On the Design of a *Periodic* Table of VDM Specifications

J.N. Oliveira ⋆

Dep. Informática, Universidade do Minho, Braga, Portugal
and
Sidereus S.A., Porto, Portugal

Abstract. This paper describes work in re-structuring a corpus of standard algorithmic knowledge written in VDM notation. Algorithms are classified and catalogued in a tabular structure according to their formal specification, once this is subject to a factorization which identifies the underlying polynomial structure and inductive behaviour.
Each polynomial gives rise to a column in the table and captures a particular "efficiency" pattern. Rows in the table correspond to classes of problems, *e.g.* sort, count *etc.* "Holes" in the table correspond to (often "new") algorithms and can be filled in by inter-combining and shifting around the elementary components of the available specifications. These are identified by factorization techniques using the relational hylomorphism calculus.

Things are different from each other, and each can be reduced to very small parts of itself. (Ancient knowledge)

1 Introduction

Formal modelling is a means of capturing and reasoning about the knowledge embodied in the solution to a problem. If performed at the right level of abstraction, using a mathematically tractable notation, it is a significant step towards reliable software description able to endure arbitrary technology change. Still the question arises at abstract level: how much is "new" in a fresh model when compared to others already available? Put in other words: where is the borderline between invention and sheer routine-work in formal modelling?

Domain analysts try to answer these questions by structuring specification repositories according to problem areas. Still it is often the case that the model one is looking for can only be found in another problem area, once stripped from its domain-specific terminology. There is a need to investigate alternative internal structures for formal model repositories able to unveil their mathematical essence and spot model "intersection" in a systematic way.

This paper describes work which, in this spirit, tries to re-structure a corpus of standard algorithmic knowledge written in VDM notation. Algorithms are classified and catalogued in a tabular structure according to their formal specification, once this is subject to a transformation which identifies the underlying inductive (recursive) pattern.

⋆ Dep. Informática, Universidade do Minho, Campus de Gualtar, 4700 Braga, Portugal. EMAIL: jno@di.uminho.pt.

2 Motivation

Algorithmics is a vast body of knowledge which is hard to grasp because it is very unstructured. Teaching algorithm design today can somehow be compared to teaching chemistry before Mendeleyev published his periodic table in 1869, whereupon each element eventually found “its place” in a systematic way.

At the industrial level, the formal methods practitioner is often frustrated by the difficulty in expressing the commonalities of specs which “look alike”. As Mendeleyev and others did in the past, in their own field of research, software theorists should invest in *factorization* methods able to spot elementary, yet formally meaningful abstract algorithmic *elements* which can combined in several (possibly novel) ways to build more and more elaborate specs.

This concern can be framed in today’s trend of component-oriented programming and software reuse. But there is something else to consider: for the meaning of a component aggregation to be predicted there is a need for calculation. This requires a formal notation and calculus. As a motivation, let us consider another field of knowledge where factorization plays a central rôle — that of elementary number theory.

Factorization versus calculation

“Brute force” arithmetic calculations such as, for instance, $\frac{756}{792} = 0.9545454\ldots$ are inaccurate and error-prone. At school one is taught an alternative, accurate way, in which one “understands” the numbers first, by what is known as prime *factorization*,

$$756 = 2^2 \times 3^3 \times 7$$
$$792 = 2^3 \times 3^2 \times 11$$

and then performs the calculation (by what might be called “prime factor *fusion*):

$$\begin{aligned}\frac{756}{792} &= \frac{2^2 \times 3^3 \times 7}{2^3 \times 3^2 \times 11} \\ &= 2^2 \times 2^{-3} \times 3^3 \times 3^{-2} \times 7 \times 11^{-1} \\ &= 2^{-1} \times 3 \times 7 \times 11^{-1} \\ &= \frac{21}{22}\end{aligned}$$

A famous result of elementary number theory, the *Fundamental Theorem of Arithmetics*, underlies this method: *every integer greater than 1 can be written in the form* $p_1^{n_1} p_2^{n_2} \ldots p_k^{n_k}$ *where* $n_i > 0$ *and the* p_i*’s are distinct primes.*

Is there a similar *fundamental theorem* applicable to software code factorization? What is the “prime number” equivalent in the software field? We will address these topics in the sequel under the adoption of a formal notation for software specification — the ISO/IEC 13817-1 standard (vulg. VDM-SL) [8].

3 Algorithm and data specification

It is widely accepted, since the structured programming discipline of the 1970s, that *data precede algorithms* in software construction. For instance, the author of the following explicit specification [9] of the *insertion sort* algorithm surely knows not only how to inspect a finite list,

```
doSort: seq of int -> seq of int
doSort(l) ==
  if l = [] then []
  else let sorted = doSort (tl l) in
           insertSorted (hd l, sorted);
```

but also how to build one:

```
insertSorted: int * seq of int -> seq of int
insertSorted(i,l) ==
  cases true :
    (l = [])    -> [i],
    (i <= hd l) -> [i] ^ l,
    others      -> [hd l] ^ insertSorted(i,tl l)
  end
```

However, it is less obvious what kind of data-structure has inspired the specifier of another sorting algorithm,

```
mergeSort : seq of int -> seq of int
mergeSort (l) ==
  cases l :
    [] -> l ,
    [e] -> l ,
    others -> let l1 ^ l2 in set {l}
                  be st abs(len l1 - len l2) < 2
              in let l_l = mergeSort (l1),
                     l_r = mergeSort (l2)
                 in lmerge (l_l, l_r)
  end;
```

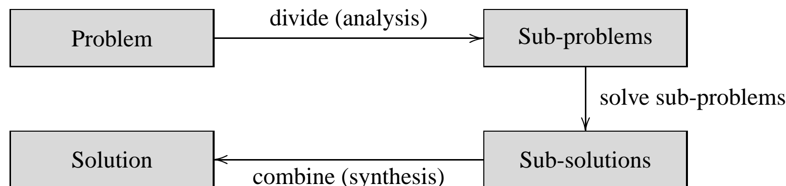
which is doubly recursive despite the fact that the data it manipulates is linearly structured. If such a data structure exists, it should be hidden in the algorithm itself.

Recent research in the mathematics of programming has provided further insight into the rôle — either *real* or *virtual* — of data structuring in program construction [5, 2]. A large database file stored on disk is surely a visible, *real* data structure. However, the binary tree which silently controls *mergeSort*'s double recursion is only evident to the software formalist who knows that real data structures may exist at specification level which disappear (*i.e.* become *virtual*) throughout the process of software refinement by calculation.

In the remainder of this paper we will not only see how to uncover such invisible data-structures but will also appreciate their rôle in algorithm factorization, while providing a prime criterion for algorithm classification. First of all, we need to identify the class of algorithms which will be covered by the approach.

3.1 The divide-and-conquer generic algorithmic pattern

Computing has to do with problem solving. The general strategy of approaching a complex problem by *a priori* splitting it into a finite number of (less complex) sub-problems which are solved independently is inherently associated to the analytical power of the human brain:



The rule is to iterate this scheme until the sub-problems are "mind-sized". Pattern-matching and case-analysis (such as in `insertSorted` and `mergeSort` above) are particular instances of this strategy.

A more expressive situation arises wherever some of the sub-problems, though "smaller", share the structure of the original problem itself. One is lead to what has become known as the *divide and conquer* generic algorithmic structure. Let A be the class of problems to solve and let $\mathsf{F}\,A$ denote the particular organization of sub-problems of A inferred by the problem analyst. The *divide* step can be expressed by arrow

$$A \xrightarrow{divide} \mathsf{F}\,A$$

Let B denote the class of problem solutions. The dual of *divide* is the *conquer* step, which consists of generating the solution once solutions to all the sub-problems have been found:

$$\begin{array}{ccc} A & \xrightarrow{divide} & \mathsf{F}\,A \\ {\scriptstyle solve}\downarrow & & \\ B & \xleftarrow[conquer]{} & \cdots \end{array}$$

By knowing that the main problem and its sub-problems share the same "type", the overall $solve$ step one is looking for can be structurally "reused" in solving the sub-problems themselves:

$$\begin{array}{ccc} A & \xrightarrow{divide} & \mathsf{F}\,A \\ {\scriptstyle solve}\downarrow & & \downarrow{\scriptstyle \mathsf{F}\,solve} \\ B & \xleftarrow[conquer]{} & \mathsf{F}\,B \end{array}$$

The notation "$\mathsf{F}\,solve$" captures the fact that $solve$ is reused in a way driven by the F-structure itself — the *inductive* step. As we will see later on, this notation is not only suggestive but also fully justifiable on theoretical grounds.

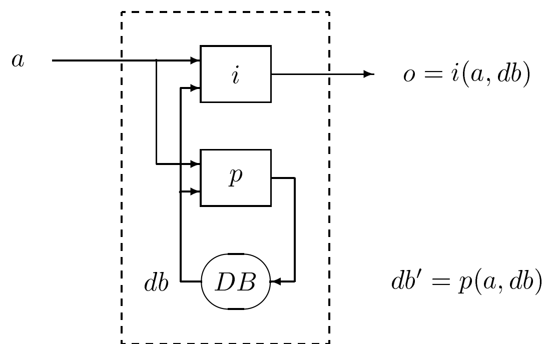
A well-known instance of *divide and conquer* arises in compiler design, where A is the particular language to be compiled, B stands for *binary code*, the divide-step is called *parse* and the conquer-step is called *code-generation*. In this case, the four arrows in the diagram can be regarded as mathematical functions. This is not the general case, however.

4 Algorithm specifications as binary relations

The following VDM-SL operation specification

```
POP() r: A
ext wr stack: Stack
pre not empty(stack)
post stack = pop(stack) and r = top(stack);
```

builds up on `pop` and `top`, two well-known functions of the ubiquitous *stack* abstract datatype. It illustrates a common procedure in specifying (deterministic) *state transactions* on top of functional libraries. In general, given some local state db, such transactions are expressed as “pairs of functions”:



One of these (i) delivers an output and the other (p) performs a state-transition. In general, i and p will be *partial* functions, as is the case of `pop` and `top` above. This requires some kind of ordered mathematical structure at the semantics level. The adoption of *binary relations*, which has a long tradition in the *pre/post* constructive specification style [13, 14], is advantageous: a binary relation is a single mathematical device able to express

- total/partial functions
- predicates, datatype invariants and loop-invariants
- orders and inductive structures
- nondeterminism
- vagueness or under-specification.

The following section sets up some basic terminology about binary relations, before one can address the issue of relational factorization. For a thorough presentation of the whole relational calculus the reader is referred to [5, 2].

5 Rudiments of the relational calculus

Let $B \xleftarrow{R} A$ denote a binary relation on datatypes A (source) and B (target). The underlying partial order on relations will be written $R \subseteq S$, meaning that S is either more defined or less deterministic than R. Equality on relations can be established by $\subseteq$-antisymmetry: $R = S \equiv R \subseteq S \wedge S \subseteq R$.

Relations can be combined by three basic operators: composition ($R \cdot S$), converse (R°) and meet ($R \cap S$). Meet corresponds to set-theoretical intersection and composition is defined in the usual way: $b(R \cdot S)c$ holds wherever there exists some $a \in A$ such that $aSc \wedge bRa$. Everywhere $T = R \cdot S$ holds, the replacement of T by $R \cdot S$ will be referred to as a "factorization" and that of $R \cdot S$ by T as "fusion". Every relation $B \xleftarrow{R} A$ admits two trivial factorizations, $R = R \cdot id_A$ and $R = id_B \cdot R$ where, for every X, id_X is the identity relation mapping every value in X onto itself. Some standard terminology arises from this relation: a (endo)relation $A \xleftarrow{R} A$ will be referred to as *reflexive* iff $id_A \subseteq R$ holds and as *coreflexive* iff $R \subseteq id_A$ holds. As a rule, subscripts are dropped wherever types are implicit or easy to infer.

The converse of R is the relation R° such that $a(R^\circ)b \equiv bRa$. Converse is of paramount importance in establishing the taxonomy of binary relations. Let us first define two derived operators, *kernel*

$$\mathit{ker}\, R \stackrel{\text{def}}{=} R^\circ \cdot R \tag{1}$$

and *image* (its dual)

$$\mathit{img}\, R \stackrel{\text{def}}{=} \mathit{ker}\,(R^\circ) \tag{2}$$

An alternative to (2) is to define $\mathit{img}\, R = R \cdot R^\circ$, since converse commutes with composition, $(R \cdot S)^\circ = S^\circ \cdot R^\circ$ and is involutive: $(R^\circ)^\circ = R$.

Kernel and image lead to the following taxonomy: a relation R is said to be

- **entire** (or total) iff its kernel is reflexive;
- **simple** (or functional) iff its image is coreflexive [1];
- a **function**, iff it is both simple and entire.

Functions will be denoted by lowercase letters (f, g, *etc.*) and are such that $f \subseteq R$ implies that R *is entire* and $R \subseteq f$ implies that R *is simple*. In general, *larger than entire means entire* and *smaller than simple means simple*.

The relational operators introduced so far enjoy a vast number of properties which are omitted at this point for the sake of brevity (see [5] for a thorough account). Instead, we will focus on their expressive power. For instance, coreflexives are fragments of the identity relation and can be used to model predicates: the "meaning" of a predicate $\mathit{bool} \xleftarrow{\phi} A$ is a coreflexive relation $[\![\phi]\!]$ such that $a[\![\phi]\!]a \equiv \phi\, a$ that is, the relation that maps every a which satisfies ϕ onto itself. In this way, predicate conjunction boils

[1] Simplicity is the dual of entireness. Simple relations are also called *partial functions*, recall *e.g.* `top` and `pop` above.

down to coreflexive composition: $[\![\phi \wedge \psi]\!] = [\![\phi]\!] \cdot [\![\psi]\!]$. In this context, the following VDM-SL implict specification of *sorting* [9]

```
ImplSort(l: seq of int) r: seq of int
post IsOrdered(r) and IsPermutation(r,l);
```

abbreviates to $ImplSort \stackrel{\mathrm{def}}{=} IsOrdered \cdot [\![IsPermutation]\!]$, where predicate `IsPermutation`

```
IsPermutation: seq of int * seq of int -> bool
IsPermutation(l1,l2) ==
  forall e in set (elems l1 union elems l2) &
    card {i | i in set inds l1 & l1(i) = e} =
    card {i | i in set inds l2 & l2(i) = e};
```

is embedded as a coreflexive. Which one? It can be checked that it coincides with $\mathsf{ker}\, seq2bag$, where $seq2bag$ ("convert a sequence into a bag") is the function which loses information about the ordering of a sequence:

```
seq2bag: seq of int -> map int to nat1
seq2bag(l) ==
  { e |-> card { i | i in set inds l & l(i) = e } |
    e in set elems l };
```

So one might have written, in the first place,

$$ImplSort \stackrel{\mathrm{def}}{=} IsOrdered \cdot (\mathsf{ker}\, seq2bag)$$

in a "pointfree" specification style which is fully based on the relation algebra and thus amenable to reasoning and calculation [2]. For instance, the fact that $IsPermutation$ defines an equivalence relation does not require an explicit logical proof anymore: it suffices to know that the kernel of a total function always is reflexive, symmetric and transitive [3].

6 A relational approach to divide-and-conquer

The divide-and-conquer scheme introduced in section 3.1 can be formalized in the above relational framework by regarding the arrows in the diagram as binary relations,

(3)
$$\begin{array}{ccc} A & \xrightarrow{S} & \mathsf{F}\,A \\ {\scriptstyle X}\downarrow & & \downarrow{\scriptstyle \mathsf{F}\,X} \\ B & \xleftarrow[R]{} & \mathsf{F}\,B \end{array}$$

[2] The move from the *pointwise* level (involving operators as well as variable symbols, logical connectives, quantifiers, *etc.*) to the *pointfree* one is compared elsewhere [19] to the *Laplace transformation*. The former is more intuitive but harder to reason about, the latter is less descriptive but more algebraic and compact. As in traditional mathematics, there is room for both in formal specification.

[3] A relation R is symmetric iff $R = R^{\circ}$ and transitive iff $R \cdot R \subseteq R$.

where S is the divide relation, R the conquer one and X the specification of interest.

The equation implicit in diagram (3) is of the form

$$X = R \cdot (\mathsf{F}\, X) \cdot S \tag{4}$$

and is known as a *hylo equation* or hylo specification [4]. Prior to discussing solutions for (4) we need to be more specific about the meaning of F in the relational framework. Symbol F is overloaded: $\mathsf{F}\, A$ means a (parametric) datatype, *e.g.* $\mathcal{P}A$ (`set of A` in VDM-SL), while $\mathsf{F}\, X$ means a relation $\mathsf{F}\, B \xleftarrow{\mathsf{F}\, X} \mathsf{F}\, A$ given some relation $B \xleftarrow{X} A$. For instance, $\mathcal{P}X$ will relate every subset s of A to the following subset of B: $\{b \in B \mid \exists a \in s.bXa\}$ [5]. Should X be a function f, $\mathcal{P}f$ can be identified with the set-comprehension $\{f\, a \mid a \in s\}$. Technically, we will say that F is a **relator** and assume a number of properties, namely that F is monotone and commutes with composition, converse and the identity:

$$\mathsf{F}\,(R \cdot S) = (\mathsf{F}\, R) \cdot (\mathsf{F}\, S) \tag{5}$$

$$\mathsf{F}\,(R^\circ) = (\mathsf{F}\, R)^\circ \tag{6}$$

$$\mathsf{F}\, id = id \tag{7}$$

Also note the following terminology: every relation of type $A \xleftarrow{R} \mathsf{F}\, A$ will be referred to as an F-*algebra* and its converse $A \xrightarrow{R^\circ} \mathsf{F}\, A$ as an F-*coalgebra*. A will be mentioned as the *carrier* of both the algebra and the coalgebra. In the divide-and-conquer scheme (3), the *divide* relation is always a coalgebra and the *conquer* relation is always an algebra.

Now we turn to the discussion of how to solve (4). In particular, we look for a unique least solution $\mu X.R \cdot (\mathsf{F}\, X) \cdot S$, if it exists. The answer to this concern is inherently related to the factorization of the least solution itself. Because F commutes with composition, one can discuss such a factorization. Suppose it makes sense to write $X = X_2 \cdot X_1$, for some intermediate datatype C which is the target of X_1 and the source of X_2. Then (3) can be expanded in the following diagram, for some suitable F-algebra $C \xleftarrow{T} \mathsf{F}\, C$:

$$\begin{array}{ccc} A & \xrightarrow{S} & \mathsf{F}\, A \\ {\scriptstyle X_1}\downarrow & & \downarrow{\scriptstyle \mathsf{F}\, X_1} \\ C & \xleftarrow[T]{} & \mathsf{F}\, C \\ {\scriptstyle X_2}\downarrow & & \downarrow{\scriptstyle \mathsf{F}\, X_2} \\ B & \xleftarrow[R]{} & \mathsf{F}\, B \end{array} \tag{8}$$

[4] Hylo (from "$\upsilon\lambda o\sigma$") means "matter". This choice of terminology purports the idea that every piece of algorithmic knowledge *(matter)* can be specified by a diagram/equation of this kind — see [5, 2] for details.

A remarkable theorem of the relational calculus establishes conditions on S, R and T for the corresponding factorization to make sense and a unique, least solution to be expressible in such a way. For economy of presentation we cannot deal with the full details of this result, which can be found in the literature [5, 3, 2], and will address it in a fairly intuitive way.

First of all, S is required to be "*well-founded*" [5]. This ensures that the "size" of a sub-problem generated by S is strictly smaller than its source, *i.e.* termination. Secondly, T is required to be *bijective* [6] over the datatype which is inductively defined by F, that is, $C = \mu\mathsf{F}$. If it exists, $\mu\mathsf{F}$ is precisely the type one "defines" by writing domain equations in VDM-SL such as, for instance, in the following specification of "leaf-trees" of integers:

```
LTree =  Leaf | Node ;
Leaf  :: value: int ;
Node  :: left: LTree right: LTree ;
```

In this case, we are defining datatype `LTree` as the least fixpoint of

$$\mathsf{F}\,X \stackrel{\mathrm{def}}{=} \mathtt{int} \mid X{*}X \tag{9}$$

(Note the use of "`*`" abbreviating the record structure defined by `Node`.) However, can we rely on (9) as defining a relator?

At this point it is preferable to abandon VDM-SL-syntax and resort to the standard "polynomial" notation for datatypes. In this notation (9) will be written as $\mathsf{F}\,X \stackrel{\mathrm{def}}{=} int + X^2$, where X^2 means the same as $X \times X$ and $\times$ is the standard binary relator associated to Cartesian product:

$$\langle b, d\rangle(R \times S)\langle a, c\rangle \equiv (bRa) \wedge (dSc) \tag{10}$$

The companion sum relator $(+)$ works over disjoint sums $A + B = \{i_1\,a \mid a \in A\} \cup \{i_2\,b \mid b \in B\}$ where i_1 are i_2 arbitrary (but consistent) disjoint injections, such as those implicit in `Leaf` and `Node` in the VDM-SL fragment above. Its definition

$$R + S = [i_1 \cdot R, i_2 \cdot S] \tag{11}$$

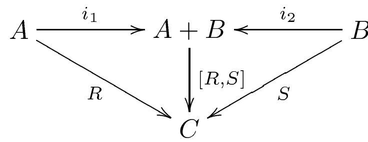
is based on a basic relational combinator which captures case analysis,

$$\begin{aligned} c[R, S](i_1\,a) &\equiv cRa \\ c[R, S](i_2\,b) &\equiv cSb \end{aligned} \tag{12}$$

[5] Note the quotes, which hide some technical subtleties. What needs to be well-founded is the composition $\in_{\mathsf{F}} \cdot S^{\circ}$, where $\in_{\mathsf{F}}$ is the so-called F-*membership* relation associated to relator R. For the (many) technical details omitted here the reader is referred to *e.g.* [5, 2].

[6] A relation is *bijective* if it is an injective and surjective function. A relation R is *injective* iff R° is simple and *surjective* iff R° is entire.

cf. diagram



Using this notation, the two `LTree` constructors `mk_Leaf` and `mk_Node` can be "packaged" together in a single algebra:

$$\texttt{LTree} \xleftarrow{[\texttt{mk_Leaf},\texttt{mk_Node}]} \texttt{int} + \texttt{LTree} \times \texttt{LTree}$$

According to the semantics of VDM-SL, this algebra is in fact a bijection (isomorphism) because `LTree` is a least fixpoint. In general, a least fixpoint solution $\mu X.\mathsf{F}\,X$ is given by exhibiting the corresponding isomorphism, often baptized "in":

$$\mu\mathsf{F} \xleftarrow{in} \mathsf{F}(\mu\mathsf{F}) \qquad (13)$$

Another remarkable fact concerns this algebra in and states that, given any other algebra $B \xleftarrow{R} \mathsf{F}\,B$, there is a unique X satisfying $X \cdot in = R \cdot (\mathsf{F}\,X)$. In order to express this uniqueness, it is common practice to denote such a single solution by $(\!|R|\!)$. This can be read (according to the local jargon) as the "inductive extension of R, "fold R" or "catamorphism of R". Since R fully determines $(\!|R|\!)$, it sometimes is referred to as the "gene" of $(\!|R|\!)$. For instance, the following VDM-SL fragment contains two functions, one adding up all leaves of a leaf-tree and the other counting them:

```
addLTree: LTree -> int
addLTree(t) ==
   cases t :
      mk_Leaf(i) -> i ,
      mk_Node(t1,t2) -> addLTree(t1) + addLTree(t2)
   end;

countLTree: LTree -> int
countLTree(t) ==
   cases t :
      mk_Leaf(-) -> 1 ,
      mk_Node(t1,t2) -> countLTree(t1) + countLTree(t2)
   end;
```

Converted to the $(\!|_|\!)$ notation, these functions abbreviate to $(\!|[id, +]|\!)$ and $(\!|[\underline{1}, +]|\!)$, respectively, where $\underline{1}$ denotes the constant function $\lambda i.1$. By focussing on the *genetic material* of specs only, this notation keeps track of what is *really important* in a specification. All the rest is repetitive syntax implicit in the particular F which underlies the $(\!|_|\!)$ construction.

Such a syntax is, in fact, what a VDM-SL practitioner will inevitably "cut and paste" from spec to spec. The outcome can be surprising: replace `id` and `+` in `addLTree` by `mk_Leaf` and `mk_Node`, respectively. The recursive function thus defined is nothing but the identity function on `LTree` and instantiates the so-called *reflection* property,

$$(\!|in|\!) = id \qquad (14)$$

a consequence of the uniqueness of $(\!| in |\!)$.

Returning to diagram (8), it should be clear that the replacement of T by in turns X_2 into $(\!| R |\!)$. Moreover, $(\!| S^\circ |\!)$ exists and its converse bears the same type as X_1:

$$
\begin{array}{ccc}
A & \xleftarrow{\;S^\circ\;} & \mathsf{F}\,A \\
{\scriptstyle (\!| S^\circ |\!)^\circ}\downarrow & & \downarrow{\scriptstyle \mathsf{F}\,(\!| S^\circ |\!)^\circ} \\
\mu\mathsf{F} & \xleftarrow{\;in\;} & \mathsf{F}\,\mu\mathsf{F} \\
{\scriptstyle (\!| R |\!)}\downarrow & & \downarrow{\scriptstyle \mathsf{F}\,(\!| R |\!)} \\
B & \xleftarrow[\;R\;]{} & \mathsf{F}\,B
\end{array}
$$

The result we are looking for — hereafter mentioned as the **hylo-factorization theorem** — establishes that, *under the above conditions*, one has

$$\mu X.(R \cdot \mathsf{F}\,X \cdot S) = (\!| R |\!) \cdot (\!| S^\circ |\!)^\circ \tag{15}$$

or, renaming S to S° and simplifying:

$$\mu X.(R \cdot \mathsf{F}\,X \cdot S^\circ) = (\!| R |\!) \cdot (\!| S |\!)^\circ \tag{16}$$

Moreover, this solution is unique [7].

This theorem provides another perspective on the divide-and-conquer scheme (3): what really matters is the particular choice of pattern (F) for sub-problem organization, which induces inductive type $\mu\mathsf{F}$. This can be left implicit or be made explicit by hylo-factorization. In this case, an explicit data-structure is built which saves the outcome of a "one go" *divide* step $(\!| S |\!)^\circ$ and passes it on to the *conquer* step $(\!| R |\!)$ for processing. Compilers are examples of programs which work in this way, because the intermediate data-structure (abstract syntax tree) is traversed several times, and so making it explicit is cost-effective. In general, however, specifiers or programmers tend to "fuse" things very early in design, thus *virtualizing* this structure — if they ever became aware of its existence.

7 Virtual data structuring

Let us apply the above result to `mergeSort`. F, R and S will be inferred by following (in a rather lightweight manner) the algorithm presented in [11], which covers case-based functional specifications such as `mergeSort`. For reasons to be explained shortly, case `[] ->l` will be removed from the specification. Two cases remain, meaning that F is a sum of two summands. Termination is ensured by case `[e] -> l`, in which the singleton list function $singl = \lambda e.[e]$ occurs. In the other case, a binary operation

[7] In the same way $(\!| R |\!)$ denotes the unique solution to equation $X = R \cdot \mathsf{F}\,X \cdot in^\circ$, notation $[\![\, R, S^\circ \,]\!]$ often abbreviates $(\!| R |\!) \cdot (\!| S |\!)^\circ$ wherever this is the unique solution of $X = R \cdot \mathsf{F}\,X \cdot S^\circ$. Thus, $(\!| R |\!) = [\![\, R, in^\circ \,]\!]$.

(`lmerge`) is applied to the outcome of two recursive calls. Altogether, (4) instantiates to

$$
\begin{array}{ll}
& mergeSort = [singl, lmerge \cdot (mergeSort \times mergeSort)] \cdot S \\
\equiv & \quad \{ \text{ absorption law } [R, S] \cdot (T + U) = [R \cdot T, S \cdot U] \textit{ cf.} \text{ [5] } \} \\
& mergeSort = [singl, lmerge] \cdot (id + mergeSort \times mergeSort) \cdot S
\end{array}
$$

So $R = [singl, lmerge]$, $\mathsf{F}f = id + f \times f$ and $\mathsf{F}X = \mathtt{int} + X \times X$. This unveils $\mathtt{LTree} = \mu X.\mathtt{int} + X \times X$ as the virtual datatype underlying the operation of `merge-Sort`. Finally, coalgebra S can be inferred by extracting "what remains":

```
S: seq of int -> ( int | seq of int * seq of int )
S(l) ==
   cases l :
      [e] -> e ,
      others -> let l1 ^ l2 in set {l} be st abs (len l1 - len l2) < 2 in
                mk_(l1,l2)
   end;
```

Rather more elegant is the following alternative definition of S,

(17) $$S = [singl, pconc]^{\circ}$$

which uses converse and "partial concatenation" [8]:

```
pconc :  seq of int * seq of int -> seq of int
pconc(l1,l2) ==  l1 ^ l2
pre abs (len l1 - len l2) < 2 ;
```

S either extracts the sole element of a singleton input list, or splits the input list into two sublists of equal (or almost equal) length. This means that $(\!|S^{\circ}|\!)^{\circ}$ builds an (almost) balanced tree whose leaves are the items to sort, and this is the tree which $(\!|R|\!)$ visits to produce the output list. This will be sorted thanks to the efforts of `lmerge`:

```
lmerge: seq of int * seq of int -> seq of int
lmerge(l1,l2) ==
  cases mk_(l1,l2):
    mk_([],l),mk_(l,[]) -> l,
    others              -> if hd l1 <= hd l2 then [hd l1] ^ lmerge(tl l1, l2)
                              else [hd l2] ^ lmerge(l1, tl l2)
  end ;
```

In summary, the factorization of `mergeSort` helps in understanding the "Equal-size, Easy Split, Hard Join" classification of the algorithm [10].

[8] The use of converse as a specification device is thoroughly dealt with in reference [17].

The inspection of the intermediate type of a recursive definition can also be of help in debugging. Suppose one is given the following (simpler) version of the algorithm:

```
mergesort : seq of int -> seq of int
mergesort (l) ==
  cases l:
    [] -> [],
    l1^l2 -> lmerge (mergesort(l1), mergesort(l2))
  end;
```

Following the same procedure as above, one infers $\mathsf{F}\,X = 1 + X^2$ as the underlying pattern of recursion, where 1 is the one-element datatype inhabited by `nil`. However, there is something wrong: $\mu X.1 + X^2$ is a "shape tree" where there is no room to store the integers to sort! In fact, the algorithm only terminates for `l = []`, as the underlying coalgebra $S = [\underline{[\,]}, \hat{}\,]^{\circ}$ is not "well-founded".

This leads us back to the reason why we didn't consider case `[] ->l` above: it "does not belong" to the algorithm because coalgebra (17) stops at the singletons and never reaches `[]`. So case `[] ->l` should be treated outside the algorithm, as in [1] (but not in [10]). In summary, `mergeSort` should be defined over type constructor `seq1 of` rather than `seq of`.

Since the pioneering work of Darlington [7], sorting has remained fertile ground for algorithm classification [9] in which factorization plays its rôle. For instance, the main distinction between *mergeSort* and *quickSort*

```
quickSort : seq of int -> seq of int
quickSort (l) ==
  cases l:
    [] -> [],
    -^[x]^- -> quickSort ([y | y in set elems l & y < x]) ^ [x] ^
               quickSort ([y | y in set elems l & y > x])
  end
```

resides in the virtual structure itself, which is a binary search tree ($\mu X.1 + \mathtt{int} \times X^2$) in

$$quickSort = (\![[\underline{[\,]}, inord]]\!) \cdot (\![[\underline{[\,]}, pinord]]\!)^{\circ}$$

where

```
inord: int * (seq of int * seq of int) -> seq of int
inord(x,mk_(l,r)) == l ^ [x] ^ r;

pinord: int * (seq of int * seq of int) -> seq of int
pinord(x,mk_(l,r)) == inord(x,mk_(l,r))
pre forall y in set elems l & y < x and forall y in set elems r & y > x;
```

In summary, *quickSort* is nothing but binary-tree *in-order* traversal *(conquer)* following the converse of a *partial* in-order traversal *(divide).* The latter does the hard job of ensuring that the intermediate tree is bi-ordered. Thus *quickSort*'s classification in [10]: "Equal-size, Hard Split, Easy Join".

[9] See *e.g.* [10, 5, 1] and the references there. Reference [5] includes the pointfree calculation of some sorting algorithms.

8 Towards a "periodic" table of algorithms

Table 1 shows a sample of a repository of VDM-SL specifications which have been factored and classified for teaching purposes. Columns represent recursion patterns and rows the types of the specifications being defined [10]. One can see that, for instance, $fibonnaci$ and $doubleFactorial$ are in the same column as $mergeSort$, all of them sharing `LTree` as the virtual intermediate datatype. Because of the reflection law (14), virtual datatypes may also occur as *real* (input/output) ones, as *e.g.* `BTree` in $pre/in/postOrder$.

The recursion pattern of the rightmost column is not dealt with in this paper. `HTree` ("hierarchical tree") is the intermediate datatype of two problems, *explode* (bill of materials) and *tar* (file system archiving utility) which are shown in [18] to be "abstractly identical".

Table 1 is not yet the one containing the most elementary "specification matter". Factorization reveals that it is meaningless to go smaller than the algebra and coalgebra implicit in the "hylo-formula" $(\!|R|\!) \cdot (\!|S^\circ|\!)^\circ$ into which every entry in Table 1 can be decomposed. So we collect such specification *elements* into another table — Table 2 below — under the convention that every S° in the table means that S is a coalgebra (or an algebra otherwise).

Table 2 is just a prototype of what a comprehensive tabulation of specification elements should eventually be [11]. A trained software formalist will not have difficulties in filling in the missing elements, eventually entailing the addition of new columns (such as *e.g. rose trees* [1]). However small, this table already contains what seems to be essential in solving a problem by computer: a formal classification of *divide* and *conquer* recipes.

9 Discussion

The column in Table 1 corresponding to the intermediate type of finite lists ($\mu X.1+A\times X$) includes many standard VDM-SL primitive operators and is more crowded than others. The reason for this is two-fold. First of all, many mathematical operators (*e.g.* the factorial function) are defined by *primitive recursion* over $\mu X.1+X$, the natural numbers. However, primitive recursion falls in a special class of recursion schemata, known as *paramorphisms* [12]. A standard result [16] establishes that every F-paramorphism can be converted into a G-hylomorphism, where $\mathsf{G}X = \mathsf{F}(\mu\mathsf{F} \times X)$. For $\mathsf{F}X = 1 + X$ (natural numbers) one has $\mathsf{G}X = 1 + (\mathtt{nat} \times X)$, *i.e.* finite lists of natural numbers. This explains why *factorial* ($n!$) and *square* (n^2) have been archived in this column, the

[10] No special effort has been put on parameterization and genericity because VDM-SL does not fully support parametric or generic types. There is much room for improvement in this respect, for instance in expressing sorting parametrically on the underlying ordering, in letting an arbitrary monoid to replace all occurrences of associative binary operators with an unit element, and so on.

[11] The remarks on lack of genericity made in footnote 10 apply also to this table.

[12] Paramorphisms generalize primitive recursion [15] to arbitrary F. For instance, the semantics of the popular `wc` ("word count") program is a list-paramorphism [19].

Table 1. Sample of a VDM-SL specification repository

$\mathsf{F}\,X$	$1+X$	$1+A\times X$	$A+X^2$	$1+A\times X^2$	$(B\times A+B\times X)^\star$
$\mu\mathsf{F}$	nat	$\mathtt{seq\,of}\ A$	$LTree$	$BTree$	$HTree$
$In \rightarrow Out$	**Specifications**				
$nat \rightarrow bool$	odd $even$				
$nat \rightarrow nat$		$square$ $factorial$	$fibonnaci$ $doubleFactorial$		
$nat \rightarrow \mathtt{set\,of}\ nat1$		$inseg$			
$\mathtt{seq\,of}\ A \rightarrow \mathtt{seq\,of}\ A$		$insertSort$ $invSeq$	$mergeSort$	$quickSort$	
$\mathtt{seq\,of}\ A \rightarrow bool$		$ordSeq$			
$\mathtt{seq\,of}\ A \rightarrow \mathtt{set\,of}\ A$		$elems$			
$\mathtt{seq\,of}\ A \rightarrow \mathtt{set\,of}\ nat1$		$inds$			
$\mathtt{seq\,of}\ A \rightarrow \mathtt{Bag\,of}\ A$		$seq2bag$			
$\mathtt{seq\,of}\ A \rightarrow nat$		len			
$LTree \rightarrow bool$			$balLTree$		
$LTree \rightarrow nat$			$depthLTree$		
$LTree \rightarrow int$			$addLTree$ $countLTree$		
$LTree \rightarrow \mathtt{seq\,of}\ A$			$tips$		
$LTree \rightarrow LTree$			$invLTree$		
$BTree \rightarrow bool$				$ordBTree$ $balBTree$	
$BTree \rightarrow nat$				$depthBTree$	
$BTree \rightarrow \mathtt{seq\,of}\ A$				$preOrder$ $inOrder$ $postOrder$	
$BTree \rightarrow \mathtt{seq\,of\,seq\,of}\ A$				$traces$	
$BTree \rightarrow BTree$				$invBTree$	
$\mathtt{set\,of}\ A \rightarrow nat$		$card$			
$\mathtt{set\,of}\ A \rightarrow \mathtt{seq\,of}\ A$		$Set2seq$			
$\mathtt{set\,of}\ bool \rightarrow bool$		$\forall$ $\exists$			
$\mathtt{set\,of\,set\,of}\ A \rightarrow \mathtt{set\,of}\ A$		$dunion$			
$\mathtt{map}\ A\ \mathtt{to}\ B \rightarrow \mathtt{set\,of}\ A$		dom			
$\mathtt{map}\ A\ \mathtt{to}\ B \rightarrow \mathtt{set\,of}\ B$		ran			
$\mathtt{set\,of}\ (\mathtt{map}\ A\ \mathtt{to}\ B) \rightarrow \mathtt{map}\ A\ \mathtt{to}\ B$		$merge$			
$PTree \rightarrow \mathtt{Bag\,of}\ A$					$explode$
$FS \rightarrow \mathtt{map}\ String\ \mathtt{to}\ A$					tar
$(Other)$				$hanoi$	

Notes: $insertSort$ corresponds to `doSort` in the main text. $\mathtt{Bag\,of}\ A$ abbreviates $\mathtt{map}A\mathtt{to\ nat1}$. For the $PTree$ (production tree) and FS (file system) datatypes see [18]. $hanoi$ refers to the well-known Towers of Hanoi problem.

Table 2. Sample of *Periodic* Table of VDM specification elements

$\mathsf{F}\,X$	$1+X$	$1+A\times X$	$A+X^2$	$1+A\times X^2$	$(B\times A+B\times X)^\star$
$\mu\mathsf{F}$	nat	$\texttt{seq of}\ A$	$LTree$	$BTree$	$HTree$
Carrier	**F-(co)algebras**				
$bool$	$[\underline{\mathrm{F}}, \neg]$ $[\underline{\mathrm{T}}, \neg]$	$[\underline{\mathrm{F}}, \vee]$ $[\underline{\mathrm{T}}, \wedge]$	$[\underline{\mathrm{F}}, \vee]$ $[\underline{\mathrm{T}}, \wedge]$		
nat	$[\underline{0}, suc]$	$[\underline{0}, +]$ $odds^\circ$ $[\underline{1}, *]$ $nats^\circ$	$[id, +]$ $[id, *]$ $[\underline{1}, +]$ $fibd^\circ$	$[\underline{1}, bmul]$ $[\underline{0}, badd]$	
$nat * nat$			$dfacd^\circ$		
$\texttt{seq of}\ A$		$[\underline{[\,]}, cons]$ $[\underline{[\,]}, rcons]$ $[\underline{[\,]}, \hat{}\,]$	$[singl, \hat{}\,]$ $[singl, pconc]$ $[singl, lmerge]$	$[\underline{[\,]}, inord]$ $[\underline{[\,]}, pinord]$ $[\underline{[\,]}, prord]$ $[\underline{[\,]}, psord]$	
$LTree$			in $in \cdot (id + sw)$		
$BTree$				in $in \cdot (id + id \times sw)$	
$HTree$					in
$\texttt{set of}\ A$		ins $pins$ $[\underline{\emptyset}, \cup]$ $ins \cdot (id + \pi_1 \times id)$ $ins \cdot (id + \pi_2 \times id)$	$[\lambda x.\{x\}, \cup]$	$[\underline{\emptyset}, bputs]$	
$\texttt{map}\ A\ \texttt{to}\ B$		ins $pins$ $[\underline{\{\mapsto\}}, munion]$			
$PTree$					$exsplit^\circ$
$\texttt{Bag of}\ A$					$exjoin$
$FileS$					$tsplit^\circ$
$\texttt{map}\ String\ \texttt{to}\ A$					$tjoin$
$(Other)$				$hsplit^\circ$	

Notes: Some of the (co)algebras above are explained in the main text. For every inductive type, in denotes the relevant isomorphism (13). suc is the successor function. $sw\langle a, b\rangle = \langle b, a\rangle$ is the *swap* function and π_1, π_2 are the two Cartesian product projections. $prord$ and $psord$ are the obvious variants of $inord$. ins and $pins$ abbreviate $[\underline{\emptyset}, puts]$ and $[\underline{\emptyset}, sput]$, respectively. $badd$, $bmul$ and $bputs$ abbreviate $(+) \cdot (id \times (+))$, $(*) \cdot (id \times (*))$ and $puts \cdot (id \times \cup)$, respectively. $hsplit$ is the coalgebra of $hanoi$. Details about the entries in the $HTree$ column can be found in [18]. Other VDM-SL functions follow:

```
cons[@A] :  @A * seq of @A -> seq of @A
cons(e,l) == [e] ^ l;

rcons[@A] :  @A * seq of @A -> seq of @A
rcons(e,l) == l ^ [e];

fibd: nat -> [ nat * nat ]
fibd(n) == if n < 2 then nil else mk_(n-1,n-2);

nats : nat -> [ nat*nat ]
nats(n) == cases n: 0 -> nil, n -> mk_(n,n-1) end;

odds : nat -> [ nat*nat ]
odds(n) == cases n: 0 -> nil, n -> mk_(2*n-1,n-1) end;

dfacd: nat * nat -> (nat | (nat * nat) * (nat * nat))
dfacd(n,m) == let k = (n+m) div 2 in if n=m then n else mk_(mk_(n,k),mk_(k+1,m));
```

former as a hylo multiplying (the list of) the n-first natural numbers and the latter as the one which adds the n-first odd-numbers.

A second reason has to do with the fact that finite lists mediate almost all standard definitions in VDM-SL involving finite sets, finite mappings and (of course) finite lists. The following instance of the usual inductive pattern underlying set-based specification illustrates this:

```
card[@A] : set of @A -> nat
card(s) ==
  cases s:
    {} -> 0,
    { e } union s' -> 1 + card[@A](s')
  end;
```

Hylo-factorization yields $card = (\!|[\underline{0}, + \cdot (\underline{1} \times id)]|\!) \cdot (\!|[\underline{\emptyset}, sput]|\!)^{\circ}$, for $\mathsf{F}\,X = 1 + A \times X$ and

```
sput[@A] :  @A * set of @A -> set of @A
sput(e,s) == {e} union s
pre not e in set s ;
```

Finite mapping recursion is performed by set-induction over the argument's domain and so also falls in this column.

However, it is apparent that the same functions could have been defined in other ways, *i.e.* under different factorizations. For instance, there would be no problem in redefining `card` by double recursion over `BTree` because — one might say — addition is commutative and associative. The same can be said about the specification which follows

```
Set2seq[@A] : set of @A -> seq of @A
Set2seq(s) ==
  cases s:
    {} -> [],
    { e } union s' -> cons[@A](e,Set2seq[@A](s'))
  end;
```

which is the converse of $elems$ — in fact a relation and not a function. The question arises: when does a hylo-equation define a function?

Reference [17] presents conditions for the least solution to a hylo-equation to be *simple*: *a* simple *relation T can be expressed as a hylo $(\!|f|\!) \cdot (\!|S|\!)^{\circ}$ provided that not only S° meets the standard requirements on termination, but also that $T \cdot S \subseteq f \cdot \mathsf{F}T$ and $\mathit{dom}\, T = \mathit{rng}\, S$ hold* [13]. Moreover, the unique solution to a hylo-equation is *entire* wherever both its algebra and coalgebra are entire [5].

It can be checked that most standard linear (*i.e.* "list-based") specifications of set or mapping functions satisfy these requirements, but there is room for other F to support the same definition, in particular in presence of commutative and associative operators. This enables us to start and fill the "holes" in Table 1 with alternative specifications,

[13] $\mathit{dom}\, T$ and $ranS$ abbreviate $\mathit{ker}\, T \cap id$ and $\mathit{img}\, S \cap id$, respectively.

for instance based on `BTree` or `LTree`. In terms of algorithmic knowledge this is good news: we are "discovering new algorithms" [14]. On formal specification grounds, however, this raises an old question — *what is* a (functional) specification? — and associated discussion on *model/property-orientation* in formal methods [14]. Isn't *polytypic i.e.* F-independent [12] specification a possible compromise? A proper answer to this question falls outside the scope of the current paper.

10 Related work

The approach presented in this paper underlies the way algorithmics are taught at Minho, based on HASKELL in the first years and on VDM-SL in the final ones. A generic visual programming tool has been developed [6] in GENERIC HASKELL which performs hylo-factorization according to the algorithm of [11]. The prospect of building a web/hypertext interface for the whole repository, allowing for navigation, composition and animation of the available specification elements is being considered.

Algorithm classification is related to polytypical programming [12]. By grouping rows in Table 1 which perform the same "abstract task" (*e.g.* summing, counting, collecting, sorting, etc) and generalizing the relevant "genetic material" one would be able to identify the generic functions of libraries such as, for instance, POLYP.

Hylo-decomposition and calculation is a subject of active research in the discipline of *constructive algorithmics* [16, 11, 5, 18, 2, 17]. The main motivation is to develop program optimization techniques by "fusion" and "deforestation". By contrast, [19] and the current paper point towards the opposite direction of "reforestation", in the context of program analysis, reverse engineering and program understanding. One of the concerns of the PURE project ("Program Understanding by Reverse Engineering") is to scale up this approach from functions and relations to processes and software components [4], bearing in mind its industrial application to legacy software.

Acknowledgements

The author wishes to thank Alcino Cunha and Shin-Cheng Mu for stimulating discussions on the topics addressed in this paper. Special thanks go to Luís Barbosa for his comments on an earlier draft of this paper. The use of VDMTools ® under a Free Academic Site License provided by IFAD is gratefully acknowledged.

The research described in this paper was carried out at the CCTC R&D Center (project PURE).

References

1. Lex Augusteijn. Sorting morphisms. In S.D. Swierstra, P.R. Henriques, and J.N. Oliveira, editors, *Advanced Functional Programming, Third International School, Braga, Portugal,*

[14] Back to the analogy which the title of this paper purports, recall that chemists only began to appreciate Mendeleyev's table when the discovery of elements predicted by the table took place.

September 12-19, 1998, Revised Lectures, volume 1608 of *Lecture Notes in Computer Science*, pages 1–27. Springer, 1999.

2. R. C. Backhouse. Fixed point calculus, 2000. Summer School and Workshop on Algebraic and Coalgebraic Methods in the Mathematics of Program Construction, Lincoln College, Oxford, UK 10th to 14th April 2000.
3. R.C. Backhouse and P.F. Hoogendijk. Final dialgebras: From categories to allegories. *Informatique Theorique et Applications*, 33(4/5):401–426, 1999. Presented at Workshop on Fixed Points in Computer Science, Brno, August 1998.
4. L. S. Barbosa. *Components as Coalgebras*. University of Minho, December 2001. Ph. D. thesis.
5. R. Bird and O. de Moor. *Algebra of Programming.* Series in Computer Science. Prentice-Hall International, 1997. C. A. R. Hoare, series editor.
6. A. Cunha, J. Barros, and J. Saraiva. Deriving animations from recursive definitions, 2002. To be presented at the 14th International Workshop on the Implementation of Functional Languages (IFL'02), Sept. 2002, Madrid.
7. J. Darlington. A synthesis of several sorting algorithms. *Acta Informatica*, 11:1–30, 1978.
8. J. Fitzgerald and P.G. Larsen. *Modelling Systems: Practical Tools and Techniques for Software Development* . Cambridge University Press, 1st edition, 1998.
9. The VDM Tool Group. VDM-SL sorting algorithms. Technical report, IFAD, Forskerparken 10, DK-5230 Odense M, Denmark, February 2000.
10. B.T. Howard. Another iteration on Darlington's 'A Synthesis of Several Sorting Algorithms'. Technical Report KSU CIS 94-8, Department of Computing and Information Sciences, Kansas State University, 1994.
11. Zhenjiang Hu, Hideya Iwasaki, and Masato Takeichi. Deriving structural hylomorphisms from recursive definitions. In *Proceedings 1st ACM SIGPLAN Int. Conf. on Functional Programming, ICFP'96, Philadelphia, PA, USA, 24–26 May 1996*, volume 31(6), pages 73–82. ACM Press, New York, 1996.
12. J. Jeuring and P. Jansson. Polytypic programming. In *Advanced Functional Programming*, number 1129 in Lecture Notes in Computer Science. Springer, 1996.
13. C. B. Jones. *Development Methods for Computer Programs including a Notion of Interference*. PhD thesis, Oxford University, June 1981. Printed as: Programming Research Group, Technical Monograph 25.
14. C. B. Jones. Specification and design of (parallel) programs. In *Proceedings of IFIP'83*, pages 321–332. North-Holland, 1983.
15. L. Meertens. Paramorphisms. *Formal Aspects of Computing*, 4:413–424, 1992.
16. 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.
17. Shin-Cheng Mu and Richard Bird. Inverting functions as folds. In *MPC'02: Mathematics of Program Construction*, Lecture Notes in Computer Science. Springer, 2002. (forthcoming).
18. J. N. Oliveira. *'Explosive' Programming Controlled by Calculation* . Technical Report UMDITR02/98, DI, University of Minho, September 1998. Presented at *AFP'98* (3rd Intern. Summer School on Advanced Functional Programming), Braga, Portugal.
19. J. N. Oliveira. "Bagatelle in C arranged for VDM SoLo". *Journal of Universal Computer Science*, 7(8):754–781, 2001. Special Issue on *Formal Aspects of Software Engineering* (*Colloquium in Honor of Peter Lucas* , Institute for Software Technology, Graz University of Technology, May 18-19, 2001).