reduction is a hylomorphism whose target algebra is a monoid M,

$$\begin{tikzcd} T \arrow[rr, "\beta"] \arrow[d, "f"'] & & \mathsf{F}\,T \arrow[d, "\mathsf{F}\,f"] \\ M & 1 + M \times M \arrow[l, "{[\,\underline{\epsilon}, \theta\,]}"] & \mathsf{F}\,M \arrow[l, "\nu"] \end{tikzcd}$$

where ϵ is the unit and θ is the binary associative operator.

Let us now speculate about what should happen to $flatten$ in case we use bifunctor $\mathsf{F}(A, B) = (A \times B)^\star$ rather than $\mathsf{F}A = A^\star$ as starting point for the hierarchical extension. That is to say, the hierarchical type of interest is now defined by

$$X = ((A + X) \times B)^\star \tag{17}$$

To begin with, is this datatype any useful? Well, if we think of the Bs as natural numbers and of $(A \times I\!N)^\star$ as a model of lists of votes in the context of electing staff for a particular position within an organization, each pair (a, n) may indicate that candidate a has obtained n votes. Then extension (17) will capture the situation in which the organization is hierarchically structured and, while pairs $(i_1\, a, n)$ may have the same meaning as (a, n) above, pairs of the form $(i_2\, x, n)$ may mean "n is the weight of the votes to be found in x" (*e.g.* lecturer votes are twice as strong as those of students, those of full professors are worth twice those of lecturers, *etc.*).

In this situation, our guess for $flatten$ is the function which takes weights into account and computes the final votes:

$$flatten\,[\,] = [\,]$$
$$flatten\,(cons(x, l)) = \left\{ \begin{array}{l} x = (i_1\, a, b) \Rightarrow [(a, b)] \\ x = (i_2\, l', b) \Rightarrow b \otimes flatten\, l' \end{array} \right. +\!\!+\; flatten\, l$$

where $\otimes : I\!N \times (A \times I\!N)^\star \longrightarrow (A \times I\!N)^\star$ is given by $b \otimes l \stackrel{\text{def}}{=} [(a, b \times n) \mid (a, b) \leftarrow l]$. (A pointfree definition of $\otimes$ can be given more generally in terms of the *strength* ϕ [9] of the list functor,

$$B \times (A \times B)^\star \xrightarrow{swap} (A \times B)^\star \times B \xrightarrow{\phi} ((A \times B) \times B)^\star \tag{18}$$

followed by

$$((A \times B) \times B)^\star \xrightarrow{assocr^\star} A \times (B \times B)^\star \xrightarrow{(id \times (\times))^\star} (A \times B)^\star \qquad (19)$$

where names $swap$ and $assocr$ denote the same functions as in [6].)

We will denote by $\mathsf{H}(A, B)$ the hierarchical type defined by (17) whose base functor is $\mathsf{B}(A, B, X) \stackrel{\mathrm{def}}{=} ((A + X) \times B)^\star$. The diagram which justifies the above guess for flatten is, after some basic simplifications, as follows:

$$\begin{tikzcd}
\mathsf{H}(A,B) \arrow[rr, "\omega = [\, nil, cons \,]^{-1}"] \arrow[d, "flatten"] & & \mathsf{J}\,\mathsf{H}(A,B) \arrow[d, "\mathsf{J}\, flatten"'] \\
(A \times B)^\star & 1 + (A \times B)^\star \times (A \times B)^\star \arrow[l, "{[\, nil, +\!\!+ \,]}"] & \mathsf{J}\,(A \times B)^\star \arrow[l, "\nu"]
\end{tikzcd} \qquad (20)$$

where $\mathsf{J}\, X = 1 + ((A + X) \times B) \times X$. Arrow $\nu = 1 + [\, wrap, \otimes \circ swap \,] \circ distl \times id$, the one which prepares things for the monoidal reduction, arises from the composition which follows ($distl$ means "distribute left" [6]):

$$\begin{tikzcd}
1 + ((A + (A \times B)^\star) \times B) \times (A \times B)^\star \arrow[d, "1 + distl \times id"] \\
1 + (A \times B + (A \times B)^\star \times B) \times (A \times B)^\star \arrow[d, "1 + (id + swap) \times id"] \\
1 + (A \times B + B \times (A \times B)^\star) \times (A \times B)^\star \arrow[d, "1 + [\, wrap, \otimes \,] \times id"] \\
1 + (A \times B)^\star \times (A \times B)^\star
\end{tikzcd}$$

The only ingredient new in the move from $\mathsf{H}\, A$ to $\mathsf{H}(A, B)$ is the addition of operator $\otimes$ to the target monoidal structure $M = (A \times B)^\star, \epsilon = [\,]$ and $\theta = +\!\!+$. Note that operator $\otimes$ is "external" to the monoid — it bears functionality $\otimes : B \times M \longrightarrow M$.

Some intuition about this operator will arise next, in the context of another instantiation of $flatten$, this time for base functor $\mathsf{B}(A, B, X) \stackrel{\mathrm{def}}{=} (A + X) \rightharpoonup B$ of equation (8), that is, for hierarchical type $\mathsf{H}(A, B) = (A + H(A, B)) \rightharpoonup B$:

$$\begin{aligned}
flatten &: \mathsf{H}(A, B) \longrightarrow (A \rightharpoonup B) \\
flatten\, \sigma &\stackrel{\mathrm{def}}{=} \bigoplus_{x \in dom\, \sigma} \left\{ \begin{array}{l} (x = i_1\, a) \Rightarrow \begin{pmatrix} a \\ \sigma\, x \end{pmatrix} \\ (x = i_2\, \sigma') \Rightarrow (\sigma\, x) \otimes flatten\, \sigma' \end{array} \right.
\end{aligned} \qquad (21)$$

(Notation $\begin{pmatrix} a \\ b \end{pmatrix}$ is used instead of pair (a, b) to emphasize the applicative nature of partial maps.) For $B = I\!N$, $A \rightharpoonup B$ becomes $A \rightharpoonup I\!N$, the type of data structures which associate numbers to the elements present in their domain. So $A \rightharpoonup I\!N$ models *multisets*, that is, sets in which membership extends to multiplicity. (For $B = 1$, we have $A \rightharpoonup 1 \cong \mathcal{P}\, A$ [20] meaning, of course, that "normal" sets can be represented by "mono" multisets.) Moreover, $\mathsf{H}(A, I\!N)$ models the hierarchical structure of a production database, for instance the production tree of some electronic equipment: for A the datatype of atomic components, $(A + \mathsf{H}(A, I\!N)) \rightharpoonup I\!N$ tells us about the quantities involved in production, either of atomic components (A) or sub-equipments ($\mathsf{H}(A, I\!N)$), and so on.

By inspection, $flatten$ (21) models the "bill of materials" calculation, also called "part explosion". We will refer to this instance of $flatten$ as the $explode$ function. In this context, $\oplus : (A \rightharpoonup I\!N) \times (A \rightharpoonup I\!N) \longrightarrow (A \rightharpoonup I\!N)$ is easily guessed as multiset *union*,

$$v \oplus r = v \dagger r \dagger \begin{pmatrix} a \\ v\,a + r\,a \end{pmatrix}_{a \in (dom\, v) \cap (dom\, r)}$$

where $v \dagger r$ means the *overwriting* of map v by map r, and $\otimes : I\!N \times (A \rightharpoonup I\!N) \longrightarrow (A \rightharpoonup I\!N)$ is such that

$$n \otimes v = \underbrace{v \oplus \cdots \ldots \oplus v}_{n \text{ times}}$$

This brings something else to mind: if we identify A with the dimensions of a *vector space*, then each "multiset" $v \in A \rightharpoonup I\!N$ may be understood as a vector. For instance, 3-dimensional vector $v = 2x + 3y + z$ will be described by multiset $m = \begin{pmatrix} x & y & z \\ 2 & 3 & 1 \end{pmatrix}$ under the convention that nullary dimensions are omitted from the multiset, *e.g.* $2x + 0y + z$ simply represented by $\begin{pmatrix} x & z \\ 2 & 1 \end{pmatrix}$. Under this analogy, $\oplus$ becomes vectorial sum and $\otimes$ becomes external (scalar) multiplication,

$$n \otimes v = (A \rightharpoonup \times_n)v \tag{22}$$

where $\times_n$ denotes $curry\,(\times)\,n$. Later calculations in this paper (see section 7) will require $\otimes$ to distribute over $\oplus$, partly justifying the analogy. But, clearly, what we are asking here from a vector space is far less it can offer (inverses, cancellation *etc.*).

The diagram which captures $explode$ as a hylomorphism is very similar to (20) provided that $(A \times B)^\star$ is replaced by $A \rightharpoonup I\!N$, nil becomes $\underline{\perp}$, $+\!\!+$ becomes $\oplus$, $wrap$ becomes $\lambda(a, b).\begin{pmatrix} a \\ b \end{pmatrix}$, $\otimes$ is given by (22) and coalgebra ω becomes β (13).

To complete our $flatten$ing trip across the hierarchical structures considered in this paper, we go back to the very first one — the hierarchical file system datatype (6). Let us first rewrite its definition in a way consistent with the above examples

$$\mathsf{H}(A, B) = B \rightharpoonup (A + \mathsf{H}(A, B))$$

where B is the "file identifier" datatype — say $B = String$ — and A abstracts the "file information" datatype. Intuitively, the compressing effect of $flatten : \mathsf{H}(A, B) \longrightarrow$

$(B \rightharpoonup A)$ is now bound to work over B, thus in the contravariant parameter of the base functor. So, it will help if we define $\otimes$ functorially over an injective operator. On the other hand, the accumulation operator of the target monoid now has nothing to do with multiset union — the aggregation of *consistent* (*e.g.* domain disjoint) partial mappings via the union operator $\cup$, which forms a monoid coupled with $\perp$, is a possible choice.

The $B = String$ instantiation provides a hint for $\otimes$. Strings are sequences of characters in Haskell, for instance. So, why not define $\otimes : String \times (String \rightharpoonup A)$ as follows,

$$b \otimes \sigma = ((\lambda s. b \mathbin{+\!\!+} s) \rightharpoonup A)\sigma \quad ?$$

The picture will be complete if we add the conspicuous `"/"` string to the concatenated strings, *i.e.* $b \mathbin{+\!\!+} \texttt{"/"} \mathbin{+\!\!+} s$ extending $b \mathbin{+\!\!+} s$. Putting things together, it is easy to see that what we have just been considering is the UNIX-typical `tar` command,

$$\begin{aligned} tar &: \mathsf{H}(A,B) \longrightarrow (B \rightharpoonup A) \\ tar\,\sigma &\stackrel{\text{def}}{=} \bigcup_{b \in dom\,\sigma} \; let\ x = \sigma\, b \\ &\qquad\qquad in \begin{cases} (x = i_1\, a) \Rightarrow \begin{pmatrix} b \\ a \end{pmatrix} \\ (x = i_2\, \sigma') \Rightarrow b \otimes tar\,\sigma' \end{cases} \end{aligned}$$

which is justified by diagram

$$\begin{tikzcd} \mathsf{H}(A,B) \arrow[rr, "\beta' = (1 + swap \times id) \circ \beta"] \arrow[d, "tar"] & & 1 + ((A + \mathsf{H}(A,B)) \times B) \times \mathsf{H}(A,B) \arrow[d, "1 + ((A + tar) \times B) \times tar"'] \\ (B \rightharpoonup A) & 1 + (B \rightharpoonup A) \times (B \rightharpoonup A) \arrow[l, "{[\,\perp, \cup\,]}"] & 1 + ((A + (B \rightharpoonup A)) \times B) \times (B \rightharpoonup A) \arrow[l, "\nu"] \end{tikzcd}$$

where $\nu = 1 + [\, wrap, \otimes \,] \circ distl \times id$.

To wrap things up, we are ready to present the generic function of which three instantiations have been considered. Its pre-requisites are as follows:

1. A "flat" two-parameter datatype $\mathsf{F}(A,B)$ defined as a fixpoint of linear domain equation $X = 1 + (A \times B) \times X$.
2. Its hierarchical extension, $\mathsf{H}(A,B)$, which is defined as a fixpoint of domain equation

$$X = \mathsf{F}((A + X), B)$$

3. A "wrap up" arrow $\mathsf{F}(A,B) \xleftarrow{wrap} A \times B$ which embeds the product $A \times B$ into the flat datatype.
4. A target monoid algebra whose carrier is the flat type,

$$\mathsf{F}(A,B) \xleftarrow{[\,\epsilon, \theta\,]} 1 + \mathsf{F}(A,B) \times \mathsf{F}(A,B)$$

enriched with a "scalar multiplication" operator [6],

$$\mathsf{F}(A,B) \xleftarrow{\otimes} B \times \mathsf{F}(A,B)$$

which, as we shall see briefly, extends polymorphically to $\mathsf{H}(A,B)$.

Then the generic function we have been investigating is given by hylomorphism $[\![[\, \underline{\epsilon}, \theta \,] \circ \nu, \beta]\!]$ depicted by

$$\begin{tikzcd}
\mathsf{H}(A,B) \arrow[rr, "\beta"] \arrow[d, "f"] & & \mathsf{J}\,\mathsf{H}(A,B) \arrow[d, "\mathsf{J}\, f"'] \\
\mathsf{F}(A,B) & 1 + \mathsf{F}(A,B) \times \mathsf{F}(A,B) \arrow[l, "{[\, \underline{\epsilon}, \theta \,]}"] & \mathsf{J}\,\mathsf{F}(A,B) \arrow[l, "\nu"]
\end{tikzcd} \tag{23}$$

where $\mathsf{J}\,X = 1 + ((A + X) \times B) \times X$, β is the appropriate input "parser" and $\nu = 1 + [\, wrap, \otimes \circ swap \,] \circ distl \times id$.

We observe that J is polynomial of degree $n = 2$, as can be checked by converting it to canonical form [13], and so f is quadratic in execution time. Our final concern in this paper will consist of deriving a linear implementation, a program calculation exercise which will but confirm our earlier intuitions about the $[\, \epsilon, \theta, \otimes \,]$ algebraic structure required by the overall "flattening" reduction.

7 Deriving a linear implementation

Our purpose is to try and find an arrow ν' able to provide a linear recursive path alternative to $\nu \circ \mathsf{J}\, f$ in diagram (23):

$$\begin{tikzcd}
\mathsf{H}(A,B) \arrow[rr, bend left, "\beta"] \arrow[d, "f"] & 1 + \mathsf{F}(A,B) \times \mathsf{H}(A,B) \arrow[d, "1+\mathsf{F}(A,B)\times f"] & \mathsf{J}\,\mathsf{H}(A,B) \arrow[l, "\nu'"] \arrow[d, "\mathsf{J}\, f"'] \\
\mathsf{F}(A,B) & 1 + \mathsf{F}(A,B) \times \mathsf{F}(A,B) \arrow[l, "{[\, \underline{\epsilon}, \theta \,]}"] & \mathsf{J}\,\mathsf{F}(A,B) \arrow[l, "\nu"]
\end{tikzcd}$$

Natural isomorphism $(A \times B) + (C \times B) \xleftarrow{distl} (A + C) \times B$ will be taken into account, as well as fact

$$[\, g, h \,] \times f = [\, g \times f, h \times f \,] \circ distl \tag{24}$$

which is easily proved by first expressing $distl$ in terms of $swap$ and $distr$ and then using $f \times [\, g, h \,] = [\, f \times g, f \times h \,] \circ distr$ (see exercise 3.28 in [6]). In detail, the reasoning is as follows:

---

[6] Altogether, we might have written $\mathsf{F}(A,B) \xleftarrow{[\, \epsilon, \theta, \otimes \,]} 1 + \mathsf{F}(A,B) \times \mathsf{F}(A,B) + B \times \mathsf{F}(A,B)$.

$$
\begin{aligned}
f &= [\,\underline{\epsilon}, \theta\,] \circ (1 + [\,wrap, \otimes \circ swap\,] \circ distl \times id) \circ (1 + ((id + f) \times id) \times f) \circ \beta \\
&= \quad \{ \text{ bifunctors } + \text{ and } \times \} \\
&\quad [\,\underline{\epsilon}, \theta\,] \circ (1 + [\,wrap, \otimes \circ swap\,] \circ distl \circ ((id + f) \times id) \times f) \circ \beta \\
&= \quad \{ \text{ } distl \text{ is natural } \} \\
&\quad [\,\underline{\epsilon}, \theta\,] \circ (1 + [\,wrap, \otimes \circ swap\,] \circ (id + f \times id) \circ distl \times f) \circ \beta \\
&= \quad \{ \text{ } +\text{-absorption and } swap \text{ is natural } \} \\
&\quad [\,\underline{\epsilon}, \theta\,] \circ (1 + [\,wrap, \otimes \circ (id \times f) \circ swap\,] \circ distl \times f) \circ \beta \\
&= \quad \{ \text{ fact (26) below and identity of composition } \} \\
&\quad [\,\underline{\epsilon}, \theta\,] \circ (1 + [\,wrap, f \circ \otimes \circ swap\,] \circ distl \times f \circ id) \circ \beta \\
&= \quad \{ \text{ bifunctor } \times \} \\
&\quad [\,\underline{\epsilon}, \theta\,] \circ (1 + ([\,wrap, f \circ \otimes \circ swap\,] \times f) \circ (distl \times id)) \circ \beta \\
&= \quad \{ \text{ fact (24) } \} \\
&\quad [\,\underline{\epsilon}, \theta\,] \circ (1 + [\,wrap \times f, f \circ \otimes \circ swap \times f\,] \circ distl \circ (distl \times id)) \circ \beta \\
&= \quad \{ \text{ identity of composition and bifunctor } \times \} \\
&\quad [\,\underline{\epsilon}, \theta\,] \circ (1 + [\,wrap \times f, (f \times f) \circ (\otimes \circ swap \times id)\,] \circ distl \circ (distl \times id)) \circ \beta \\
&= \quad \{ \text{ bifunctor } + \} \\
&\quad [\,\underline{\epsilon}, \theta\,] \circ (1 + [\,wrap \times f, (f \times f) \circ (\otimes \circ swap \times id)\,]) \circ (1 + distl \circ (distl \times id)) \circ \beta \\
&= \quad \{ \text{ } +\text{-absorption, } +\text{-fusion and fact (27) below } \} \\
&\quad [\,\underline{\epsilon}, [\,\theta \circ (wrap \times f), \underbrace{f \circ \theta \circ (\otimes \circ swap \times id)}_{\rho}\,]\,] \circ \underbrace{(1 + distl \circ (distl \times id)) \circ \beta}_{\beta'} \\
&= \quad \{ \text{ introducing abbreviations } \rho \text{ and } \beta' \} \\
&\quad [\,\underline{\epsilon}, [\,\theta \circ (wrap \times f), \rho\,]\,] \circ \beta' \\
&= \quad \{ \text{ forcing } \epsilon \text{ in, which is the unit of } \theta \} \\
&\quad [\,\underline{\epsilon}, [\,\theta \circ (wrap \times f), \theta \circ \langle \underline{\epsilon}, \rho \rangle\,]\,] \circ \beta' \\
&= \quad \{ \text{ reverse } +\text{-fusion followed by exchange law } \} \\
&\quad [\,\underline{\epsilon}, \theta \circ \langle [\,wrap \circ \pi_1, \underline{\epsilon}\,], [\,f \circ \pi_2, \rho\,] \rangle\,] \circ \beta' \\
&= \quad \{ \text{ expansion of } \rho \text{ followed by reverse } +\text{-fusion } \} \\
&\quad [\,\underline{\epsilon}, \theta\,] \circ (1 + \langle [\,wrap \circ \pi_1, \underline{\epsilon}\,], f \circ [\,\pi_2, \theta \circ (\otimes \circ swap \times id)\,] \rangle) \circ \beta' \\
&= \quad \{ \text{ reverse } \times\text{-absorption and bifunctor } + \}
\end{aligned}
$$

$$
\begin{array}{ll}
& [\,\underline{\epsilon}, \theta\,] \circ (1 + id \times f) \circ (1 + \langle [\,wrap \circ \pi_1, \underline{\epsilon}\,], [\,\pi_2, \theta \circ (\otimes \circ swap \times id)\,] \rangle) \circ \beta' \\
= & \quad \{ \text{ expanding } \beta' \text{ and indroducing } \nu' \ \} \\
& [\,\underline{\epsilon}, \theta\,] \circ (1 + id \times f) \circ \nu' \circ \beta
\end{array}
$$

From this reasoning we extract

$$
\nu' = (1 + \langle [\,wrap \circ \pi_1, \underline{\epsilon}\,], [\,\pi_2, \theta \circ (\otimes \circ swap \times id)\,] \rangle) \circ (1 + distl \circ (distl \times id))
$$

which is equivalent to

$$
\nu' = 1 + \langle [\,wrap \circ \pi_1 \circ distl, \underline{\epsilon}\,], \theta \circ [\,\underline{\epsilon}, \otimes \circ swap\,] \times id \rangle \circ (distl \times id) \tag{25}
$$

and is central to the overall "deforestation" effect. Of the two facts required above,

$$
f \circ \otimes = \otimes \circ (id \times f) \tag{26}
$$

$$
f \circ \theta = \theta \circ (f \times f) \tag{27}
$$

which can be justified by fixpoint induction arguments [18], the first one is the most interesting because it bears semantic implications: its proof requires properties

$$
b \otimes (x \,\theta\, y) = (b \otimes x) \,\theta\, (b \otimes y) \tag{28}
$$

and

$$
(n \star b) \otimes x = n \otimes (b \otimes x) \tag{29}
$$

to hold, where $B \xleftarrow{\ \star\ } B \times B$ is the *internal multiplication* operator which underlies the generic definition of scalar multiplication, recall (18) and (19):

$$
B \times \mathsf{F}(A,B) \xrightarrow{swap} \mathsf{F}(A,B) \times B \xrightarrow{\ \varphi\ } \mathsf{F}(A, B \times B) \xrightarrow{\mathsf{F}(id,\star)} \mathsf{F}(A,B) \tag{30}
$$

For $\mathsf{F}(A,B) = A \rightharpoonup I\!N$, for instance, $\star$ is the product of two natural numbers and *strength*-like arrow φ is given by $\varphi(\sigma, n) = (A \rightharpoonup \lambda m.m \star n)\,\sigma$.

Facts (28) and (29) can be found in the axiomatization of *vector spaces* in Universal Algebra compendia. Adopting the full axiomatization would turn B into a field (of scalars) and $\mathsf{F}(A,B)$ into a commutative group (of vectors). Another operation would become available, that of addition of scalars (say $+$), which, despite being absent from our reasoning, would make perfect sense in our context, including its axiom

$$
(n + m) \otimes x = (n \otimes x) \,\theta\, (m \otimes x) \tag{31}
$$

In the *bill of materials* context, for instance, $+$ would be natural number addition, while in the *file system* context it would bring us into regular expression algebra.

8 Going imperative

Knowing that ϵ, θ form a monoid makes it possible to convert the linear version of f derived in the previous section into a `while`-loop, via the technique of *accumulation parameter* introduction [6]. We omit the details, which can be found in [18, 20], and only present the outcome of the reasoning leading to an iterative version of *explode* in the *bill of materials* context, written in a "pseudo-C"-like notation:

```
{ Parts     r = ⊥;
  Structure ff = y;
  while     (ff != ⊥)
            { Unit x = get(dom ff);
              Quantity n = ff x;
              r = r ⊕ { (x==i1 k) ⇒ (k n) ;
                       { (x==i2 y') ⇒ ⊥
              ff = ff \{x} ⊕ { (x == i1 k)  ⇒ ⊥           ;
                              { (x == i2 y') ⇒ n ⊗ y'
            };
}
```

Datatype identifiers `Parts`, `Unit` and `Structure` refer to $A \rightharpoonup I\!N$, $A + \mathsf{H}(A, I\!N)$ and $A + \mathsf{H}(A, I\!N) \rightharpoonup I\!N$, respectively. Variable y contains the input and the result is delivered into variable r. A bit of `if-then-else` logic finally turns this into something even simpler:

```
{ Parts r = ⊥;
  Structure ff = y ;
  Quantity n ;
  Unit x;
  while (ff != ⊥)
    { x = get(dom ff) ;
      n = ff x;
      ff = ff \{x};
      if (x == i1 k) r = r ⊕ (k n) ;
      if (x == i1 y') ff = ff ⊕ (n ⊗ y');

    }
}
```

Report [1] describes the embedding of this code into the (functional) rapid prototyping environment of the CAMILA toolkit.

9 Conclusions and current work

Some years ago we asked our students of the *Formal Methods* course at Minho to contribute to the design of a *data mining* library which should become available in ORACLE database technology after a series of careful steps: formal specification, rapid prototyping (in the CAMILA functional language) and calculation following a data refinement calculus [17].

A group assigned to the *bill of materials* functionality produced a correct and fairly compact ORACLE implementation of *part explosion*, despite the fact that their derivation included false steps (!). Their project report was kept as a good sample of how cumbersome pointwise reasoning may happen to be in the software design field, but also as a sample of good programming intuition.

By reworking and correcting their calculation, this time in the pointfree style, we became aware of how close they were to other groups calculating seemingly disparate functionalities, and of the need to redesign the library in a polytypic style.

This paper describes part of this later work, covering two main points: first, the identification and specification of a particular class of *general hierarchical systems* [3, 19] as a base-functor construction; second, a study of how polytypic $flatten$ evolves towards $explode$ once one more parameter is added to the hierarchical construction which, as we have seen, is accompanied by the move from *monoids* to *vector spaces* as target reduction algebras.

However, this is work in progress and the results are incomplete and unsatisfactory in several respects. First of all, we are still far from a truly polytypic characterization of our hierarchical type construction and of its functionality. Some attention has been paid to the move towards arbitrary "flat" datatypes $\mathsf{F}(A, B)$ but more work is needed concerning exponentials and contravariance [14]. We have also ignored the key issue of guaranteeing that our hierarchical arbitrary types do in fact exist as least fixpoints of their equations. Moreover, other hierarchical "extensors" might be considered (*e.g.* based on $A + X^n$ rather than on $A + X$) and indeed many hierarchical types exist which are not covered by our constructions. Last but not least, the nondeterminism inherent in some of the types considered in this paper requires a move from $\mathbf{Fun}$ to $\mathbf{Rel}$ [6].

References

1. J. J. Almeida, L. S. Barbosa, F. L. Neves, and J. N. Oliveira. Bringing CAMILA and SETS together — the `bams.cam` and `ppd.cam` CAMILA Toolset demos. Technical report, DI/UM, Braga, December 1997. Technical Report, 45 p.
2. J. J. Almeida, L. S. Barbosa, F. L. Neves, and J. N. Oliveira. CAMILA: Prototyping and refinement of constructive specifications. In M. Johnson, editor, *Algebraic Methodology and Software Technology*, pages 554–559. Springer LNCS, December 1997. 6th International Conference, AMAST'97, Sydney, Australia, 13–17 December 1997, Proceedings.
3. Micheal Mac an Airchinnigh. Some reflections on mathematics education for formal methods. Technical report, University of Dublin, 1996.
4. R. C. Backhouse and P. F. Hoogendijk. Elements of a relational theory of datatypes. In S.A. Schuman, B. Möller, and H.A. Partsch, editors, *Formal Program Development*, number 755 in Lecture Notes in Computer Science, pages 7–42. Springer, 1993.

5. R. C. Backhouse and T. Sheeard (org.). WGP'98 — Workshop on Generic Programming, 1998. Marstrand, Sweden, 18th June, 1998 (`http://www.cse.ogi.edu/PacSoft/conf/wgp/`).
6. R. Bird and O. de Moor. *Algebra of Programming*. Series in Computer Science. Prentice-Hall International, 1997. C. A. R. Hoare, series editor.
7. O. de Moor. An exercise in polytypic programming: repmin. Technical report, Oxford University, September 1996.
8. J. Fitzgerald and P.G. Larsen. *Modelling Systems: Practical Tools and Techniques*. Cambridge University Press, 1^{st} edition, 1997.
9. P. Hoogendijk and O. de Moor. What is a data type? Technical Report 96/16, Eindhoven University of Technology and Oxford PRG, August 1996.
10. P. Jansson and J. Jeuring. Polylib — a library of polytypic functions. In *Workshop on Generic Programming (WGP'98), Marstrand, Sweden*, 1998.
11. J. Jeuring and P. Jansson. Polytypic programming. In *Advanced Functional Programming*, number 1129 in Lecture Notes in Computer Science. Springer, 1996.
12. G. Malcolm. Data structures and program transformation. *Science of Computer Programming*, 14:255–279, 1990.
13. E. G. Manes and M. A. Arbib. *Algebraic Approaches to Program Semantics*. Texts and Monographs in Computer Science. Springer-Verlag, 1986. D. Gries, series editor.
14. E. Meijer and G. Hutton. Bananas in space: Extending fold and unfold to exponential types. In S. Peyton Jones, editor, *Proceedings of Functional Programming Languages and Computer Architecture (FPCA95)*, 1995.
15. P. Mukherjee. Automatic translation of VDM-SL specifications into Gofer. In J. Fitzgerald, C. B. Jones, and P. Lucas, editors, *FME'97: Industrial Applications and Strengthened Foundations of Formal Methods*, number 1313 in Lecture Notes in Computer Science, pages 258–277. Springer, 1997.
16. D. A. Naumann. Beyond fun: Order and membership in polytypic programming. In J. Jeuring, editor, *MPC'98: Mathematics of Program Construction*, number 1422 in Lecture Notes in Computer Science, pages 286–314. Springer, 1998.
17. J. N. Oliveira. Software reification using the SETS calculus. In *Proc. of the BCS FACS 5th Refinement Workshop, Theory and Practice of Formal Software Development, London, UK*, pages 140–171. Springer-Verlag, 8–10 January 1992. .
18. J. N. Oliveira. *Métodos Formais de Programação*. University of Minho, 4^{th} edition, 1997. Textbook (489 p. in Portuguese [7]). English version under preparation at the time of writing.
19. J. N. Oliveira. University education in Formal Methods - Report on the Minho experience, 1997. Tutorial. Training & Education Workshop, FME'97, Graz, Austria, 15-19 September.
20. J. N. Oliveira. A data structuring calculus and its application to program development, May 1998. Lecture Notes of M.Sc. Course (150 p.[8]). Maestria em Ingeneria del Software, Departamento de Informatica, Facultad de Ciencias Fisico-Matematicas y Naturales, Universidad de San Luis, Argentina.
21. A. Pardo. Monadic corecursion — definition, fusion laws, and applications. *Electronic Notes in Theoretical Computer Science*, 11, 1998.
22. A. Takano and E. Meijer. Shortcut to deforestation in calculational form. In *Proc. FPCA'95*, 1995.
23. P. Wadler. Theorems for free! In *4th International Symposium on Functional Programming Languages and Computer Architecture*, London, Sep. 1989. ACM.

[7] 846K gzipped PS file available from `http://www.di.uminho.pt/~jno/ps/mfp.ps.gz`.
[8] 390K gzipped PS file available from `http://www.di.uminho.pt/~jno/ps/sanl98.ps.gz`.