Formal Aspects of Computing (1990) 2: 1-23

Formal Aspects of Computing

A Reification Calculus for Model-Oriented Software Specification

J. N. Oliveira

Grupo de C. Computação, Universidade do Minho/INESC, Rua D. Pedro V, 88-3, 4700 Braga, Portugal

Keywords: Software engineering; Formal methods; Algebraic specification; Transformational design

Abstract. This paper presents a transformational approach to the derivation of implementations from model-oriented specifications of abstract data types.

The purpose of this research is to reduce the number of formal proofs required in model refinement, which hinder software development. It is shown to be applicable to the transformation of models written in META-IV (the specification language of VDM) towards their refinement into, for example, Pascal or relational DBMSs. The approach includes the automatic synthesis of retrieve functions between models, and data-type invariants.

The underlying algebraic semantics is the so-called *final* semantics "à la Wand": a specification "is" a *model* (heterogeneous algebra) which is the final object (up to isomorphism) in the category of all its implementations.

The transformational calculus approached in this paper follows from exploring the properties of finite, recursively defined sets.

This work extends the well-known strategy of program transformation to model transformation, adding to previous work on a transformational style for operation-decomposition in META-IV. The model-calculus is also useful for improving model-oriented specifications.

1. Introduction

It is widely accepted nowadays that the industrial production of reliable software, at low cost, should be based on technologies which, at least, discuss such a

Correspondence and offprint requests to: J. N. Oliveira, Grupo de C. Computação, Universidade do Minho/INESC, Rua D. Pedro V, 88-3, 4700 Braga, Portugal.

reliability formally, i.e. based on mathematically written *specifications.* Such technologies involve the additional notion of *refinement* (or *reification* [Jon86]), i.e. any systematic process of building *implementations* from formal specifications.

Much research in this area has concentrated on devising languages and tools for formal specification. Well-known techniques for algebraic specification [GoT78, GuH78, BaW82] make it possible to define algebraic structures from which programs are developed, in the form of hierarchies of *abstract data types.* These correspond to algebras whose functionality (syntax) is fixed by a heterogeneous signature (Σ), and whose theory (semantics) is a quotient $W_\Sigma/\equiv$, where W_Σ denotes the Σ-*word algebra* (i.e. the "language" generated by Σ). There are two standard ways of finitely presenting such a quotient.

In *property*-oriented specification, $\equiv$ is the smallest Σ-congruence induced by a finite collection of Σ-equations [GoT78]. In *model*-oriented specification [Jon86], semantics are given by describing a model, i.e. a Σ-algebra $\mathscr{A}$, and $\equiv$ is then the kernel congruence relation induced by the unique homomorphism from W_Σ to $\mathscr{A}$ [BaW82].

This paper focuses on model-refinement (reification) technology, i.e. on specification refinement in the model-oriented specification style.

An approach aiming at developing a *refication calculus* for software engineering is presented. When compared with the historical development of the scientific bases for other engineering areas (e.g. mechanical and civil engineering etc.), the introduction of algebraic reification-calculi in *software engineering* appears to be a natural evolution, which may be (roughly) sketched as follows:

Until the 1960s: intuition and craft.
1970s: *Ad hoc* (informal) methods.
1980s: formal methods.
1990s: formal calculi.

Formal calculi are intended to scale up the scope of formal methods.

The reification-calculus put forward in this paper is specification-dialect independent. However, acquaintance with the VDM method and the META-IV notation [Jon80, Jon86] will help in understanding the examples. The approach was first presented in [Oli87] and further developed in [Oli88a]. Both these references resort to basic *category theory* [Mac71] following [MaA86] and [Wan79], which should be read as contextual research. To improve readability in this paper, the category-theoretical notions are replaced by set-theoretical ones.

1.1. Overview of Open Problems

Formal specifications should be as *abstract* as possible, in the sense that they should record the *essence* of problems and ignore irrelevant details. By contrast, implementations are usually full of machine-dependences which explore a concrete machine-architecture for run-time efficiency. Refinement fills in the *abstraction gap* between specifications and implementations, by providing correctness arguments proving that the latter satisfy the former. In this sense, refinement is the "kernel" phase of software development using formal methods.

The standard techniques for data refinement assume that the software engineer has sufficient intuition to "guess" (efficient) low-level model-implementations. This is unlikely, in general. Moreover, two kinds of phenomenon occur wherever

model-refinement is in progress: either one is led to more *redundant* data-representations, or one has to filter *invalid* data-representations (or both).

In real-life software design, it is sometimes cumbersome to record formally the relationship between data-models, and prove facts (adequacy, invariant-preservation etc. [Jon80]) about them. Unfortunately, it may take a considerable effort to prove facts which are intuitively obvious.

For example, consider the following toy-example, a META-IV syntax for a very simple bank accounting system:

$$BAMS = AccNr \xrightarrow{m} Status$$

$$Status :: H : AccHolder - set$$
$$A : Amount$$

$$Amount = Nat0$$

where the following data-type invariant should hold,

$$inv - BAMS(\sigma) \stackrel{def}{=} \forall n \in \mathrm{dom}\ \sigma : H(\sigma(n)) \neq \emptyset$$

enforcing that every account has, at least, one account-holder.

A VDM practitioner may take a while to formally discuss the correctness of the following (obvious!) *relational-model* implementation, where BAMS is modelled in terms of two binary relations (vulg. "tables"):

$$BAMS1 :: HT : Row1 - set \quad /^*\text{table of account-holders}^*/$$
$$AT : Row2 - set \quad /^*\text{table of amounts}^*/$$

$$Row1 :: K : AccNr\ H : AccHolder$$

$$Row2 :: K : AccNr\ A : Amount$$

subject to the following data-type invariant,

$$inv - BAMS1(mk - BAMS1(ht, at)) \stackrel{def}{=} dom(at) = dom(ht) \wedge$$
$$depKA(at) \qquad (1)$$

where

$$dom : (A\ B) - set \to A - set$$
$$dom(\rho) \stackrel{def}{=} \{a \in A \mid \exists b \in B : \langle a, b \rangle \in \rho\} \qquad (2)$$

is a generic (*domain*) relational-operator, and predicate $depKA : (Row2 - set) \to Bool$:

$$depKA(\rho) \stackrel{def}{=} \forall r, s \in \rho : (K(r) = K(s) \Rightarrow A(r) = A(s))$$

expresses a $K \to A$ *functional dependence.*

When toy-examples are scaled-up to real examples, formal proofs are either discarded (and the method no longer acceptable as formal), or they become a

serious bottleneck in development. Moreover, no definite answers have been given to questions such as:

How can we define an invariant as being "correct", "too strong", or "sufficient"?
What is the "least" abstraction (*retrieve*) function [Jon80, Jon86] between two models?
How can we keep data *redundancy* and *validity* easily under control?
Can we equationally *derive* low-level data-models from high-level data-models?

1.2. Main Objectives

Former work [Oli85] on alternative techniques in the area referred to above, is strengthened in this paper by presenting a basis for *transformational calculi* for the derivation of implementations of abstract data types. This adds to the well-known strategy of *program transformation* [BuD77, Dar82, BaW82] insofar as *whole* data-models are synthesised by transformations.

In [Oli85] only the functional-part of specification-models is subject to transformations. It follows the strategy of developing operations on the concrete level from those on the abstract level by means of the abstraction function, cf. [BuD77, Dar82, HJS87]. A target of this paper is to show how retrieve-maps can themselves be obtained by transformations enabled by a simple calculus of data-models based on set-theory.

The basic idea is that *data-redundancy* is an ordering on data-models compatible with data-model building operators. This ordering is, in turn, relaxed to a *super-redundancy* ordering whereby data *validity* is taken into account. A model can be refined up to any of its super-redundant relatives. Since these orderings are preserved by all data-constructors, refinement may proceed in a structural, stepwise manner, according to an algebra of model-transformations.

The remainder of the paper is structured as follows: Section 2 presents the underlying formalisms and overall strategy, illustrated by a simple example. The basic laws and theorems of the calculus are presented in Section 3. Section 4 gives examples of calculated reification, illustrating the inference of retrieve-maps and data-type invariants. Finally, Sections 5 and 6 draw conclusions and address technical issues for future work.

2. Formal Basis

2.1. Notation Background

The algebraic semantics underlying the formalisms below is the so-called *final semantics* [Wan79]:[1] a specification is given by a *model*, i.e. a many-sorted Σ-algebra $\mathcal{A}$ which is the *final* object (up to isomorphism) in the class of all its implementations (="more redundant" models). This approach to abstract data type semantics is detailed below by presenting some standard definitions from the literature, cf. e.g. [GoT78, Wan79, BaW82].

[1] Or *terminal semantics*, opposed to the standard initial interpretation, cf. [GOT78] for instance.

Given a set Ω of function symbols, and a set S of *sorts* ("types"), a *signature* Σ is a syntactical assignment $\Sigma: \Omega \to (S^* \times S)$ of a functionality to each function symbol; as usual, we will write $\sigma: s_1 \ldots s_n \to s$ or $s_1 \ldots s_n \to^{\sigma} s$ as shorthands of $\Sigma(\sigma) = \langle \langle s_1, \ldots, s_n \rangle, s \rangle$. Let *Sets* denote the class of all finite sets whose morphisms are set-theoretical functions. Let these be denoted by $f: X \to Y$ or $X \to^{f} Y$, where X and Y are sets.

A Σ-*algebra* $\mathscr{A}$ is a semantic assignment described by a *functor*

$$\mathscr{A}: \Sigma \to \mathit{Sets}$$

that is, $\mathscr{A} = \langle \mathscr{A}_\Omega, \mathscr{A}_S \rangle$ where $\mathscr{A}_S$ maps sorts to corresponding carrier-sets, $\mathscr{A}_\Omega$ maps operator-symbols to set-theoretical functions, and

$$\mathscr{A}_\Omega(\sigma): \mathscr{A}_S(s_1) \times \ldots \times \mathscr{A}_S(s_n) \to \mathscr{A}_S(s) \tag{3}$$

holds. Subscripts Ω and S may be omitted wherever they are clear from the context, e.g. by writing

$$\mathscr{A}(\sigma): \mathscr{A}(s_1) \times \ldots \times \mathscr{A}(s_n) \to \mathscr{A}(s)$$

instead of formula (3).

A particular Σ-algebra is the one whose carrier-set for each sort $s \in S$ contains all the "words" (*terms*, or *morphisms*) that describe objects of that sort:

$$W_\Sigma(s) \stackrel{\text{def}}{=} C(s) \cup \{\sigma(t_1, \ldots, t_n) \mid \sigma: s_1 \ldots s_n \to s \wedge \forall 1 \leq i \leq n: t_i \in W_\Sigma(s_i)\}$$

where $C(s) = {}^{\text{def}} \{\sigma \in \Omega \mid \Sigma(\sigma) = \langle \langle\ \rangle, s \rangle\}$ is the set of all "constants" of type s.

Given two algebras $\mathscr{A}, \mathscr{B}: \Sigma \to \mathit{Sets}$, $\mathscr{B}$ is said to be an *implementation* of $\mathscr{A}$ iff there is one and only one epimorphism (abstraction map) from $\mathscr{B}$ to $\mathscr{A}$. In category-theoretical terminology, $\mathscr{A}$ is said to be the *final* algebra in the category $K_{\mathscr{A}}$ of all its implementations [Wan79]. In set-theoretical terminology, one has $\mathscr{A} \sqsubseteq \mathscr{B}$ in the complete lattice of all Σ-algebras [BaW82].

Finally, a semantic congruence $\equiv$ is induced by $\mathscr{A}$ into W_Σ such that, for all terms $t, t' \in W_\Sigma$, $t \equiv t'$ *iff* $\mathscr{A}(t) = \mathscr{A}(t')$. This approach to presenting such a congruence covers, implicitly, model-oriented (or constructive) specification such as in VDM [Jon80, Jon86], Z [Hay87, Spi89] or "me too" [Hen84].

2.2. Overall Strategy

The standard way of refining a model $\mathscr{A}: \Sigma \to \mathit{Sets}$ would lead us to:

Conjecture an implementation-model $\mathscr{B}: \Sigma \to \mathit{Sets}$;

Relate $\mathscr{B}$ to $\mathscr{A}$ via a *retrieve* function;

Finally, to use such a function in arguing that $\mathscr{B}$ is a *valid* realisation of $\mathscr{A}$.

The strategy put forward in this paper is different: one resorts to *Sets* actually to derive $\mathscr{B}$ from $\mathscr{A}$. That is to say, "$\mathscr{A}$ is *transformed* into $\mathscr{B}$", using a calculus which implicitly guarantees the correctness of such a derivation. This is based upon the definitions and theorems given in the sequel.

Definition 1 (Redundancy Ordering in Sets). $X \preccurlyeq Y$ (*read* "X is less redundant than Y") *is the cardinality ordering on Sets, that is, the ordering defined by*:

$$X \preccurlyeq Y \stackrel{\text{def}}{=} \exists\, Y \stackrel{\alpha}{\to} X: \alpha \text{ is surjective} \tag{4}$$

Epimorphism α *will be referred to as being a* (*not unique, in general*) "*retrieve map*" *from* Y *to* X. □

For example, it can be stated that, for a finite set X,

$$\mathcal{P}(X) \preccurlyeq X^* \tag{5}$$

($\mathcal{P}(X)$ denotes the set of all finite subsets of X) since $\exists \mathsf{elems}: X^* \to \mathcal{P}(X)$, where

$$\mathsf{elems}\langle a, \ldots, b\rangle = \{a, \ldots, b\}$$

which is a well-known surjective function.

Note that $\preccurlyeq$ is reflexive and transitive, and that $\preccurlyeq$-antisymmetry induces set-theoretical-isomorphism, i.e. for all X, Y and Z in *Sets*, the following facts hold:

$$X \preccurlyeq X \tag{6}$$

$$X \preccurlyeq Y \wedge Y \preccurlyeq Z \;\Rightarrow\; X \preccurlyeq Z \tag{7}$$

$$X \preccurlyeq Y \wedge Y \preccurlyeq X \;\Rightarrow\; X \cong Y \tag{8}$$

Definition 2 (Morphism Refinement). *Let $X \to^{\phi} Y$, $X' \to^{\alpha} X$ and $Y' \to^{\beta} Y$ be morphisms in Sets. Let α and β be epimorphisms ($\Rightarrow X \preccurlyeq X' \wedge Y \preccurlyeq Y'$). Then any morphism $X' \to^{\phi'} Y'$ satisfying the equation*

$$\beta \circ \phi' = \phi \circ \alpha \tag{9}$$

is said to be an $\langle \alpha, \beta \rangle$-refinement of ϕ, cf. Fig. 1.

If $Y = Y'$ then ϕ' is uniquely determined,

$$\phi' = \phi \circ \alpha$$

and is said to be the α-refinement of ϕ. □

Morphism-refinements may be regarded as algorithmic "implementations" induced by the introduction of data-redundancy. For example, let $X = \mathcal{P}(A)$, $X' = A^*$, $Y = Y' = \mathbb{N}_0$, $\alpha = \mathsf{elems}$ and $\phi = \mathsf{card}$, in Definition 2. Then

$$\phi' = \mathsf{card} \circ \mathsf{elems}$$

is the α-refinement of ϕ, and may be regarded as an "implementation" of card, at A^*-level.

Theorem 1 (Refinement Theorem). *Let $\mathcal{A}: \Sigma \to$ Sets be a specification model. Any functor $\mathcal{B}: \Sigma \to$ Sets obtained from $\mathcal{A}$ by object-transformation into "more redundant" objects (Definition 1) and adoption of corresponding "morphism refinements" (Definition 2), is a valid realization of $\mathcal{A}$, i.e. $\mathcal{A} \sqsubseteq \mathcal{B}$ in the complete lattice of all Σ-models [BaW82].*

Proof: Let $s \to^{\sigma} r$ be a Σ-morphism, i.e. a Σ-term denoting an abstract transaction from s-objects into r-objects (including primitive or derived Σ-operators) whose semantics are specified by the *Sets* morphism $\phi = \mathcal{A}(\sigma)$. Since $\mathcal{B}$ is obtained from $\mathcal{A}$ by object-transformation into more-redundant objects, we have:

$$\mathcal{A}(s) \preccurlyeq \mathcal{B}(s) \wedge \mathcal{A}(r) \preccurlyeq \mathcal{B}(r)$$

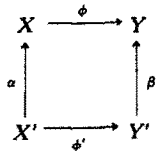

Fig. 1. Morphism refinement

Let $\mathscr{B}(s) \rightarrow^{h_s} \mathscr{A}(s)$ and $\mathscr{B}(r) \rightarrow^{h_r} \mathscr{A}(r)$ be retrieve-maps which record such a relationship. $\phi' = \mathscr{B}(\sigma)$ may be regarded as the "unknown" of our constructive proof. Let ϕ' be a $\langle h_s, h_r \rangle$-refinement of ϕ, i.e.

$$h_r \circ \phi' = \phi \circ h_s$$

that is

$$h_r(\mathscr{B}(\sigma)(x)) = \mathscr{A}(\sigma)(h_s(x))$$

Since h_s and h_r are surjections, this clause means that $h: \mathscr{B} \rightarrow \mathscr{A}(h = \{h_s\}_{s \in S})$ is a Σ-epimorphism. Thus $\mathscr{A} \sqsubseteq \mathscr{B}$ in the complete lattice of all Σ-algebras, that is, algebra $\mathscr{B}$ is an implementation of $\mathscr{A}$. □

This theorem is illustrated by the commutative diagram of Fig. 2, for all Σ-operators $\sigma: s \rightarrow r$, which means:

$$\beta \circ \mathscr{B}(\sigma) \equiv \mathscr{A}(\sigma) \circ \alpha \tag{10}$$

One may say that the function mapping s to α, r to β, and so on (for all Σ-objects and Σ-morphisms) is a *natural transformation* from $\mathscr{B}$ to $\mathscr{A}$ [Oli88a]. Note however, that this natural transformation is not explicitly derived; instead, retrieve-maps are found out first, and the $\mathscr{B}$-morphisms derived next so that the former become a natural transformation.

A simple illustration of Theorem 1 follows.

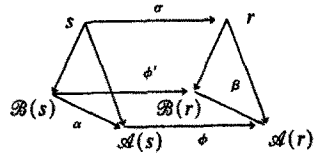

Fig. 2. Refinement diagram

2.3. An Example

Let Σ_{SPELL} be the syntax of a SPELLing module, with sorts ω (word), δ (dictionary) and τ (truth values), involving an operation $Ok: \omega \times \delta \rightarrow \tau$. In terms of semantics, Ok is intended to test whether a given word is correctly spelled according to a given finite dictionary. Let *Words* be the spelling vocabulary (i.e. a set of words), and

$$\mathscr{A}(\omega) = Words$$

$$\mathscr{A}(\delta) = \mathscr{P}(Words)$$

$$\mathscr{A}(\tau) = \{0, 1\}$$

$$\mathscr{A}(Ok) = \lambda(w, d) \,.\, w \in d$$

be the specification-model $\mathscr{A}$ for SPELL. Let

$$\mathscr{B}(\omega) = Words$$

$$\mathscr{B}(\delta) = Words^*$$

$$\mathscr{B}(\tau) = \{0, 1\}$$

Let us apply Theorem 1 to the inference of $\mathscr{B}(Ok)$. We have

$$\alpha = [id, \mathsf{elems}]$$

$$\beta = id$$

adopting an FP-like [Bac78] notation for product-maps; $id = 1_A$ (for every set A) is a "polymorphic" identity operator. Clearly, for every sort $s \in \Sigma_{SPELL}$,

$$\mathscr{A}(s) \preccurlyeq \mathscr{B}(s)$$

cf. equations (5) and (6). According to equation (10), $\mathscr{B}(Ok)$ is any solution to the equation:

$$id \circ \mathscr{B}(Ok) \equiv (\lambda(w, d) \,.\, w \in d) \circ [id, \mathsf{elems}]$$

Since $id \circ f = f$ for all f, we have

$$\begin{aligned}\mathscr{B}(Ok) &\equiv (\lambda(w, d) \,.\, w \in d) \circ (\lambda(x, y) \,.\, \langle x, \mathsf{elems}\ y\rangle) \\ &\equiv \lambda(x, y) \,.\, x \in \mathsf{elems}\ y\end{aligned}$$

as expected.

The properties of the $\langle \{0, 1\};\ \vee, 0\rangle$ *monoid* and the "fold/unfold" method [BuD77, Dar82] can be used to obtain algorithmic solutions for $\mathscr{B}(Ok)$ (cf. [Oli88a] for details), for instance:

$$\mathscr{B}(Ok) \equiv belongs$$

$$belongs\ (x, y) \equiv beloop\ (x, y, 0)$$

$$beloop\ (x, y, b) \stackrel{\text{def}}{=} \textit{if}\ (y = \langle\rangle) \vee b\ \textit{then}\ b\ \textit{else}\ beloop(x, \mathsf{tl}\ y, (x = \mathsf{hd}\ y))$$

which "is" the (expected) *while-loop*:

```
{bool found = 0;
 list p;
 {    p = y;
      while ((p != ⟨⟩) && notfound)
            {p = tl(p);
             found = (x == hd(p))};
 }
}
```

encoded above in an *ad hoc*, "C-like" procedural notation.

Reference [Oli85] presents further examples of transformational operation-refinement in VDM, based on rule (10). A rather elaborate of these examples is the synthesis (towards Pascal) of a procedural realisation of the *apply* operation on *abstract mappings*, implied by the reification of these in terms of *binary trees* (cf. [Fie80]). However, this kind of transformational operation-refinement is strongly dependent on a known, formal relationship between the high-level and the low-level VDM-models – that is to say, a *retrieve-map* such as elems above. How does one "compute" such a relationship?

The remainder of this paper shows how retrieve-maps can themselves be obtained by transformations performed at *Sets*-level. In the SPELL-example, this amounts to showing how to transform $\mathscr{A}(\delta)$ into $\mathscr{B}(\delta)$. In the example of Section 1.1, instead of "guessing" the *BAMS*1-reification for *BAMS*, *BAMS*1 should be actually "derived" from *BAMS*. A *Sets*-transformational calculus will be presented in the sequel which complements the technique described in [Oli85].

3. Introduction to the *Sets* Calculus

It is well-known that *Sets* has a "cartesian closed" structure, i.e. it admits finite-products $(A \times B)$ and finite exponentiations (A^B) for all finite *Sets*-objects A and B:

$$A \times B \stackrel{\text{def}}{=} \{\langle a, b\rangle \mid a \in A \wedge b \in B\}$$

$$A^B \stackrel{\text{def}}{=} \{f \mid f : B \to A\}$$

The empty set $\emptyset$ is said to be the *initial* object 0 of *Sets*. Any singleton set

$$\{0\} \cong \{1\} \cong \ldots \cong \{x\} \tag{11}$$

can be abstracted by the *final Sets*-object 1.[2] Furthermore, *Sets* admits co-products $(A+B)$:

$$A+B \stackrel{\text{def}}{=} (\{1\} \times A) \cup (\{2\} \times B)$$

and solutions to most domain equations of the form

$$X \cong \mathscr{F}(X)$$

for functors $\mathscr{F}$ involving such operations.[3] From exploring such a structure, we obtain useful laws for the transformations we want to perform at *Sets*-level [MaA86]. The first set of laws,

$$A \times B \cong B \times A \tag{12}$$

$$A \times (B \times C) \cong (A \times B) \times C \tag{13}$$

$$A \times 1 \cong A \tag{14}$$

$$A + B \cong B + A \tag{15}$$

$$A + (B + C) \cong (A + B) + C \tag{16}$$

$$A + 0 \cong A \tag{17}$$

$$A \times 0 \cong 0 \tag{18}$$

$$A \times (B + C) \cong (A \times B) + (A \times C) \tag{19}$$

---

[2] See [MaA86] for technical details about the concept of an initial/final object, which will not be developed further in the sequel.

[3] In general, $X \cong \mathscr{F}(X)$ does not always have solutions in *Sets* if exponentiation is allowed. A well known conter-example, due to Scott and Strachey, is $X \cong A + X^X$. [MaA86] give a thorough discussion of this problem, which leads beyond sets to *domains*. However, functors $\mathscr{F}(X)$ involving X in the exponent are unusual in data-type specification, and are of theoretical interest only. As pointed out by [MaA86], one may stay with *Sets* in data-type definition, resorting to domains only in program specification.

establishes that $Sets/\cong$ may be regarded as a commutative semiring under $\times$ and $+$. Concerning exponentiation, one has:

$$A^1 \cong A \tag{20}$$

$$(A \times B)^C \cong A^C \times B^C \tag{21}$$

$$A^{B \times C} \cong (A^C)^B \tag{22}$$

$$1^A \cong 1 \tag{23}$$

$$A^{B+C} \cong A^B \times A^C \tag{24}$$

$$A^0 \cong 1 \tag{25}$$

The *Sets*-object 2

$$2 \cong 1+1$$

is our canonical denotation of $Bool = \{TRUE, FALSE\}$. Clearly, $2 \cong Bool$. At a lower level, other useful facts hold in *Sets*, for example:

$$2^A \cong \mathcal{P}(A) \tag{26}$$

$$A \cap B = \emptyset \;\Rightarrow\; A \cup B \cong A + B \tag{27}$$

$$A^B \cong A^X \times A^{B-X} \;\Leftarrow\; X \subseteq B \tag{28}$$

$$A^n \cong A \times A^{n-1} \tag{29}$$

$$n \neq m \;\Rightarrow\; A^n \cap A^m = \emptyset \tag{30}$$

Law (29) is mere instantiation of law (28), since $n-1 \subseteq n$ (n denotes the initial segment of $\mathbb{N}$ whose cardinality is n).

The *Sets*-relation hierarchy depicted in Fig. 3[4] is based on the following facts, for all A, B in *Sets*:

$$A = B \;\Rightarrow\; A \cong B \tag{31}$$

$$A \cong B \;\Rightarrow\; A \preccurlyeq B \tag{32}$$

$$A \subseteq B \;\Rightarrow\; A \preccurlyeq B \tag{33}$$

The following corollary establishes an obvious connection between Theorem 1 and the isomorphism laws (12) to (29).

Corollary 1 (Object Isomorphism). *Theorem* 1 *holds for object-transformations within Sets-isomorphism.*

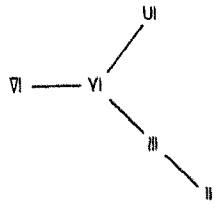

Fig. 3. A hierarchy of relations on *Sets*

[4] The meaning of relation $\preccurlyeq$ will be explained later on, in Section 4.1.

Proof: fact (32). □

By the following theorem, *Sets* may be regarded as a $\preccurlyeq$-ordered algebra.

Theorem 2 ($\preccurlyeq$-Monotonicity of *Sets* Operators). *The operators* $\times$, *exponentiation and* $+$ *are monotone w.r.t. the redundancy-ordering of Definition* 1, *i.e. given Sets-objects A, B, X, Y such that* $A \preccurlyeq X$ *and* $B \preccurlyeq Y$, *then facts*

$$A \times B \preccurlyeq X \times Y \tag{34}$$

$$A + B \preccurlyeq X + Y \tag{35}$$

$$A^B \preccurlyeq X^Y \tag{36}$$

hold.

Proof (*Outline*): Let $X \rightarrow^{\alpha} A$ and $Y \rightarrow^{\beta} B$ be the retrieve-maps corresponding to $A \preccurlyeq X$ and $B \preccurlyeq Y$. Then

Equation (34) - the product-morphism $[\alpha, \beta]$,

$$[\alpha, \beta](\langle x, y \rangle) \stackrel{\text{def}}{=} \langle \alpha(x), \beta(y) \rangle$$

is the retrieve-map between $X \times Y$ and $A \times B$.

Equation (35) - the coproduct-morphism $\alpha \oplus \beta$,

$$\alpha \oplus \beta(x) \stackrel{\text{def}}{=} is\text{-}X(x) \rightarrow \alpha(x)$$
$$is\text{-}Y(x) \rightarrow \beta(x)$$

(where the "is−" predicates are the canonical projections associated to the arguments of a disjoint-union, cf. [Jon80]) is the retrieve-map between $X + Y$ and $A + B$.

Equation (36) - the retrieve-map γ between X^Y and A^B is such that, for each $f \in X^Y$,

$$\alpha \circ f = \gamma(f) \circ \beta$$ □

The following theorem extends $\preccurlyeq$-monotonicity towards recursively defined data domains.

Theorem 3 ($\preccurlyeq$-Monotonicity of *Sets*-Recursion). *Let* $\mathcal{F}$ *and* $\mathcal{G}$ *be two functors in Sets built by composition of the* $\preccurlyeq$*-monotone operators of Theorem* 2. *If*

$$\mathcal{F}(X) \preccurlyeq \mathcal{G}(X) \tag{37}$$

for any X, then the solution to domain equation

$$X \cong \mathcal{G}(X) \tag{38}$$

is a $\preccurlyeq$*-refinement of the solution to*

$$X \cong \mathcal{F}(X) \tag{39}$$

Proof: We will prove that any fixpoint solution $X_{\mathcal{G}}$ to equation (38) is a $\preccurlyeq$-refinement of $\mu\mathcal{F}$, the least fixpoint solution to equation (39). Firstly, if $X_{\mathcal{G}}$ is a solution of $\mathcal{G}$, then $\mathcal{G}(X_{\mathcal{G}}) \cong X_{\mathcal{G}}$, i.e. $\mathcal{G}(X_{\mathcal{G}}) \preccurlyeq X_{\mathcal{G}}$ cf. equation (32). Then

$$\mathcal{F}(X_{\mathcal{G}}) \preccurlyeq X_{\mathcal{G}} \tag{40}$$

by equation (37) and $\preccurlyeq$-transitivity (equation (8)). Since $\mathscr{F}$ involves only monotone operators, $\mathscr{F}$ is also monotone [Man74]. Then we may regard equation (40) as the antecedent of a fixpoint induction argument [Man74], whose consequent is,

$$\mu\mathscr{F} \preccurlyeq X_{\mathscr{G}}$$

completing the proof. □

By Theorems 2 and 3, the components of each data domain of a *Sets* expression can be refined in isolation. This allows for the stepwise introduction of redundancy in formal models of software, towards implementation levels.

Finally, the $\preccurlyeq$-ordering is extended to models in the obvious way. Given a mode $\mathscr{A}$ whose syntax involves a sort s, and a set X such that $\mathscr{A}(s) \preccurlyeq X$, we will write

$$\mathscr{A}[X/s]$$

to denote the model obtained from $\mathscr{A}$ by replacing X for $\mathscr{A}(s)$ and adopting the corresponding morphism-refinements. Clearly, $\mathscr{A}[X/s] \sqsupseteq \mathscr{A}$ in the lattice of all Σ-models (cf. Theorem 1).

3.1. *Sets*-Objects Useful in Specifications

Constructive (model-oriented) specification makes extensive use of *Sets*-constructs. Table 1 shows how some finite object constructions in *Sets* are written in the META-IV, Z and me too notations.

The definitions of A^* and $A \hookrightarrow B$ are as follows:

$$A^* \cong \bigcup_{n=0}^{\infty} A^n \tag{41}$$

$$A \hookrightarrow B \cong \bigcup_{X \subseteq A} B^X \tag{42}$$

The *Sets*-denotation for META-IV *omission* is explained as follows,

$$\begin{aligned} [A] &= A \mid \{NIL\} \\ &\cong A \cup \{NIL\} \\ &\cong A + 1 \end{aligned}$$

from equations (11) and (27) (since $NIL \notin A$ is assumed). Note that it may be convenient to think of mappings in terms of total functions, by introducing an

Table 1. *Sets* versus model-oriented specification notations

Sets	META-IV	Z	me too	Descrption
2^A	$A-set$	PA	set(A)	Finite sets
A^*	$A-list$	seq A	seq(A)	Finite lists
$A \hookrightarrow B$	$A \xrightarrow{m} B$	$A \nrightarrow B$	ff(A, B)	Finite mappings
A×B	A B	A×B	tup(A, B)	Tuples
A+B	A|B			Unions
A+1	[A]			Omissions

undefined value $\perp$, i.e. B in $A \hookrightarrow B$ is extended to $B \cup \{\perp\} \cong B+1$, and one may write:

$$A \hookrightarrow B \cong (B+1)^A \tag{43}$$

4. Examples of Calculated Reification

This section illustrates the purpose of the transformational calculus introduced in the previous sections, with a few examples. A small extension to the calculus will be shown to be necessary in order to accommodate reasoning about data-type invariants.

We begin with a simple example of object transformation geared towards a final encoding into Pascal. It shows how to refine the META-IV domain $A-$*list* (i.e. A^* in *Sets*, cf. Table 1 and equation (41)) into its usual "linked-list" representation:

$$\begin{aligned} A^* &\cong \bigcup_{n=0}^{\infty} A^n \\ &\cong \sum_{n=0}^{\infty} A^n \\ &\cong 1+A+A^2+\ldots \qquad (44) \\ &\cong 1+N \qquad (45) \end{aligned}$$

introducing a variable

$$N = A+A^2+\ldots$$

and resorting to laws (16), (30), (27), (25), (20), (35) and (32). Now,

$$\begin{aligned} N &\cong A\times 1+A\times A+A\times A^2+\ldots \\ &\cong A\times(1+A+A^2+\ldots) \\ &\cong A\times A^* \qquad (46) \end{aligned}$$

resorting to laws (20), (29), (25) and an infinitary version of (19). Step (46) was obtained by "folding" through step (44). In summary,

$$A^* \cong L \tag{47}$$

where $L = 1+N$

$N \cong A\times L$

or

$$A^* \cong 1+A\times A^*$$

cf. steps (45) and (46). An alternative reading of this reasoning is: A^* is an *initial* solution to the equation

$$L \cong 1+A\times L$$

cf. [MaA86].

Finally, the transliteration of (47) into META-IV notation is:

$$L = [N]$$

$$N :: C : A$$

$$P : L$$

which leads to the following Pascal code:

```
L = ^N;
N = record
      C: A;
      P: L
    end;
```

The next example shows how to transform binary relations into abstract mappings, and vice-versa. This is one of a set of results which prove useful in model-refinement towards *relational database* systems. For each relation in the META-IV domain

$$A \overset{m}{\leftrightarrow} B \overset{\text{def}}{=} (A\ B) - set$$

we want to obtain a mapping in $A \rightarrow^m (B - set)$. In *Sets*, one writes $2^{A \times B}$ instead of $A \leftrightarrow^m B$. Moreover,

$$2^{A \times B} \cong 2^{B \times A}$$

$$\cong (2^B)^A \tag{48}$$

cf. laws (12) and (22). Let $2_+^B = 2^B - \{\lambda b.FALSE\}$, where $\lambda b.FALSE$ denotes the *everywhere FALSE* predicate on B, i.e. the predicate which induces the empty set $\emptyset$ on B. Therefore,

$$2_+^B \cong \mathcal{P}(B) - \{\emptyset\}$$

From facts (26), (27) and (11) one draws:

$$2^B = 2_+^B \cup \{\lambda b.FALSE\}$$

$$\cong 2_+^B + 1$$

whereby equation (48) - combined with law (43) - rewrites to:

$$2^{A \times B} \cong (2_+^B + 1)^A$$

$$\cong A \hookrightarrow 2_+^B \tag{49}$$

Equation (49) is an abstract-mapping-level counterpart of equation (22), whose isomorphism can be established by the following bijection (written in META-IV notation):

$$collect : (A \overset{m}{\leftrightarrow} B) \rightarrow (A \overset{m}{\rightarrow} (B - set))$$

$$collect(\rho) \overset{\text{def}}{=} [a \mapsto \{x \in B \mid a \rho x\} \mid \langle a, b \rangle \in \rho] \tag{50}$$

and its inverse *discollect*.

Note that $A \hookrightarrow 2^B_+$ is less general a data-domain than $A \hookrightarrow 2^B$ - which is our target, cf. $A \xrightarrow{m} (B - set)$ - since it does not allow for empty images in mappings. As a matter of fact,

$$2^{A \times B} \preccurlyeq A \hookrightarrow 2^B \tag{51}$$

since - extended to $A \hookrightarrow 2^B$ - *collect* is no longer surjective, and *discollect* is no longer injective. This means that the data domain $A \hookrightarrow 2^B$ can be accepted as a refinement of $A \hookrightarrow 2^B_+$ provided that such a restriction is taken into account. This leads to the notion of a *data-type invariant*, which is discussed in the next section.

4.1. Data-type Invariants

Data-type invariants are Boolean-valued morphisms (predicates) in *Sets* which are required wherever the mathematical definition of a class of data is too generic, and has to be restricted by a *validity predicate* (cf. $inv - BAMS$ in the example of Section 1.1). Data-refinement decisions may lead to adequate low-level data-domains which, however, may contain *invalid* data-representatives. In such cases, data-type invariants are not *intrinsic* to data-domain specification; they are *consequences* of data-refinement. In this context, the redundancy ordering ($\preccurlyeq$) turns out to be too strong, and has to be extended to a "super-redundancy" ordering, defined by

$$X \trianglelefteq Y \overset{\text{def}}{=} \exists S \subseteq Y : X \preccurlyeq S \tag{52}$$

$X \trianglelefteq Y$ may be regarded as meaning that there is a partial surjection from Y to X.

The subset $S \subseteq Y$ in equation (52) is our formal basis for data-type invariant definition and inference: one will say that an invariant, $inv - S$, has been induced upon the refinement of X into Y. Predicate $inv - S$ is easy to define: it simply is the *characteristic function* of S in Y, i.e.

$$inv - S(y) = \begin{cases} TRUE & \text{if } y \in S \\ FALSE & \text{if } y \in Y - S \end{cases}$$

Note that $\preccurlyeq$ is a special case of $\trianglelefteq$, i.e.

$$X \preccurlyeq Y \;\Rightarrow\; X \trianglelefteq Y$$

(make $S = Y$ in equation (52)), the induced invariant being the *everywhere TRUE* predicate on Y, and thus omitted in practice. In general, data-type invariants imply partial morphisms, which become total if restricted to valid data.

The following illustration of $\trianglelefteq$-reasoning is targetted at proving a law,

$$A \hookrightarrow (B \times C) \trianglelefteq (A \hookrightarrow B) \times (A \hookrightarrow C) \tag{53}$$

which is another example of specification-transformation useful in refining towards relational data-models (see example in Section 4.2 later on). This distributive law is the counterpart of law (21), at $\hookrightarrow$-level. Our constructive proof will encompass the inference of the associated low-level data-type invariant. We know that:

$$\begin{aligned} A \hookrightarrow (B \times C) &\cong \bigcup_{K \subseteq A} (B \times C)^K \\ &\cong \bigcup_{K \subseteq A} (B^K) \times (C^K) \\ &\cong \{\langle f, g \rangle | f \in B^K \wedge g \in C^K \wedge K \subseteq A\} \\ &= \{\langle f, g \rangle | f \in A \hookrightarrow B \wedge g \in A \hookrightarrow C \wedge \mathrm{dom}\, f = \mathrm{dom}\, g\} \end{aligned}$$

cf. Table 1 and law (21). Thus, there is $S \subseteq (A \hookrightarrow B) \times (A \hookrightarrow C)$ such that:

$$A \hookrightarrow (B \times C) \cong S$$

and such that $inv-S$ is:

$$inv-S(\langle f, g\rangle) \stackrel{\text{def}}{=} \text{dom}\, f = \text{dom}\, g \tag{54}$$

Therefore,

$$A \hookrightarrow (B \times C) \trianglelefteq (A \hookrightarrow B) \times (A \hookrightarrow C)$$

holds. The corresponding retrieve-map is:

$$retr(\langle f, g\rangle) = f \bowtie g \tag{55}$$

where $\bowtie$ denotes the following "pairing" operator on mappings obeying (54):

$$f \bowtie g \stackrel{\text{def}}{=} [a \mapsto \langle f(a), g(a)\rangle \mid a \in \text{dom}\, f] \tag{56}$$

□

Another basic result useful in relational-database transformations is:

$$A \hookrightarrow B \trianglelefteq 2^{A \times B} \tag{57}$$

which records the well-known fact that *every mapping "is" a relation.* Of course, not all relations are functions. This suggests that the associated invariant should express a functional-dependence. In fact,

$$\begin{aligned} A \hookrightarrow B &\cong \{\rho \subseteq A \times B \mid \forall \langle a, b\rangle, \langle a', b'\rangle \in \rho : (a = a' \Rightarrow b = b')\} \\ &= \{\rho \in 2^{A \times B} \mid fdp(\rho)\} \\ &\trianglelefteq 2^{A \times B} \end{aligned} \tag{58}$$

where a predicate $fdp(\rho)$, introduced as an abreviation of the universal quantifier of equation (58), defines the induced invariant over $2^{A \times B}$. This is written in META-IV as follows:

$$fdp : (A \stackrel{m}{\leftrightarrow} B) \to Bool$$

$$fdp(\rho) \stackrel{\text{def}}{=} \forall \langle a, b\rangle, \langle a', b'\rangle \in \rho : (a = a' \Rightarrow b = b') \tag{59}$$

A valid retrieve-map for this $\trianglelefteq$-relationship is:

$$mkf : (A \stackrel{m}{\leftrightarrow} B) \to (A \stackrel{m}{\to} B)$$

$$mkf(\rho) \stackrel{\text{def}}{=} [a \mapsto b \mid a \in dom\,(\rho) \wedge b \in B \wedge a \rho b] \tag{60}$$

which is well-defined for every relation ρ satisfying (59) (*dom* is the operator defined above by equation (2)). □

4.2. Systematic Inference of Retrieve Functions and Data-type Invariants

Similarly to the redundancy ordering ($\preccurlyeq$), the super-redundancy ordering ($\trianglelefteq$) introduced in Section 4.1 is reflexive and transitive,

$$X \trianglelefteq X$$

$$X \trianglelefteq Y \wedge Y \trianglelefteq Z \Rightarrow X \trianglelefteq Z$$

and compatible with *Sets*-operators, i.e.:

$$A \times B \trianglelefteq X \times Y \tag{61}$$

$$A + B \trianglelefteq X + Y \tag{62}$$

$$A^B \trianglelefteq X^Y \tag{63}$$

for $A \trianglelefteq X$ and $B \trianglelefteq Y$ (the retrieve-maps and data-invariants being obtained in a way similar to Theorem 2). This means that both data-type invariants and retrieve-maps can be inferred in a stepwise, structural manner. For a chain of $n \trianglelefteq$-steps, involving n retrieve-maps $retr_i$ $(i = 1, \ldots, n)$ and n invariants inv_i $(i = 1, \ldots, n)$, the overall retrieve-map is obtained by:

$$retr = \bigcirc_{i=1}^{n} retr_i \tag{64}$$

and the overall invariant is obtained by:

$$inv = \lambda x. \bigwedge_{i=1}^{n} inv_i((\bigcirc_{j=i+1}^{n} retr_j)(x)) \tag{65}$$

In summary, the systematic inference of retrieve-maps (between models) is achieved by structural composition of morphisms implicit in the *Sets*-rules presented above. Wherever $\trianglelefteq$-reasoning is involved, data-type invariants are synthesised in a similar way, together with retrieve-maps. This is illustrated in the following, final example.

We want to transform *BAMS* into *BAMS*1 (cf. Section 1.1) and infer the corresponding retrieve-map and induced data-type invariant. The *Sets*-notation for the META-IV syntax of *BAMS* is,

$$BAMS = AccNr \xrightarrow{} (2_+^{AccHolder} \times Amount)$$

where $2_+^{AccHolder}$, instead of $2^{AccHolder}$, takes $inv - BAMS$ into account. Using laws (53), (49) and (57), *BAMS* is subject to transformational reasoning,

$$\begin{aligned} BAMS &= AccNr \xrightarrow{} (2_+^{AccHolder} \times Amount) \\ &\trianglelefteq (AccNr \xrightarrow{} 2_+^{AccHolder}) \times (AccNr \xrightarrow{} Amount) \\ &\cong (2^{AccNr \times AccHolder}) \times (AccNr \xrightarrow{} Amount) \\ &\trianglelefteq (2^{AccNr \times AccHolder}) \times (2^{AccNr \times Amount}) \\ &= BAMS1 \end{aligned}$$

which has led to *BAMS*1 in an easy way. The first $\trianglelefteq$-step induces an invariant:

$$inv_1(\langle f, g \rangle) \stackrel{\text{def}}{=} \mathrm{dom}\, f = \mathrm{dom}\, g$$

matching with the retrieve-map:

$$retr_1(\langle f, g \rangle) \stackrel{\text{def}}{=} [a \mapsto \langle f(a), g(a) \rangle \mid a \in \mathrm{dom}\, f]$$

cf. equations (55) and (56). The subsequent $\cong$-step induces the retrieve-map:

$$retr_2 \stackrel{\text{def}}{=} [collect, id]$$

cf. equations (34) and (50). The last $\trianglelefteq$-step induces invariant (59) on the second argument:

$$inv_3(\langle \rho, \sigma \rangle) \stackrel{\text{def}}{=} fdp(\sigma)$$

and the retrieve-map:

$$retr_3 \stackrel{\text{def}}{=} [id, mkf]$$

cf. equation (60). The overall retrieve-map is obtained by chained morphism-composition, cf. rule (64):

$$\begin{aligned} retr_1 \circ retr_2 \circ retr_3 &= retr_1 \circ retr_2 \circ [id, mkf] \\ &= retr_1 \circ [collect, id] \circ [id, mkf] \\ &= retr_1 \circ [collect, mkf] \\ &= \bowtie \circ [collect, mkf] \\ &= \lambda \langle \rho, \sigma \rangle .\ let\ f = collect(\rho) \\ &\qquad in\ [a \mapsto \langle f(a), mkf(\sigma)(a) \rangle \mid a \in \mathrm{dom}\, f] \end{aligned}$$

The overall data-type invariant is obtained using rule (65). Writing $inv(\langle \rho, \sigma \rangle)$ as a shorthand for $inv - BAMS1(mk - BAMS1(\rho, \sigma))$, one has:

$$\begin{aligned} inv(\langle \rho, \sigma \rangle) &= inv_1(retr_2(retr_3(\rho, \sigma))) \wedge \\ &\quad\ inv_2(retr_3(\rho, \sigma)) \wedge \\ &\quad\ inv_3(\rho, \sigma) \\ &= inv_1(retr_2(\rho, mkf(\sigma))) \wedge \\ &\quad\ TRUE \wedge \\ &\quad\ fdp(\sigma) \\ &= inv_1(collect(\rho), mkf(\sigma)) \wedge \\ &\quad\ fdp(\sigma) \\ &= (\mathrm{dom}\ collect(\rho) = \mathrm{dom}\ mkf(\sigma)) \wedge \\ &\quad\ fdp(\sigma) \\ &= (dom(\rho) = dom(\sigma)) \wedge fdp(\sigma) \end{aligned}$$

that is, the same invariant as postulated by equation (1). The last step above relies on two simple facts relating the relation-domain operator (equation (2)) and the META-IV dom mapping-operator:

$$\mathrm{dom}\ collect(\rho) = dom(\rho)$$

$$\mathrm{dom}\ mkf(\rho) = dom(\rho)$$

Note in passing that we were saved from writing explicit proofs for two standard VDM verification-steps about retrieve-maps, *adequacy* and *totality* over valid data, which are implicitly guaranteed by the whole transformational process.

5. Conclusions

The main motivation for the work described in this paper has been the need for "proof discharge" strategies in formal methods for software design. It is suggested that the transformational paradigm [BaW82, Dar82] should be extended to the refinement of model-oriented specifications, and shown how program transformation leads to model transformation in a natural way. A set-theoretical basis for a comprehensive reification calculus handling data-structure transformation is presented, whereby efficiency is gradually induced into algorithms, in a controlled way.

Such a transformational calculus is applicable to methodologies such as VDM, matching with a transformation-style formerly proposed, at operation-level, in [Oli85]. Following its rules in a structured way,

Retrieve-maps and lower-level data-type invariants are systematically synthesised.

Data-type invariants are deduced by formal reasoning instead of being stated in an *ad hoc* way; this means that there is little danger of over-strengthening them, in which case proofs may become over-complicated.

Standard proofs about retrieve-maps such as adequacy and totality over valid data are not required because they are implicitly guaranteed by the method.

It should be stressed that a formal notion of "model redundancy" (and associated calculus) is useful at specification-level itself. In fact, it enables the software engineer to decide upon "better models" for his/her specifications. For instance, suppose that two domains A or B seem adequate as the semantic domain $\mathscr{A}(s)$ for some syntactic domain s (in a software model $\mathscr{A}$), and that $A \trianglelefteq B$. Then $\mathscr{A}[A/s]$ - the model obtained by making $\mathscr{A}(s) = A$ - will be a "better" model than $\mathscr{A}[B/s]$ - *mutatis mutandis* $\mathscr{A}(s) = B$. The latter model would require spurious data-type invariants and would involve too complex morphism specifications. In this context, $\trianglelefteq$-reasoning proceeds in reverse order: given a rule $L \trianglelefteq R$, an instance of R is replaced by the corresponding instance of L, obtaining more abstract data-domains while removing unnecessary data-type invariants.

A "laboratory" version of our model-algebra has been successfully applied to a sizeable case-study [MRJ88] for industry. Real examples such as this are relevant because theoretical results need feedback from practice. For example, new transformation rules were found out throughout the exercise reported in [MRJ88]. Reference [Oli89a] shows how the calculus can be applied to the transformation of VDM-specification models into object-oriented modules.

6. Future Work

This is work under progress and requires further research in several respects:

1. The calculus described in this paper is still in its infancy. Further laws and results are required before it becomes a pratical tool for imperative software development. For example, [Oli88b] refers to current research on laws for *recursion removal* from data-structures, by introduction of *pointers*, *keys* or *names* typical of imperative programming (including database design and object-oriented programming), for instance, the law

$$A \cong \mathscr{F}(A) \quad \trianglelefteq \quad K \times K \hookrightarrow \mathscr{F}(K)$$

which makes "pointers" (K) to "heaps" ($K \hookrightarrow \mathcal{F}(K)$) explicit. Such laws induce fairly elaborate invariants and retrieve functions, because of the danger of nontermination and/or pointer undefinedness.

The exercise reported in [MRJ88] suggests that *normal-form theory* can perhaps be regarded as a sub-calculus of the reification calculus. This should be investigated. A limitation of the calculus developed so far is that all transformations are "context-free", in that they do not take invariants into account. For instance, in the *BAMS*1-refinement of the *BAMS*-syntax (cf. Section 1.1), the specifier might wish to save space in the amounts-table by removing all entries whose amount is 0:

$$Row2 :: K : AccNr\, A : (Amount - \{0\})$$

leading to a weaker version of formula (1):

$$inv\text{-}BAMS1(mk\text{-}BAMS1(ht, at)) \stackrel{\text{def}}{=} dom(at) \subseteq dom(ht) \wedge dep KA(at) \tag{66}$$

cf. [Oli88a]. The "invariant-sensitive" rule required by the transformation of 1 into 66 is the following: let S be the subset $S \subseteq (A \hookrightarrow B) \times (A \hookrightarrow C)$ induced by invariant (54), and let R be the subset $R \subseteq (A \hookrightarrow B) \times (A \hookrightarrow (C - \{c\}))$, where $c \in C$, induced by

$$inv\text{-}R(\langle f, g\rangle) \stackrel{\text{def}}{=} \mathrm{dom}\, f \subseteq \mathrm{dom}\, g$$

Then $S \preccurlyeq R$ with

$$retr(\langle f, g\rangle) \stackrel{\text{def}}{=} \langle f, g \sqcup [a \mapsto c \mid a \in \mathrm{dom}\, f - \mathrm{dom}\, g]\rangle$$

"Invariant-sensitive" transformations such as above seem to be common in VDM, and should be studied in detail.

2. At operation-level, the *pre-/post-condition* style of specification (which is able to express non-deterministic behaviour) is dealt with by regarding such conditions as Boolean-valued morphisms. For instance, if

   $$post\text{-}OP : \Sigma A \Sigma \to Bool$$

   is a post-condition on a class of Σ-states, accepting arguments in a class A, and $retr_\Sigma$ and $retr_A$ are (respectively) the retrieve-maps implicit in two given refinement decisions, $\Sigma \preccurlyeq \Omega$ and $A \preccurlyeq B$, then the implied reification of OP,

   $$post\text{-}OP1 : \Omega B \Omega \to Bool$$

   is any solution to the equation

   $$post\text{-}OP1(\omega, b, \omega') \Rightarrow post\text{-}OP(retr_\Sigma(\omega), retr_A(b), retr_\Sigma(\omega'))$$

   However, it may be preferable to redefine the very notions of *signature* and *model* in order to accommodate "procedural" formal specifications, cf. e.g. [NiP86, Fia89, Oli89b]. [NiP86] generalises the model-theoretic basis for data types from algebras to multi-algebras, introducing the notion of a *nondeterministic data type* and providing it with a basis for correctness of implementations. The relationship between abstract and concrete data is recorded in terms of relations rather than functions. At data-domain

functions. At data-domain level, the calculus presented in this paper is applicable to this wider notion of an implementation. However, special attention should be paid to the implications of generalizing homomorphisms to behavioural simulations. [Oli89b] resorts to CCS [Mil89] to incorporate behaviour in VDM-modules, approaching the expressive power of object-oriented specification. [Fia 89] develops *modal logic* framework for algebraic, object-oriented specification.

3. The rudimentary category-theoretical foundations of the original approach [Oli88a] should be better exploited. We believe that a more thorough supported on category theory (following [MaA86], for instance) may significantly improve it. In particular, the formalisms dependent on *Sets* should be generalised to other cartesian-closed categories with co-products. *Sets* is perhaps too restricted a category for formal specification of imperative software models. Alternative, object-oriented approaches to formal specification are being investigated, cf. e.g. FOOPS [GoM87]. Reference [SFS87] describes a categorical approach to object-oriented specification.

4. Past work on dataflow program semantics [Oli84] showed the advantage of variableless, function-level notations (such as FP [Bac78]) in program transformation, because of their compactness and associated algebra of programs. The present research has increased our interest on such notations, because of their strong connections with the "morphism-language" of category theory (see also the *f-NDP* notation of [Val87]).

5. The relationship between this approach and Hoare [Hoa87]'s categorical setting for data refinement should be investigated.

The introduction of algebraic reification-calculi in software engineering appears to be a natural evolution, when compared with the historical development of the scientific bases for other engineering areas (e.g. civil and mechanical engineering etc.) which, some centuries ago, started omitting complicated geometrical proofs in favour of algebraic reasoning. The reader is left with the following quotation by a Portuguese mathematician of the 16th century, when classic algebra was emerging and started being applied to practical problems:

> "Quien sabe por Algebra, sabe scientificamente." Pedro Nunes (1502–1578), *in libro de algebra*, 1567, fol. 270v.

Acknowledgements

The author is indebted to the referees for detailed and helpful comments which improved the paper's presentation and technical contents. In particular, the invariant-sensitive kind of transformation was pointed out by one of the referees.

Special thanks go to Prof. Cliff Jones for his comments and interest in this paper.

This research was partially supported by JNICT, under PMCT-contract Nr.87.64.

References

[Bac78] Backus, J.: Can Programming Be Liberated from the von Neumann Style? A Functional Style and Its Algebra of Programs. *CACM*, 21(8), 613-641 (1978).

[BaW82] Bauer, F. L. and Wössner, H.: *Algorithmic Language and Program Development.* Springer-Verlag, 1982.

[BuD77] Burstall, R. M. and Darlington J.: A Transformation System for Developing Recursive Programs. *JACM*, 24(1), 44-67 (1977).

[Dar82] Darlington, J.: Program Transformation. *Functional Programming and Its Applications: An Advanced Course*, Newcastle University, Cambridge University Press, 1982.

[Fia89] Fiadeiro, J. L.: Cálculo de Objectos e Eventos. Ph.D. dissertation, IST-University of Lisbon, Portugal, 1989 (in Portuguese).

[Fi80] Fielding, E.: The Specification of Abstract Mappings and Their Implementation as B^+-Trees. PRG-18, Oxford University, September 1980.

[GoM87] Goguen, J. and Meseguer, J.: *Unifying Functional, Object-oriented and Relational Programming with Logical Semantics.* SRI International, 1987.

[GoT78] Goguen, J., Thatcher, J. W. and Wagner, E. G.: *Initial Algebra Approach to the Specification, Correctness and Implementation of Algebraic Data Types.* Current Trends in Programming Technology, Vol. IV, Prentice-Hall, 1978.

[GuH78] Guttag, J. V. and Horning, J. J.: The Algebraic Specification of Abstract Data Types. *Acta Informatica*, 10, 27-52 (1978).

[Hay87] Hayes, I. (Ed.): *Specification Case Studies.* Series in Computer Science, C. A. R. Hoare (ed.), Prentice-Hall International, 1987.

[Hen84] Henderson, P.: me too: A Languare for Software Specification and Model-Building - Preliminary Report. University of Stirling, December 1984.

[Hoa87] Hoare, C. A. R.: Data Refinement in a Categorical Setting. PRG, Oxford University, June 1987.

[HJS87] Hoare, C. A. R., Jifeng, He and Sanders, J. W.: Prespecification in Data Refinement. *Information Processing Letters*, 25, 71-76 (1987).

[Jon80] Jones, C. B.: *Software Development - a Rigorous Approach.* Series in Computer Science, C. A. R. Hoare (ed.), Prentice-Hall International, 1980.

[Joa86] Jones, C. B.: *Systematic Software Development Using VDM.* Series in Computer Science, C. A. R. Hoare (ed.), Prentice-Hall International, 1986.

[Mac71] MacLane, S.: *Categories for the Working Mathematician.* Springer-Verlag, New-York, 1971.

[MaA86] Manes, E. G. and Arbib, M. A.: *Algebraic Approaches to Program Semantics.* Texts and Monographs in Computer Science, D. Gries (ed.), Springer-Verlag, 1986.

[Man74] Manna, Z.: *Mathematical Theory of Computation.* McGraw-Hill, New York, 1974.

[Mil89] Milner, R.: *Communication and Concurrency.* Series in Computer Science, C. A. R. Hoare (ed.), Prentice-Hall International, 1989.

[MRJ88] Moreira, C., Reis, A., Jesus, R. and Barros, A.: Especificação Formal de um Sistema para Conservatórias de Registo Predial. U. Minho, Technical Report CCES-OP3:R1/88, May 1988 (in Portuguese).

[Nip86] Nipkow, T.: Nondeterministic Data Types: Models and Implementations. *Acta Informatica*, 22, 629-661 (1986).

[Oli85] Oliveira, J. N.: The Transformational Paradigm as a Means of Smoothing Abrupt Software Design Steps. U. Minho, Technical Report CCES-JNO:R2/85, December 1985.

[Oli87] Oliveira, J. N.: Refinamento Transformacional de Especificações (Terminais). *Proc. XII Iberian "Jornadas" on Mathematics*, May 1987, Braga, Portugal (in Portuguese).

[Oli88a] Oliveira, J. N.: Transformational Refinement of Formal (Model-oriented) Specifications. Internal Report CCES-JNO:R1/87, University of Minho, Braga, Portugal (updated: June 1988).

[Oli88b] Oliveira, J. N.: *SETS* - Uma Linguagem de Especificação Quase Centenária. CCES Seminar (1988/89 series), University of Minho, July 1988 (in Portuguese).

[Oli89a] Oliveira, J. N.: Transforming Specifications: Between the Model-oriented and the Object-oriented Style. CCES, U. Minho, Technical Report (in preparation).

[Oli89b] Oliveira, J. N.: Algebraic Specification in a CCS-Extension to Modular-VDM. Invited communication, Section C4 (Computing Science Foundations), 1989 *National Meeting of the Portuguese Mathematical Society, Oporto*, 3 April 1989.

[SFS87] Sernadas, A., Fiadeiro, J., Sernadas, C. and Ehrich, H.-D.: Abstract Object Types: A Temporal Perspective. *Colloquium on Temporal Logic and Specification*, A. Pnueli, H. Barringer and Banieqbal, B. (eds), Springer-Verlag.

[Spi89] Spivey, J. M.: *The Z Notation - A Reference Manual.* Series in Computer Science, C. A. R. Hoare (ed.), Prentice-Hall International, 1989.

[Val87] Valença, J. M.: Formal Programming: an Algebraic Approach - Part 1: An Algebra of Functions and a Semantics for Imperative Languages. Invited paper, *Proc. XII Iberian "Jornadas" on Mathematics,* May 1987, Braga, Portugal.

[Wan79] Wand, M.: Final Algebraic Semantics and Data Type Extensions. *JCSS,* 19, 27–44 (1979).

Received February 1989
Accepted in a revised form in October 1989 by C. B. Jones