Software Reification using the SETS Calculus

José N. Oliveira
DI/INESC, Universidade do Minho
Braga, Portugal

Abstract

SETS is an emerging reification calculus for the derivation of implementations of *model-oriented* specifications of abstract data types.

This paper shows how *abstraction invariants* can be synthesized by calculation in SETS, and the potential of this calculus for assessing, comparing or classifying specifications.

The main results of the paper are concerned with a functorial approach to reification, particularly *wrt.* the systematic implementation of recursive data domains on non-recursive run-time environments. A final example of this class of implementations is provided.

1 Introduction

Research in software technology has shown the need for splitting software design in two complementary steps: *formal specification* first (in which a mathematical text is written prescribing "what" the intended software system should do) and then *implementation* (in which machine code is produced instructing the hardware about "how" to do it).

In general, there is more than one way in which a particular machine can accomplish "what" the specifier bore in mind. Thus, the relationship between specifications and implementations is one-to-many, that is, specifications are more *abstract* than implementations. In fact, specifications are intended to be read, understood and reasoned about by humans. They prescribe behaviour rather than describe it. Implementations are intended to be executed by machines, as efficiently as possible. The programming "tricks" introduced for the sake of efficiency (in implementations) are thus irrelevant details at specification level. In summary, the "epistemological gap" between specifications and implementations is far from being a "smooth" one and is the major concern of the so-called *reification* (or *refinement*) technology, a recent branch of software engineering using formal methods.

Of course, one wants an implementation to behave exactly in the way prescribed by its specification. Thus the notion of *formal correctness* is central to any reliable reification discipline.

In the well-known *constructive style* for software development [19, 20] design is factored into as many "mind-sized" design steps as required. Every intermediate design is first proposed and then proved to follow from its antecedent. Despite improving the primitive approach to correctness (full implementation prior to the overall correctness argument), such an "invent-and-verify" style is often impractical due to the complexity of the mathematical reasoning involved in real-life software problems.

Recent research seems to point at alternative reification styles. The idea is to develop a *calculus* allowing *programs* to be actually calculated from their specifications. In this approach, an intermediate design is drawn from a previous design according to some *law* available in the calculus, which must be *structural* in order for the components of an expression to be refined in isolation (*i.e.* pre-existing refinement results can be "re-used"). Proof discharge is achieved by performing structural calculation instead of proofs from first principles. This is the point of a calculus, as witnessed elsewhere in the past (*cf.* the differential and integral calculi, linear algebra, *etc.*). After a decade of intensive research on the foundations of *formal methods* for software design, one is tempted to forecast that the 1990s will witness the maturation phase of software technology, developing and using tools based on formal calculi.

The target of this paper is to describe the evolution of a particular reification calculus — Sets — whose foundations can be found elsewhere [34, 35, 22]. We start by overviewing related work on reification calculi. For conciseness, this overview is concerned only with calculi for *static semantics*, that is, the very important area of *event reification* is deliberately left out. (The reader is referred to *e.g.* [26, 2, 18, 44, 21] for results in this area.) We proceed to a summary of Sets taken from [35] but adding further basic results. The main results of the paper are presented in section 4 and have to do with reasoning about recursive data types. The scope of these results is illustrated in section 5. The last two sections contain some conclusions and suggestions for future work.

2 Overview of Reification Calculi for Software Design

Wirth's "formula" *Algorithms + Data Structures = Programs* [46] has become famous as the title of a mandatory textbook on structured programming since the mid 1970s. It may be regarded as being probably the first, widespread message that *programs "are" algebraic structures* — *cf. Operators + Sets = Algebras* — which has become the 'moto' of a vast amount of research undertaken in the last two decades on applying universal algebra to computer programming, namely to the formalization of *abstract data types* [14, 12], to *denotational semantics* [11, 31], to *concurrent processing* [25, 16, 15, 27], to *formal specification languages* [5, 28] *etc.*

One of the schools of thought in algebraic specification is termed *model oriented*, or *constructive* because specifications are written explicitly as *models*, *i.e.* algebras, instead of being axiomatically defined. Constructive specification languages (such as Vdm [20] and Z [43]) allow nondeterministic operators and local states, and work with *relational algebras* [32], that is, the above "formula" becomes *Relations + Sets = (Relational) Algebras*.

The algebraic-model approach to programming has the benefit that, whatever is said about (or happens to) programs, can always be decomposed across two complementary axes: the *data* programs deal with (*i.e.* data-structures) and the *operations* which manipulate them (*e.g.* procedures, functions *etc.*).

Reification is no exception in this respect. The way the Vdm practitioner proceeds from abstract specifications to more concrete models (closer and closer to a target machine, or available programming language) is by "model-refinement". VDM recommends that one of the above axes — *data-refinement* — be considered first, possibly encompassing several iterations. Once a satisfactory *implementation-model* is reached, refinement decisions are then taken on the orthogonal direction (*algorithmic-refinement*) so as to, eventually, reach executable code (*e.g.* in PL/1, PASCAL *etc.*).

This view is in slight contrast with [30, 29], where *data-refinement* is regarded as a special case of *algorithmic-refinement*, being the action of replacing an abstract type by a more concrete type in a program, while preserving its algorithmic structure. The corresponding refinement calculus stems from an extension of Dijkstra's calculus [9] of *predicate transformers*. Such a "calculational style" was first introduced by [17] in a relational setting.

Backus' Turing Award paper [3] has become the 'ex-libris' of a vast collection of works on calculating algorithmic refinements in a *functional* setting, *cf.* the well-known *program transformation* school (see for instance [4, 7, 28]). Reference [7] is one of the first in the literature to characterize functional data-refinement by calculation (transformation). Reference [33] shows how to apply this strategy in the relational (*pre/post*-condition) context of the VDM methodology.

However, a slight difficulty persists in both relational and functional data-refinement: one has to choose (*i.e.* guess) the *abstraction invariant* which links abstract values to concrete values. Such an invariant, which in the functional style can be factored into a *concrete invariant* and an *abstraction function* [30], can be hard to formulate in practical, realistic examples. It would be preferable to be able to *calculate* such an invariant. That is to say, one needs calculi for the stepwise refinement of the data themselves.

This is precisely the main target of the SETS calculus which is the subject of this paper. Because its emphasis is on data structuring laws independently of algorithmic control, we regard it as a "pure" data-refinement calculus when compared to the approaches described above. It is grounded on elementary properties of the cartesian closed category of *Sets* [24] which underlies formal specification in the *constructive* (model-oriented) style. It is therefore easy to understand and to use in reasoning about software models. Following the pragmatical presentation of [35], most category theoretical notions will be replaced by set-theoretical ones.

3 Overview of the SETS Calculus

In this section we present a summary of SETS which is sufficient for the understanding of the main results of this paper (see [35] for further details). Some basic algebraic terminology is required first. For space economy, we stick to "functional" algebraic notation, the appropriate generalization to *non-deterministic* or *relational* algebras being available from *e.g.* [32].

3.1 Basic Terminology

Given a set Ω of function symbols, and a set S of *sorts* ("types"), a *signature* Σ is a syntactical assignment $\Sigma : \Omega \rightarrow (S^\star \times S)$ of a functionality to each function symbol; as usual, we will write $\sigma : s_1 \ldots s_n \rightarrow s$ or $s_1 \ldots s_n \stackrel{\sigma}{\rightarrow} s$ as shorthands of $\Sigma(\sigma) = \langle <s_1, \ldots, s_n>, s\rangle$. Let *Sets* denote the class of all finite or denumerable sets [1] operated by set-theoretical functions. Let these be denoted by $f : X \to Y$ or $X \xrightarrow{f} Y$, where X and Y are sets.

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

$$\mathcal{A} : \Sigma \to Sets$$

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

$$\mathcal{A}_\Omega(\sigma) : \mathcal{A}_S(s_1) \times \ldots \times \mathcal{A}_S(s_n) \to \mathcal{A}_S(s) \tag{1}$$

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

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

instead of expression (1).

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:

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

where

$$C(s) \stackrel{\text{def}}{=} \{\sigma \in \Omega |\ \Sigma(\sigma) = \langle <>, s \rangle\}$$

is the set of all "constants" of type s.

In model-oriented specification, a software module is specified by producing a *model*, which is just a Σ-algebra $\mathcal{A}$, for a particular signature Σ describing its syntactical structure.

3.2 Underlying Formalisms

Given a specification model $\mathcal{A} : \Sigma \to Sets$, a *reified model* (or refinement) of $\mathcal{A}$ is any other model $\mathcal{B} : \Sigma \to Sets$ such that there is an epimorphism (*abstraction map*) from $\mathcal{B}$ to $\mathcal{A}$. In general, $\mathcal{B}$ is more *redundant* a model than $\mathcal{A}$, in the sense that a given abstract value $x \in \mathcal{A}(s)$, for some sort $s \in S$, may be represented (reified, refined) in $\mathcal{B}$ by more than one value in $\mathcal{B}(s)$. Take, for instance, the usual reification of finite sets in terms of arbitrary finite lists. The empty set $\emptyset$ is implemented by a single list value, the empty list $<>$. But any other set, *e.g.* $\{a, b\}$, has many list representatives, *e.g.* $<a, b>$, $<b, a>$, $<b, a, b>$, the abstraction map being the usual $elems$ function [19].

The collection of all (surjective) abstraction maps, one for each sort $s \in S$, makes up an epimorphism $h : \mathcal{B} \longrightarrow \mathcal{A}$ [2]. The algorithmic structure of $\mathcal{A}$ is

[1] Following the terminology of [1], by a *denumerable* set A we mean a set whose cardinal number $card(A)$ equals $\aleph_0 = card(\mathbb{N})$.

[2] The so-called *final* approach to reification [45] imposes h to be unique. See section 7 for a discussion about this point.

preserved by $\mathcal{B}$ in the sense that every epimorphism h is a homomorphism, and thus,

$$\mathcal{A}(\sigma)(h(b_1), \ldots, h(b_n)) = h(\mathcal{B}(\sigma)(b_1, \ldots, b_n)) \tag{2}$$

for every n-ary Σ-operator σ applied to (correctly type-checked) n-arguments in $\mathcal{B}$ [3].

In our calculational style, instead of conjecturing $\mathcal{B} : \Sigma \to Sets$ and proving that there is an epimorphism from $\mathcal{B}$ to $\mathcal{A}$, we want to effectively calculate $\mathcal{B}$ from $\mathcal{A}$, gradually synthesizing the appropriate epimorphism, as explained in the sequel. We will concentrate on data-level transformations, for the sake of brevity. See [35] for a refinement theorem which fits such transformations to the preservation of algorithmic behaviour. Section 5 will provide an illustration of model calculation in SETS, combining data-level and operation-level transformations.

3.3 On Redundancy Orderings

The SETS calculus is "naively" based on the cardinality ordering $A \preceq B$ among sets:

$$A \preceq B \stackrel{\text{def}}{=} \exists B \xrightarrow{f} A : f \textit{ is surjective.} \tag{3}$$

where f plays the rôle of an *abstraction map* such as discussed above (section 3.2). We will write $A \preceq_f B$ wherever we want to keep track of abstraction maps in $\preceq$-reasoning. For instance,

$$2^A \preceq_{elems} A^\star \tag{4}$$

for A a finite set. The ordering $\preceq$ is reflexive and transitive, and $\preceq$-antisymmetry induces set-theoretical isomorphism [35]. Transitivity means that one can chain $\preceq$-steps and synthesize overall abstraction maps, that is

$$A \preceq_f B \wedge B \preceq_g C \Rightarrow A \preceq_{f \circ g} C \tag{5}$$

The finitary version of (5), for n $\preceq$-reasoning steps, is

$$\underbrace{A \preceq_{f_1} A_1 \preceq_{f_2} \cdots \preceq_{f_n} A_n}_{f = f_1 \circ \cdots \circ f_n}$$

Reflexivity is naturally established by identity maps,

$$A \preceq_{1_A} A$$

while $\preceq$-antisymmetry

$$A \preceq_f B \wedge B \preceq_g A \Rightarrow A \cong B$$

enforces two isomorphisms f, g for finite A, B ($g = f^{-1}$, in particular).

Unfortunately, many reification steps cannot be described by facts of the form $A \preceq B$, because B is "too wide" a data model and contains invalid

[3] For improved readability, subscripts S, Ω, s, $s_1, \ldots, s_n$ have been omitted from (2), *cf.* [35].

representatives of A. For instance, there is an implementation of sets in terms of lists which is "finer" than (4) because one wants to leave out lists which contain repeated elements (for the sake of efficiency, maybe). The basic fact we can assert in this case is

$$2^A \preceq_{elems} \{l \in A^\star \mid inv(l)\} \tag{6}$$

where

$$\begin{array}{rcl} inv : A^\star & \longrightarrow & 2 \\ l & \rightsquigarrow & len(l) = card(elems(l)) \end{array} \tag{7}$$

where $2 \cong \{TRUE, FALSE\}$ denotes the set of Boolean values, and len, $elems$ and $card$ are well-known list and set operators [19].

Validity predicates such as inv (7) have become known in the literature as *data-type invariants*. Because of the need for data-type invariants, the ordering above is too "strong". As is explained in [35], (3) may be superseded by the ordering

$$A \trianglelefteq B \stackrel{\text{def}}{=} \exists S \subseteq B : A \preceq_f S. \tag{8}$$

Let ϕ denote the characteristic function of S in B [35]. Then

$$A \trianglelefteq_f^\phi B \tag{9}$$

may be written to mean exactly the same thing as (8). For instance, instead of (6), we will write

$$2^A \trianglelefteq_{elems}^{inv} A^\star \tag{10}$$

Subscripts f, ϕ in (9) may be omitted wherever implicit in the context. Invariant ϕ determines the *domain* of the abstraction map f, which can be regarded as a *partial* surjection from B to A [4].

This wider notion of a redundancy ordering ($\trianglelefteq$) enjoys the same properties as $\preceq$, *e.g.*

$$A \trianglelefteq_{1_A}^{\lambda a.TRUE} A$$

etc. It is relevant to see how invariants are synthesized by transitivity:

$$A \trianglelefteq_f^\phi B \wedge B \trianglelefteq_g^\gamma C \Rightarrow A \trianglelefteq_{f \circ g}^\rho C$$

for

$$\rho(c) = \gamma(c) \wedge \phi(g(c)) \tag{11}$$

[4] Strictly speaking, the $\preceq$ and $\trianglelefteq$ orderings are mathematically equivalent, since $A \preceq B \Rightarrow A \trianglelefteq B$ (let $S = B$) and $A \trianglelefteq B \Rightarrow A \preceq B$ (choose an arbitrary $a \in A$ and "totalize" f by setting $f(b) = a$ for every $b \notin S$). However, such a "totalized" abstraction map f would be semantically unnatural and confusing, a being represented by valid and invalid data at the same time!

It is better to retain the partial nature of f explicit by using the $\trianglelefteq$-notation, $\preceq$ being a shorthand used to denote $\trianglelefteq$-facts such that f is total.

where logical conjunction ($\wedge$) should be regarded as a non-strict connective at its second argument (*e.g.* $FALSE \wedge \perp = FALSE$). For a chain of n $\unlhd$-steps,

$$\unlhd_i \left\{ \begin{array}{l} f_i \\ \phi_i \end{array} \right.$$

the overall abstraction map and invariant are given by:

$$\unlhd \left\{ \begin{array}{l} f \stackrel{\mathrm{def}}{=} \bigcirc_{i=1}^{n} f_i \\ \phi \stackrel{\mathrm{def}}{=} \lambda x.\ \bigwedge_{i=n}^{1} \phi_i((\bigcirc_{j=i+1}^{n} f_j)(x)) \end{array} \right. \tag{12}$$

3.4 A Redundancy Calculus

Readers familiar with model specification such as in VDM are aware that a limited number of set-theoretical constructs is enough for modelling fairly elaborate abstract objects. In $Sets$, such "primitive" constructs are,

- *cartesian product* of two sets A and B (*cf.* records):

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

- *disjoint union* of two sets A and B (*cf.* variant records):

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

- *exponentiation* (A raised to a finite set B, see examples below):

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

On top of these one can build the following *derived* constructs:

- finite subsets of a finite set A (*cf.* set of A in VDM):

  $$2^A$$

- finite binary relations from A to B (*cf.* set of $(A \times B)$ in VDM):

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

- finite partial maps from A to B (*cf.* map A to B in VDM):

  $$A \hookrightarrow B = \bigcup_{K \subseteq A} B^K$$

- finite sequences on a set A (*cf.* seq of A in VDM):

  $$A^\star = \bigcup_{n \geq 0} A^{\overline{n}}$$

  where each exponent $\overline{n}$ denotes the initial segment of $\mathbb{N}$ whose cardinality is n. For simplicity, we will write n instead of $\overline{n}$, and therefore 0 instead of the empty set $\emptyset$.

- union types (*cf.* [A] in VDM):

$$A + 1 \tag{13}$$

where 1 stands for any singleton set, *cf.* [35]. Typically, $1 \cong \{NIL\}$.

A last ("meta") construct is recursive definition,

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

where X is the name of a data sort and $\mathcal{F}$ is a *Sets*-expression ("functor") involving the above primitive or derived constructs. For example, the following recursive VDM syntax for abstract *decision trees*,

$$\begin{array}{rcll} DecTree & :: & Q: & What \qquad /*Question\ or\ Decision\ */ \\ & & R: & Answer \xrightarrow{m} DecTree \qquad /*Subtrees\ */ \\ What & = & ... & \\ Answer & = & ... & \end{array}$$

(where decisions are modelled by $DecTree$ nodes with no answers) is written in the $Sets$ notation as follows:

$$DecTree \cong What \times (Answer \hookrightarrow DecTree) \tag{14}$$

The following theorem is central to the SETS calculus.

Theorem 1 ($\trianglelefteq$-Monotonicity of *Sets*-constructs) *The Sets-constructs $\times$ and $+$ are monotone* wrt. *the $\trianglelefteq$-ordering (8); exponentiation requires isomorphical exponents.*

That is, given Sets-objects A, B, X, Y, M, N such that $A \trianglelefteq X$, $B \trianglelefteq Y$ and $M \cong N$, then facts

$$A \times B \trianglelefteq X \times Y \tag{15}$$

$$A + B \trianglelefteq X + Y \tag{16}$$

$$A^M \trianglelefteq X^N \tag{17}$$

hold.

Outline of Proof: *Let*

$$A \trianglelefteq_f^\phi X, \quad B \trianglelefteq_g^\varphi Y, \quad M \cong_h N \tag{18}$$

Then

1. *Equation (15):*

$$A \times B \trianglelefteq_{f \times g}^{\phi \times \varphi} X \times Y$$

where the product of two functions f and g is "parallel application" [24] [5]

$$(f \times g)\langle a, b\rangle = \langle f(a), g(b)\rangle \tag{19}$$

and the product of two predicates ϕ and φ is "parallel conjunction"

$$(\phi \times \varphi)\langle a, b\rangle = \phi(a) \wedge \varphi(b)$$

[5] For improved readability, functional application involving tupled arguments, *e.g.* $f(\langle x_1, \ldots, x_n\rangle)$, will be abbreviated to $f\langle x_1, \ldots, x_n\rangle$ or $f(x_1, \ldots, x_n)$.

2. *Equation (16):*

$$A + B \trianglelefteq_{f+g}^{\phi+\varphi} X + Y$$

where the sum of two functions f and g is [24]

$$\begin{cases} (f+g)\langle 1, a\rangle & = & \langle 1, f(a)\rangle \\ (f+g)\langle 2, b\rangle & = & \langle 2, g(b)\rangle \end{cases} \tag{20}$$

and the sum of two predicates ϕ and φ is

$$\begin{cases} (\phi+\varphi)\langle 1, a\rangle & = & \phi(a) \\ (\phi+\varphi)\langle 2, b\rangle & = & \varphi(b) \end{cases}$$

3. *Equation (17): Since M and N have the same cardinality, say m, then*

$$\begin{array}{rcl} A^M & \cong & A^m \\ & \cong & \underbrace{A \times \ldots \times A}_{m} \end{array}$$

and

$$\begin{array}{rcl} X^N & \cong & X^m \\ & \cong & \underbrace{X \times \ldots \times X}_{m} \end{array}$$

cf. *(23) below. That is, (17) can be reduced to a finitary version of (15) in which we have*

$$A^m \trianglelefteq_{f^m}^{\phi^m} X^m$$

where

$$\begin{array}{l} f^m\langle a_1, \ldots, a_m\rangle = \langle f(a_1), \ldots, f(a_m)\rangle \\ \phi^m\langle a_1, \ldots, a_m\rangle = \forall 1 \leq i \leq m : \phi(a_i) \end{array} \tag{21}$$

Alternatively, we may define the overall abstraction map t and invariant τ in

$$A^M \trianglelefteq_t^\tau X^N$$

as follows, for $\sigma \in X^N$ and h the isomorphism between M and N (18):

$$\begin{array}{rcl} t(\sigma) & = & f \circ \sigma \circ h^{-1} \\ \tau(\sigma) & = & \forall n \in dom(\sigma) : \phi(\sigma(n)) \end{array}$$

It remains to be proved that all composite abstraction maps above are surjective, which is not hard work, cf. *[22].* □

By this theorem, the components of each data domain of a $Sets$ expression can be refined in isolation, allowing for the stepwise introduction of redundancy in formal models of software.

3.4.1 The $\cong$-Subcalculus

The SETS calculus is fairly rich in algebraic properties. A collection of $\cong$-equalities is given in [35] which establishes that, up to set-theoretical isomorphism, *Sets* may be regarded as a commutative semiring under $\times$ and $+$. Therefore, it makes sense to write finitary products

$$A_1 \times \ldots \times A_n \tag{22}$$

as well as finitary disjoint unions

$$A_1 + \cdots + A_n \quad \textit{that is} \quad \sum_{i=1}^{n} A_i$$

and we have

$$\underbrace{A \times \ldots \times A}_{n} \cong A^n \tag{23}$$

$$\underbrace{A + \ldots + A}_{n} \cong n \times A \tag{24}$$

as expected.

Concerning exponentiation, we recall the following laws from [35]:

$$A^0 \cong 1 \tag{25}$$
$$A^{(B+C)} \cong A^B \times A^C \tag{26}$$
$$A^1 \cong A \tag{27}$$
$$1^A \cong 1 \tag{28}$$
$$(A \times B)^C \cong A^C \times B^C \tag{29}$$
$$C^{A \times B} \cong \left(C^B\right)^A \tag{30}$$
$$A \hookrightarrow B \cong (B+1)^A \tag{31}$$

and note that

$$A^\star \cong \sum_{n \geq 0} A^n \tag{32}$$

$$A \hookrightarrow B \cong \sum_{K \subseteq A} B^K \tag{33}$$

since

$$\begin{array}{rcl} A \neq B & \Rightarrow & X^A \cap X^B = \emptyset \\ A \cap B = \emptyset & \Rightarrow & A \cup B \cong A + B \end{array}$$

Many other basic results are derivable within this subcalculus. The following one will be useful in the sequel:

$$\begin{array}{rcl} 2^A & \cong & (1+1)^A \\ & \cong & A \hookrightarrow 1 \end{array} \tag{34}$$

(= finite sets can be modelled by finite maps), *cf.* $2 \cong 1 + 1$ (24) and (31).

A more interesting example of $\cong$-reasoning is the following: let us re-write (14) avoiding semantically biased symbols such as $DecTree$, $What$, $Answer$ and using more "neutral" symbols such as T, I, A:

$$T \cong I \times (A \hookrightarrow T) \tag{35}$$

Now let $A \cong 2$. Then

$$\begin{aligned} I \times (A \hookrightarrow T) &\cong I \times (2 \hookrightarrow T) \\ &\cong I \times (T + 1)^2 \\ &\cong I \times (T + 1) \times (T + 1) \end{aligned}$$

— *cf.* (31) and (23) — and (35) becomes

$$T \cong I \times (T + 1) \times (T + 1) \tag{36}$$

Let us now compare (36) against the following VDM abstract data model of *genealogical diagrams* (the "pedigree view" of family relationship):

$$\begin{array}{lrll} GenDia :: & I: & Ind & /^{*} \textit{data about an individual} ^{*}/ \quad (37) \\ & F: & [GenDia] & /^{*} \textit{genealogy of his/her father (if known)} ^{*}/ \\ & M: & [GenDia] & /^{*} \textit{genealogy of his/her mother (if known)} ^{*}/ \end{array}$$

Clearly, (36) and (37) are the "same" data model (recall (13)). We conclude that *genealogical diagrams* "are" particular cases of *decision trees*, that is, if a package implementing the latter is available, then it may be *re-used* to implement the former. Of course, a full comparison between both models should examine the algorithmic structure as well [6]. But the point of this example is just to show the usefulness of SETS in comparing or assessing data specification.

3.4.2 The $\trianglelefteq$-Subcalculus

Equations (4) and (10) above record basic results about implementing finite sets. Most other $\trianglelefteq$-results concern the reification of finite partial maps.

Recall laws (25) to (30) concerning exponentiations, that is, sets of total maps. It would be useful if such laws were to hold for partial maps, that is, for expressions of the form $A \hookrightarrow B$ instead of B^A. It can be easily checked that only (26) and (25) are in fact preserved, *cf.* respectively,

$$\begin{aligned} (B + C) \hookrightarrow A &\cong (A + 1)^{B+C} \\ &\cong (A + 1)^B \times (A + 1)^C \\ &\cong (B \hookrightarrow A) \times (C \hookrightarrow A) \end{aligned} \tag{38}$$

and

$$\begin{aligned} 0 \hookrightarrow A &\cong (A + 1)^0 \\ &\cong 1 \end{aligned}$$

[6] For instance, making decisions in $DecTree$ corresponds exactly to ascending genealogical data in $GenDia$, for a fixed *menu* (set of available "answers") at every "decision" level: $2 \cong \{father, mother\}$.

Concerning (27) and (28) we obtain slightly different results: for (28) see (34); for (27) we have

$$\begin{aligned} 1 \hookrightarrow A &\cong (A+1)^1 \\ &\cong A+1 \end{aligned}$$

still in the $\cong$-subcalculus, *cf.* (31) and (27) itself. The remaining two laws lead to $\trianglelefteq$-results, as follows.

(29) "holds" at $\hookrightarrow$ -level provided that $\trianglelefteq$ replaces $\cong$,

$$A \hookrightarrow B \times C \trianglelefteq (A \hookrightarrow B) \times (A \hookrightarrow C) \tag{39}$$

cf. [35]. The $\hookrightarrow$ -version of the "currying" law of exponentiation (30) is another $\trianglelefteq$-result,

$$(A \times B) \hookrightarrow C \trianglelefteq A \hookrightarrow (B \hookrightarrow C) \tag{40}$$

and not the desirable

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

because the expected currying and uncurrying maps are, respectively, not surjective and partial — think of those values in $A \hookrightarrow (B \hookrightarrow C)$ containing the empty map in their range. The invariant induced by (40) is thus

$$\phi(\sigma) \stackrel{\text{def}}{=} \{\ \} \notin rng(\sigma)$$

The law "symmetrical" *wrt.* (40) is therefore not valid, although a somewhat more involved fact holds,

$$A \hookrightarrow (B \hookrightarrow C) \trianglelefteq 2^A \times ((A \times B) \hookrightarrow C) \tag{41}$$

which is a particular case of

$$A \hookrightarrow D \times (B \hookrightarrow C) \trianglelefteq (A \hookrightarrow D) \times ((A \times B) \hookrightarrow C) \tag{42}$$

(let $D = 1$ and apply basic properties of $\times$ and (34)). The invariant induced by (42) is

$$\phi(\sigma, \sigma') \stackrel{\text{def}}{=} \pi_1[dom(\sigma')] \subseteq dom(\sigma) \tag{43}$$

where $\pi_1[dom(\sigma')]$ means $\{\pi_1(p) |\ p \in dom(\sigma')\}$, for π_1, π_2 the standard product selector maps:

$$\begin{aligned} \pi_1(a, b) &= a \\ \pi_2(a, b) &= b \end{aligned}$$

This invariant recommends that all $a \in A$ found in $\sigma \in A \hookrightarrow D$ which cannot be found in $\sigma' \in A \times B \hookrightarrow C$ are exactly the entries, at abstract level, to be mapped to the empty map (in $B \hookrightarrow C$), *cf.* the corresponding abstraction map:

$$f(\sigma, \sigma') \stackrel{\text{def}}{=} \{a \mapsto \langle \sigma(a), \{b \mapsto \sigma'(a, b) |\ \langle a, b\rangle \in proj(a, \sigma')\}\rangle |\ a \in dom(\sigma)\} \tag{44}$$

where

$$proj(a, \sigma') \stackrel{\text{def}}{=} \{\langle a', b\rangle \in dom(\sigma') |\ a' = a\}$$

On the other hand, the $\trianglelefteq$-subcalculus encompasses $\hookrightarrow$ -laws which cannot be adapted from similar results concerning exponentiation. For instance,

$$(B+C)^A \ncong B^A \times C^A$$

and yet we have the following map-decomposition law:

$$A \hookrightarrow (B+C) \trianglelefteq_f^\phi (A \hookrightarrow B) \times (A \hookrightarrow C) \quad (45)$$

for

$$\begin{aligned} \phi(\sigma,\rho) &\stackrel{\text{def}}{=} dom(\sigma) \cap dom(\rho) = \emptyset \\ f(\sigma,\rho) &\stackrel{\text{def}}{=} i_1 \circ \sigma \cup i_2 \circ \rho \end{aligned}$$

where i_1, i_2 are the standard co-product injection maps:

$$\begin{aligned} i_1(a) &= \langle 1, a \rangle \\ i_2(a) &= \langle 2, a \rangle \end{aligned}$$

Laws such as (39,41,42) and (45) play an important rôle in finite map reification. They can be used to "decompose" complex/nested maps into tuples of simpler maps. Recalling from [35] how immediate it is to refine finite maps into binary relations,

$$A \hookrightarrow B \trianglelefteq_{mkf}^{fdp} 2^{A \times B} \quad (46)$$

for [7]

$$fdp(R) \stackrel{\text{def}}{=} \forall \langle a,b \rangle, \langle a',b' \rangle \in R : a = a' \Rightarrow b = b' \quad (47)$$

$$mkf(R) \stackrel{\text{def}}{=} \{ a \mapsto the(\{x \in B |\ a\ R\ x\}) |\ \langle a,b \rangle \in R \} \quad (48)$$

it makes sense to say that the $\trianglelefteq$-subcalculus fragment studied above is just what we need in order to refine elaborate, finite-map based data models into relational database schemata. This will become apparent from the illustration which will be given in section 5. But we need further, more ambitious $\trianglelefteq$-results concerning recursion removal.

4 Dealing with Recursive Data Models

Equations (14) and (36) presented above are examples of recursively defined data models in *Sets*. Many others could have been presented since recursion

[7] The partial operator

$$the : 2^A \to A$$

yields the unique element of a singleton set, that is,

$$the(\{a\}) = a$$

while $the(\{a,b\})$, $the(\emptyset)$ are undefined expressions.

normally provides "elegant" solutions for a wide range of problems, namely:

$$X \cong 1 + A \times X \quad /\text{*}\mathit{finite\ lists\ on}\ A\ \text{*}/ \tag{49}$$
$$X \cong 1 + A \times X^2 \quad /\text{*}\mathit{binary\ trees\ on}\ A\ \text{*}/ \tag{50}$$
$$X \cong 1 + A \times X^\star \quad /\text{*}\mathit{generic\ trees\ on}\ A\ \text{*}/$$
$$X \cong V + A \times X^\star \quad /\text{*}\mathit{formal\ terms\ on}\ V\ \mathit{and}\ A\ \text{*}/$$

etc.

What can we do about refining a recursive data model?

Some languages support recursion directly (*e.g.* LISP) but may others do not (*e.g.* FORTRAN). Languages such as PASCAL are "half-way through" — recursion is supported in an indirect way by providing *pointers* to *dynamic storage* ("heaps"). Programming with pointers is error-prone (*cf.* pointer undefinedness, non-termination caused by cyclic referencing *etc.*) and tends to produce "tricky" code (pointers are data-structuring counterparts of "gotos" [46]). A way of overcoming this situation is to think of a generic rule for refining recursive data structures into safe non-recursive ones, thus ruling out non-systematic use of pointers [8].

By way of motivation, let us recall Fielding's VDM *representation 1* of abstract mappings [10],

$$\begin{array}{l} S1 = [Bt1] \\ Bt1 :: S1\ Key\ Data\ S1 \end{array} \tag{51}$$

which means nothing more than recursive definition (50) in *Sets* (let $A \cong Key \times Data$). *Representation 2* presented in the same work [10] "maps a binary tree onto linear storage":

$$\begin{array}{l} S2 :: ROOT : [Ptr]\ ARRAY : Ptr \xrightarrow{m} Node2 \\ Node2 :: [Ptr]\ Key\ Data\ [Ptr] \end{array} \tag{52}$$

The abstract schema of (52) in *Sets* is

$$\begin{array}{rcl} S2 & \cong & (K+1) \times ARRAY \\ ARRAY & \cong & K \hookrightarrow Key \times Data \times (K+1)^2 \end{array}$$

for an abstract domain $K \cong \mathbb{N}$ of pointers, *cf.* Ptr in (52).

From [10] we know that (52) refines (51) under a fairly elaborate datatype invariant ensuring pointer definedness and absence of pointer loops. A similar invariant is required to justify the following well-known "linked-list" representation of finite lists (49):

$$X_1 \cong (K+1) \times (K \hookrightarrow A \times (K+1))$$

Can these two "recursion removing" reification steps be generalized?

We may conjecture that, if X_1 is a fixpoint solution of a recursive definition in *Sets* of the form

$$X \cong 1 + \mathcal{G}(X), \tag{53}$$

[8] Much in the same way that one can do "Structured Programming with Goto Statements" *cf.* [23].

then

$$X_1 \trianglelefteq_f^\phi (K+1) \times (K \hookrightarrow \mathcal{G}(K+1)) \tag{54}$$

for $K \cong \mathbb{N}$. For this conjecture to hold we need to find which f, ϕ are generically induced by $\mathcal{G}$. We will show below that an elegant way of doing it is to regard $\mathcal{G}$ as a polynomial (endo)functor in $Sets$ [24].

Some necessary concepts about functors in $Sets$ will be given next, adapted from [24] where the same concepts can be found under the appropriate categorial generalization.

4.1 Polynomial Functors

Let A, B, C be objects in $Sets$. Let $f : A \to B$ be a morphism in $Sets$ and let $1_C : C \to C$ denote the identity morphism on C. Then C may be regarded as the *constant functor*

$$C : Sets \longrightarrow Sets \tag{55}$$

such that $C(A) = C$ and $C(f) = 1_C$.

Let $\mathcal{F}, \mathcal{G} : Sets \longrightarrow Sets$ be functors. Define their *product*

$$\mathcal{F} \times \mathcal{G} : Sets \longrightarrow Sets$$

by $(\mathcal{F} \times \mathcal{G})(A) = \mathcal{F}(A) \times \mathcal{G}(A)$ and by $(\mathcal{F} \times \mathcal{G})(f) = \mathcal{F}(f) \times \mathcal{G}(f)$, *cf.* (19). For instance, $\mathcal{F}(X) = X \times C$ defines the product functor of the identity functor $\lambda X.X$ and the constant functor C (55), and we have $\mathcal{F}(f)\langle a, c\rangle = (f \times 1_C)\langle a, c\rangle = \langle f(a), c\rangle$.

We may have n-ary products of functors, of which $\mathcal{F}(X) = X^n$ is a particular case. For f the same morphism as above, we will have $\mathcal{F}(f) = f^n : A^n \to B^n$, *cf.* (21).

The notion of a product functor has a dual — the *co-product functor* (or sum of two functors) — which resorts to (20):

$$\begin{array}{rcl} \mathcal{F} + \mathcal{G} & : & Sets \longrightarrow Sets \\ (\mathcal{F} + \mathcal{G})(A) & = & \mathcal{F}(A) + \mathcal{G}(A) \\ (\mathcal{F} + \mathcal{G})(f) & = & \mathcal{F}(f) + \mathcal{G}(f) \end{array}$$

Co-product functors may also be iterated to n-arguments.

Finally, we are ready to define what a *polynomial functor* is — every functor $\mathcal{F} : Sets \longrightarrow Sets$ which is either a constant functor or the identity functor, or the composition or (finitary) product or sum of other polynomial functors, is said to be *polynomial*.

For instance,

$$\mathcal{F}(X) = 1 + A \times X \tag{56}$$

(*cf.* (49)) is a polynomial functor and so is $\mathcal{F}(X) = X^\star$, *cf.* (32). $\mathcal{F}(X) = C \hookrightarrow X$ is also polynomial for C a finite exponent (*cf.* $C \hookrightarrow X \cong (X+1)^C$ and [24]) and we have

$$\begin{array}{rcl} (C \hookrightarrow f)(\sigma) & = & f \circ \sigma \\ & = & \{c \mapsto f(\sigma(c)) | \; c \in dom(\sigma)\} \end{array} \tag{57}$$

The following result is valid in *Sets* [24]: every polynomial functor

$$\mathcal{F} : Sets \longrightarrow Sets$$

may be put into the canonical form:

$$\begin{array}{rcl} \mathcal{F}(X) & \cong & C_0 + (C_1 \times X) + (C_2 \times X^2) + \cdots + (C_n \times X^n) \\ & = & \sum_{i=0}^{n} C_i \times X^i \end{array} \tag{58}$$

For instance, for $\mathcal{F}(X) = X^\star$ we have every $C_i = 1$, since

$$\begin{array}{rcl} X^\star & \cong & \displaystyle\sum_{i \geq 0} X^i \\ & \cong & \displaystyle\sum_{i \geq 0} 1 \times X^i \end{array}$$

(recall (32) and the fact that 1 is the identity of $\times$ [9]).

A suggestive result for converting polynomial functors into canonical form is *Newton's binomial formula* itself,

$$(A + B)^n \cong \sum_{p=0}^{n} {}^nC_p \times A^{n-p} \times B^p \tag{59}$$

as illustrated by the following treatment of the "pedigree" functor

$$\mathcal{F}(T) \quad = \quad I \times (T + 1)^2$$

of section 3.4.1. We will have,

$$\begin{array}{rcl} \mathcal{F}(T) & = & I \times (T + 1)^2 \\ & \cong & I \times (T^2 + 2 \times T + 1) \\ & \cong & I \times T^2 + 2 \times I \times T + I \end{array}$$

which literally means: *"about an individual I either both father and mother are known (T^2), or one of either father or mother are known ($2 \times T$), or both father and mother are unknown (1)"*.

Combining (58,59) with the map-decomposition law (45), we obtain a general rule for decomposing a mapping of the form

$$A \hookrightarrow \mathcal{F}(X)$$

into a cartesian product of mappings, *i.e.*

$$\Pi_{i=0}^{n}(A \hookrightarrow C_i \times X^i) \tag{60}$$

each of which may in turn be refined into tabular form, *cf.* relational data-base design.

Finally, with the aid of polynomial functors we may extend theorem 1 as follows:

[9] We regard a denumerable sum of polynomial functors as being a polynomial functor.

Table 1: Summary of Functorial Calculus

$\mathcal{F}(X)$	$\mathcal{F}(f)$	$\mathcal{F}(\phi)$
X	f	ϕ
C	1_C	$\lambda c.TRUE$
2^X	$2^f = \lambda s.\{f(x) \mid x \in s\}$	$2^\phi = \lambda s.\forall b \in s : \phi(b)$
$X^\star$	$f^\star = \lambda l.<f(x) \mid x \leftarrow l>$	$\phi^\star = \lambda l.\forall 1 \leq i \leq length(l) : \phi(l(i))$
X^C	$f^C = \lambda\sigma.f \circ \sigma$	$\phi^C = \lambda\sigma.\forall c \in C : \phi(\sigma(c))$
$C \hookrightarrow X$	$C \hookrightarrow f = \lambda\sigma.f \circ \sigma$	$C \hookrightarrow \phi = \lambda\sigma.\forall x \in rng(\sigma) : \phi(x)$
$\mathcal{G}(X) \times \mathcal{H}(X)$	$\mathcal{G}(f) \times \mathcal{H}(f)$	$\lambda(x,y).\mathcal{G}(\phi)(x) \wedge \mathcal{H}(\phi)(y)$
$\mathcal{G}(X) + \mathcal{H}(X)$	$\mathcal{G}(f) + \mathcal{H}(f)$	$\lambda x.\begin{cases} x = \langle 1, a\rangle & \Rightarrow \quad \mathcal{G}(\phi)(a) \\ x = \langle 2, b\rangle & \Rightarrow \quad \mathcal{H}(\phi)(b) \end{cases}$
$\mathcal{G}(X) + 1$	$\lambda x.\begin{cases} x = \langle 2, NIL\rangle & \Rightarrow \quad x \\ x = \langle 1, a\rangle & \Rightarrow \quad \langle 1, \mathcal{G}(f)(a)\rangle \end{cases}$	$\lambda x.\begin{cases} x = \langle 2, NIL\rangle & \Rightarrow \quad TRUE \\ x = \langle 1, a\rangle & \Rightarrow \quad \mathcal{G}(\phi)(a) \end{cases}$

Theorem 2 *Let $\mathcal{F}$ be a polynomial (endo)functor in $Sets$. If*

$$A \trianglelefteq_f^\phi B \tag{61}$$

then

$$\mathcal{F}(A) \trianglelefteq_{\mathcal{F}(f)}^{\mathcal{F}(\phi)} \mathcal{F}(B) \tag{62}$$

Proof: *It is performed by induction on the structure of polynomial functor $\mathcal{F}$. Sum and product (and exponentiation) have been dealt with in theorem 1.*

The base case concerning the identity functor $\mathcal{F}(X) = X$ is trivial, since $\mathcal{F}(f) = f$ and $\mathcal{F}(\phi) = \phi$, i.e. *(62) reduces to (61). The other base case concerns the constant functor $\mathcal{F}(X) = C$. We have $\mathcal{F}(f) = 1_C$ and $\mathcal{F}(\phi) = \lambda b.TRUE$, (62) holding trivially.* $\square$

This theorem provides an elegant, functorial strategy for computing complex abstraction invariants throughout a reification process. Although $\mathcal{F}(X) = 2^X$ is not polynomial [24], (62) still holds for this functor, for finite A and B [22]. Table 1 presents a summary of this "functorial calculus" for the most common $Sets$ constructs.

4.2 A Result for Recursion Removal

We now come back to our conjecture about (54). The result which will be presented below is applicable to recursive definitions of the form

$$X \cong \mathcal{F}(X) \tag{63}$$

of which (53) is a particular case, let $\mathcal{F}(X) = 1 + \mathcal{G}(X)$.

Let $(X_0, \delta : \mathcal{F}(X_0) \to X_0)$ be a fixpoint of functor $\mathcal{F}$ (63) in $Sets$. A least fixpoint solution to (63) is guaranteed for every *co-continuous* functor $\mathcal{F}$. Moreover, every polynomial functor is co-continuous [24].

We want to discuss the following reification step:

$$X_0 \trianglelefteq_f^\phi K \times (K \hookrightarrow \mathcal{F}(K)) \tag{64}$$

for K a domain of "pointers" such that $K \cong \mathbb{N}$. (64) will hold provided that we define a surjection

$$f : K \times (K \hookrightarrow \mathcal{F}(K)) \longrightarrow X_0$$

which is total over $\{\langle k, \sigma\rangle \in K \times (K \hookrightarrow \mathcal{F}(K)) |\ \phi(k, \sigma)\}$. We start by temporarily assuming that $\sigma \in K \hookrightarrow \mathcal{F}(K)$ is a total function, and draw the following diagram

$$\begin{array}{c} K \\ \downarrow \sigma \\ \mathcal{F}(K) \end{array} \tag{65}$$

Assuming a given piece of "linear storage" σ ("database"), let f_σ denote the function which, for each "pointer" $k \in K$, retrieves the value of X_0 corresponding to a σ "scan" starting from k. We may add f_σ to (65), that is

$$\begin{array}{ccc} X_0 & \xleftarrow{f_\sigma} & K \\ & & \downarrow \sigma \\ & & \mathcal{F}(K) \end{array}$$

Since (X_0, δ) is a fixpoint of $\mathcal{F}$, we may add δ to the above diagram, and obtain

$$\begin{array}{ccc} X_0 & \xleftarrow{f_\sigma} & K \\ \delta \uparrow & & \downarrow \sigma \\ \mathcal{F}(X_0) & & \mathcal{F}(K) \end{array}$$

Finally, our diagram may be "closed" by $\mathcal{F}(f_\sigma)$,

$$\begin{array}{ccc} X_0 & \xleftarrow{f_\sigma} & K \\ \delta \uparrow & & \downarrow \sigma \\ \mathcal{F}(X_0) & \xleftarrow{\mathcal{F}(f_\sigma)} & \mathcal{F}(K) \end{array} \tag{66}$$

since $\mathcal{F}$ ia a functor.

The equation implicit in commutative diagram (66) is

$$f_\sigma(k) = \delta(\mathcal{F}(f_\sigma)(\sigma(k)))$$

We may write $f(k, \sigma)$ instead of $f_\sigma(k)$, obtaining the following abstraction map for (64)

$$\begin{array}{rcl} f & : & K \times (K \hookrightarrow \mathcal{F}(K)) \longrightarrow X_0 \\ f(k, \sigma) & \stackrel{\text{def}}{=} & \delta(\mathcal{F}(f_\sigma)(\sigma(k))) \end{array} \tag{67}$$

Let us see an example involving polynomial functor (56). $(A^\star, \delta)$ is a well-known fixpoint solution of this functor [24, 35] for

$$\begin{array}{rcl} \delta & : & 1 + A \times A^\star \longrightarrow A^\star \\ \delta(x) & \stackrel{\text{def}}{=} & \left\{ \begin{array}{lcl} <> & \Leftarrow & x = \langle 1, NIL\rangle \\ cons(a, l) & \Leftarrow & x = \langle 2, \langle a, l\rangle\rangle \end{array} \right. \end{array} \tag{68}$$

For $\mathcal{F}(f_\sigma)$ we obtain

$$\begin{aligned}
\mathcal{F}(f_\sigma)(x) &= (1 + A \times f_\sigma)(x) \\
&= (1_1 + 1_A \times f_\sigma)(x) \\
&= \left\{ \begin{array}{lcl} x & \Leftarrow & x = \langle 1, NIL \rangle \\ \langle 2, \langle a, f_\sigma(l) \rangle \rangle & \Leftarrow & x = \langle 2, \langle a, l \rangle \rangle \end{array} \right.
\end{aligned} \tag{69}$$

Composing (69) with (68) we obtain, following (67),

$$f(k,\sigma) \stackrel{\text{def}}{=} \left\{ \begin{array}{lcl} <> & \Leftarrow & \sigma(k) = \langle 1, NIL \rangle \\ cons(a, f_\sigma(k')) & \Leftarrow & \sigma(k) = \langle 2, \langle a, k' \rangle \rangle \end{array} \right.$$

which is written in a more "palatable" notation, as follows:

$$\begin{array}{lcl}
f(k,\sigma) & \stackrel{\text{def}}{=} & if\ is\text{-}NIL(\sigma(k)) \\
& & then\ <> \\
& & else\ \ let\ \ \ \sigma(k) = \langle a, k' \rangle \\
& & \qquad\ \ in\ \ \ cons(a, f(k', \sigma))
\end{array}$$

Removing the assumption that σ is a total function, we have to face pointer undefinedness — $\sigma(k)$ in (67) is undefined wherever $k \notin dom(\sigma)$ and other pointers k' in the range of σ, reachable from k, may be in the same situation.

Pointer reachability can be characterized by the transitive closure $<^+_\mathcal{F}$ of the following ordering on K, induced by $\mathcal{F}$ and σ:

$$k_1 <_\mathcal{F} k \quad \stackrel{\text{def}}{=} \quad k \in dom(\sigma) \wedge k_1 \in_\mathcal{F} \sigma(k) \tag{70}$$

where logical conjunction is once again regarded as non-strict (*cf.* (11)), and $\in_\mathcal{F}$ is structurally defined, for polynomial $\mathcal{F}$, as follows:

$$\begin{aligned}
k \in_C x &\stackrel{\text{def}}{=} FALSE \\
k \in_{\lambda X.X} x &\stackrel{\text{def}}{=} k = x \\
k \in_{\mathcal{F} \times \mathcal{G}} \langle x, y \rangle &\stackrel{\text{def}}{=} k \in_\mathcal{F} x \vee k \in_\mathcal{G} y \\
k \in_{\mathcal{F} + \mathcal{G}} x &\stackrel{\text{def}}{=} \left\{ \begin{array}{lcl} k \in_\mathcal{F} y & \Leftarrow & x = \langle 1, y \rangle \\ k \in_\mathcal{G} z & \Leftarrow & x = \langle 2, z \rangle \end{array} \right.
\end{aligned} \tag{71}$$

The following invariant for (64) prevents reachable pointers being undefined:

$$\begin{array}{lcll}
\phi(k,\sigma) & \stackrel{\text{def}}{=} & let & P = \{k\} \cup \{k' \in K \mid k' <^+_\mathcal{F} k\} \\
& & in & P \subseteq dom(\sigma)
\end{array}$$

However, this invariant is insufficient if we want to restrict our interpretation of recursion to least fixpoints [24], that is, if we want to guarantee that $f(k,\sigma)$ does not yield infinite results. It remains to enforce that P is a well-founded set *wrt.* $<_\mathcal{F}$:

$$\begin{array}{lcll}
\phi(k,\sigma) & \stackrel{\text{def}}{=} & let & P = \{k\} \cup \{k' \in K \mid k' <^+_\mathcal{F} k\} \\
& & in & P \subseteq dom(\sigma) \wedge \\
& & & \forall \emptyset \subset C \subseteq P : \exists m \in C : \forall k' <_\mathcal{F} m : k' \notin C
\end{array} \tag{72}$$

For instance, let $\mathcal{F}(X) = C \hookrightarrow \mathcal{G}(X)$. It is not hard to obtain $\in_{\mathcal{F}}$ for this case,

$$k \in_{C \hookrightarrow \mathcal{G}} x \stackrel{\text{def}}{=} \exists c \in dom(x) : k \in_{\mathcal{G}} x(c) \tag{73}$$

Condition (73) will be helpful in understanding the illustration of calculated reification which will be presented in section 5.

The proofs about the surjectiveness of f (67), for $(X_0, \delta : \mathcal{F}(X_0) \to X_0)$ a finite or denumerable fixpoint of $\mathcal{F}$, and its termination over ϕ (72), are too lengthy to be pursued here, and are available from [22]. The first of these proofs is inductively performed over the structure of polynomial functor $\mathcal{F}$, and requires $K \cong \mathbb{N}$.

It is interesting to analyze the meaning of (64) for a few simple, particular cases of $\mathcal{F}$:

– Constant functor $\mathcal{F}(X) = C$. $(C, 1_C)$ is of course a fixpoint of $\mathcal{F}$. From (67) and (55) we obtain abstraction map

$$\begin{array}{rcl} f & : & K \times (K \hookrightarrow C) \longrightarrow C \\ f(k, \sigma) & \stackrel{\text{def}}{=} & \delta(\mathcal{F}(f_\sigma)(\sigma(k))) \\ & = & 1_C(C(f_\sigma)(\sigma(k))) \\ & = & 1_C(1_C(\sigma(k))) \\ & = & \sigma(k) \end{array}$$

From (70) and (71) we obtain an empty ordering $<^+_{\mathcal{F}}$ on pointers (K), whereby invariant (72) reduces to

$$\phi(k, \sigma) \stackrel{\text{def}}{=} k \in dom(\sigma) \tag{74}$$

as expected. This particular case corresponds to a popular programming technique (typical of C or PASCAL): instead of handling C data directly (statically stored), the program handles *dynamic* references to them.

– Same as above, for $C = 0$. This case trivializes (63) to the definition $X \cong 0$ of an empty data domain; note that $K \times (K \hookrightarrow \mathcal{F}(K))$ becomes $K \times (K \hookrightarrow 0)$ which reduces to

$$\begin{array}{rcl} K \times (K \hookrightarrow 0) & \cong & K \times \bigcup_{X \subseteq K} 0^X \\ & \cong & K \times 0^0 \end{array}$$

That is, $\sigma : 0 \to 0$ is bound to be completely undefined, and $dom(\sigma) = \emptyset$; then $k \in dom(\sigma) = FALSE$ and (74) reduces to

$$\phi(k, \sigma) \stackrel{\text{def}}{=} FALSE$$

As expected, we are led to the empty reification (every datum is invalid).

– Identity functor $\mathcal{F}(X) = X$. This case trivializes (63) to $X \cong X$, which accepts any fixpoint $(X_0, \delta : X_0 \to X_0)$ for δ a bijection. Let us see how the least fixpoint interpretation implicit in (72) reduces this case to the previous one ($X_0 = 0$), by showing that definedness and well-foundedness cannot be met at the same time.

Assuming that P is well-founded *wrt.* $k_1 <_{\lambda X.X} k$ (which we abbreviate to $k_1 < k$ and is logically equivalent to $k \in dom(\sigma) \wedge k_1 = \sigma(k)$), we have

$$\exists m \in P : \sigma(m) \notin P \tag{75}$$

But if $m \in P$ then m is reachable from k ($m <^+ k$), $m \in dom(\sigma)$ and $\sigma(m) < m$. Then

$$\sigma(m) < m <^+ k$$

i.e. $\sigma(m) <^+ k$, which entails $\sigma(m) \in P$ and contradicts (75). Thus the conjunction of definedness and well-foundedness in (74) is impossible, and we obtain

$$\phi(k, \sigma) \stackrel{\text{def}}{=} FALSE$$

Equation (72) is a generalization of some invariant definitions found in the VDM literature, *cf. e.g.* [10, 47, 48]. It also is less constrained in the sense that only the set P of pointers in σ reachable from k is affected by non-circularity and definedness restrictions ([10, 48] constrain $dom(\sigma)$ as a whole).

Some algorithmic flavour can be added to (72) by introducing an auxiliary function

$$reach(k, \sigma) \stackrel{\text{def}}{=} \{k\} \cup \{k' \in K | \; k' <^+_{\mathcal{F}} k\}$$

(that is, $P = reach(k, \sigma)$) subject to transformations as follows:

$$\begin{array}{rl} & reach(k,\sigma) \stackrel{\text{def}}{=} \\ & \{k\} \cup \{k' \in K | \; k' <^+_{\mathcal{F}} k\} \\ = & \{k\} \cup \left\{ \begin{array}{lcl} \emptyset & \Leftarrow & k \notin dom(\sigma) \\ \bigcup_{k' <_{\mathcal{F}} k} \{k'\} \cup \{k'' \in K | \; k'' <^+_{\mathcal{F}} k'\} & \Leftarrow & k \in dom(\sigma) \end{array} \right. \\ = & \{k\} \cup \left\{ \begin{array}{lcl} \emptyset & \Leftarrow & k \notin dom(\sigma) \\ \bigcup_{k' <_{\mathcal{F}} k} reach(k', \sigma) & \Leftarrow & k \in dom(\sigma) \end{array} \right. \end{array}$$

The test for definedness may be encapsulated in a predicate

$$defined(k, \sigma) \stackrel{\text{def}}{=} P \subseteq dom(\sigma)$$

Since $P = reach(k, \sigma)$ we obtain, after the expected substitutions and transformations:

$$defined(k, \sigma) \stackrel{\text{def}}{=} \left\{ \begin{array}{lcl} FALSE & \Leftarrow & k \notin dom(\sigma) \\ \forall k' <_{\mathcal{F}} k : defined(k', \sigma) & \Leftarrow & k \in dom(\sigma) \end{array} \right.$$

Note that both *reach* and *defined* do not compute the intended set-theoretical fixpoint implicit in the transitive closure of $<_{\mathcal{F}}$ wherever this relation is cyclic. A $<_{\mathcal{F}}$-cycle is detected wherever we revisit the same $k \in K$. Since cycle detection matches with wellfoundedness testing, one may put everything together and write

$$\phi(k, \sigma) = \phi_{aux}(k, \sigma, \emptyset) \tag{76}$$

where

$$\phi_{aux}(k, \sigma, C) \stackrel{\text{def}}{=} \left\{ \begin{array}{lcl} FALSE & \Leftarrow & k \notin dom(\sigma) \vee k \in C \\ \forall k' <_{\mathcal{F}} k : \phi_{aux}(k', \sigma, C \cup \{k\}) & \Leftarrow & k \in dom(\sigma) \end{array} \right. \tag{77}$$

5 An Illustration of Calculated Reification

The *Sets*-model $DecTree$ for decision trees presented above (14) will be explored in this section and subject to a series of calculations, illustrating the main results of this paper. Refinement steps will be indexed by natural numbers to make it easy to apply rule (12) for inferring abstraction maps and data-type invariants.

This exercise about $DecTree$ is a simplified version of a similar exercise reported in [40], a project where legal knowledge about public property records, expressed in terms of decision trees, had to be incorporated in a pre-existing relational database system.

5.1 Data-level Reification

Recursion removal according to (64) is the first transformation applicable to (14). One obtains

$$DecTree \quad \trianglelefteq_1 \quad DecTree_1$$

where

$$DecTree_1 \quad = \quad K \times (K \hookrightarrow What \times (Answer \hookrightarrow K)) \tag{78}$$

Before proceeding, let us think a little about what has been achieved in $DecTree_1$ (78). Two interpretations (at least) are admissible:

1. For K a domain of *pointers*,

   $$K \hookrightarrow What \times (Answer \hookrightarrow K) \tag{79}$$

   models a heap-segment of dynamic storage. In PASCAL (where the heap is "hidden" in the run-time system) we would write something like [10]

```
type  DecTree1 =  ^DecTree;
      DecTree =   record
                  Q: What;                                  (80)
                  R: array [Answer] of ^DecTree
                  end;
```

[10] Strictly speaking, (80) is a further refinement of (79) because PASCAL pointers implement $K+1$ rather than K (the 1 alternative corresponds to the `nil` value). Therefore, an invariant over (80) will be required ruling the `nil` alternative out of `DecTree1`.

2. For K a domain of *object names*, (79) models the object database

$$name \rightarrow object$$

implicit in an object-oriented programming environment [47]. This is better suggested by the following VDM "sugaring" of (78):

$$\begin{array}{rcrl} DecTree_1 & :: & ObjName: & K \\ & & Archive: & ObjBase \\ ObjBase & = & \multicolumn{2}{l}{K \stackrel{m}{\rightarrow} Attributes} \\ Attributes & :: & Q: & What \\ & & SubObjs: & Answer \stackrel{m}{\rightarrow} K \end{array}$$

Let us proceed via law (42):

$$\begin{array}{rcl} DecTree_1 & = & K \times (K \hookrightarrow What \times (Answer \hookrightarrow K)) \\ & \trianglelefteq_2 & K \times ((K \hookrightarrow What) \times ((K \times Answer) \hookrightarrow K)) \\ & = & DecTree_2 \end{array} \qquad (81)$$

For K a state descriptor, $DecTree_2$ accepts the following interpretation:

- $(K \times Answer) \hookrightarrow K$ = *state transition diagram* of a (deterministic) *finite state automaton* ($Answer$ = input stimuli);
- the first factor $k \in K$ in (81) = current state of the automaton;
- $K \hookrightarrow What$ = semantic table assigning a meaning to each state.

Step 2 led us to a lower level where we met "good old friends": finite state automata can be implemented using *arrays* and *jumps*! Carrying on our reasoning,

$$\begin{array}{rcl} DecTree_2 & = & K \times ((K \hookrightarrow What) \times ((K \times Answer) \hookrightarrow K)) \\ & \trianglelefteq_3 & K \times (2^{K \times What} \times 2^{(K \times Answer) \times K}) \\ & = & DecTree_3 \\ & \cong_4 & K \times (2^{K \times What} \times 2^{K \times Answer \times K}) \\ & = & DecTree_4 \end{array}$$

cf. (46) and (22). Our final model,

$$DecTree_4 = K \times (2^{K \times What} \times 2^{K \times Answer \times K}) \qquad (82)$$

is nothing but a relational database schema for implementing $DecTree$ in terms of two database files (tables) where K plays the rôle of a domain of *keys*.

In summary, our reasoning can be sketched by

$$DecTree \trianglelefteq_1 DecTree_1 \trianglelefteq_2 DecTree_2 \trianglelefteq_3 DecTree_3 \cong_4 DecTree_4 \qquad (83)$$

5.2 Inference of the Abstraction Map and Data-type Invariant

The overall abstraction map of (83) can be inferred via (12), *i.e.*

$$f(k, \langle t, t' \rangle) = f_1(f_2(f_3(f_4(k, \langle t, t' \rangle))))$$

where f is obtained by functorial composition of abstraction maps f_1 to f_4, as follows: f_4 is given by

$$f_4 = 1_K \times (1_{2^{K \times What}} \times 2^f)$$

where

$$(A \times B) \times C \cong_f A \times B \times C$$

that is,

$$f_4(\langle k, \langle t, t' \rangle \rangle) \quad = \quad \langle k, \langle t, \{ \langle \langle k, a \rangle, k' \rangle | \ \langle k, a, k' \rangle \in t' \} \rangle \rangle$$

In a similar way, f_3 resorts to the mkf abstraction map (48) between relations and functions,

$$f_3 \quad = \quad 1_K \times (mkf \times mkf)$$

and f_2 is simply

$$f_2 = 1_K \times f$$

where f is given by (44). Finally,

$$\begin{array}{rcll} f_1(k, \sigma) & = & f_\sigma(k) & \\ f_\sigma(k) & \stackrel{\mathrm{def}}{=} & let & \sigma(k) = \langle w, \sigma' \rangle \\ & & in & \langle w, \{ a \mapsto f_\sigma(\sigma'(a)) | \ a \in dom(\sigma') \} \rangle \end{array} \tag{84}$$

cf. (67,19) and (57). Then

$$\begin{array}{rcl} f_3(f_4(\langle k, \langle t, t' \rangle \rangle)) & = & \langle k, \langle mkf(t), mkf(\{ \langle \langle k, a \rangle, k' \rangle | \ \langle k, a, k' \rangle \in t' \}) \rangle \rangle \\ & = & \langle k, \langle \{ k \mapsto w | \ \langle k, w \rangle \in t \}, \{ \langle k, a \rangle \mapsto k' | \ \langle k, a, k' \rangle \in t' \} \rangle \rangle \end{array}$$

and

$$\begin{array}{l} f_2(f_3(f_4(\langle k, \langle t, t' \rangle \rangle))) = \\ \qquad \langle k, \{ x \mapsto \langle w, \{ a \mapsto k' | \ \langle k'', a, k' \rangle \in t' \wedge k'' = x \} \rangle | \ \langle x, w \rangle \in t \} \rangle \end{array} \tag{85}$$

Combining (85) with (84) we obtain, after some simplifications

$$\begin{array}{rll} f(k, \langle t, t' \rangle) \stackrel{\mathrm{def}}{=} & let & w = the(\{ \pi_2(r) | \ r \in t \wedge \pi_1(r) = k \}) \\ & in & \langle w, \{ a \mapsto f(k', \langle t, t' \rangle) | \ \langle k'', a, k' \rangle \in t' \wedge k'' = k \} \rangle \end{array} \tag{86}$$

The overall data-type invariant can be calculated in similar way,

$$\phi(k, \langle t, t' \rangle) = \phi_3(f_4(k, \langle t, t' \rangle)) \wedge \phi_2(f_3(f_4(k, \langle t, t' \rangle))) \wedge \phi_1(f_2(f_3(f_4(k, \langle t, t' \rangle))))$$

since ϕ_4 is universally true. ϕ_3 resorts to fdp (47),

$$\phi_3 = (\lambda k.TRUE) \times fdp \times fdp$$

and ϕ_2 is given by (43). It may be checked that $\phi_2(f_3(f_4(k, \langle t, t' \rangle)))$ reduces to $\pi_1[t'] \subseteq \pi_1[t]$. Then

$$\begin{array}{rcl} \phi_3(f_4(k, \langle t, t' \rangle)) & = & TRUE \wedge fdp(t) \wedge fdp(\{\langle \langle k, a \rangle, k' \rangle | \ \langle k, a, k' \rangle \in t' \}) \\ & = & fdp(t) \wedge fdp(\{\langle \langle k, a \rangle, k' \rangle | \ \langle k, a, k' \rangle \in t' \}) \end{array} \tag{87}$$

Further calculating $fdp(\{\langle \langle k, a \rangle, k' \rangle | \ \langle k, a, k' \rangle \in t' \})$ in (87) yields

$$\forall \langle k_1, a, k_1' \rangle, \langle k_2, b, k_2' \rangle \in t' : (k_1 = k_2 \wedge a = b) \Rightarrow k_1' = k_2'$$

Finally, ϕ_1 (72) is based on the ordering

$$\begin{array}{rcl} k_1 < k & \stackrel{\mathrm{def}}{=} & k \in dom(\sigma) \wedge \\ & & (FALSE \vee \exists a \in dom(\pi_2(\sigma(k))) : k_1 = \pi_2(\sigma(k))(a)) \\ & \Leftrightarrow & k \in dom(\sigma) \wedge \ \begin{array}{ll} let & \sigma' = \pi_2(\sigma(k)) \\ in & \exists a \in dom(\sigma') : k_1 = \sigma'(a) \end{array} \end{array}$$

cf. (73). Recalling (76,77) above, we may reduce $\phi_1(f_2(f_3(f_4(k, \langle t, t' \rangle))))$ to $aux(k, t, t', \emptyset)$ where

$$aux(k, t, t', C) \stackrel{\mathrm{def}}{=} \left\{ \begin{array}{lcl} FALSE & \Leftarrow & k \notin \pi_1[t] \vee \\ & & k \in C \\ \forall r \in \{\pi_1(s) = k | \ s \in t'\} : aux(\pi_3(r), t, t', C \cup \{k\}) & \Leftarrow & k \in \pi_1[t] \end{array} \right. \tag{88}$$

Putting everything together, we finally obtain the following (non-trivial) datatype invariant over (82):

$$\begin{array}{l} \phi(k, \langle t, t' \rangle) = \\ \quad (\forall \langle k, w \rangle, \langle k', w' \rangle \in t : k = k' \Rightarrow w = w') \wedge \\ \quad (\forall \langle k_1, a, k_1' \rangle, \langle k_2, b, k_2' \rangle \in t' : (k_1 = k_2 \wedge a = b) \Rightarrow k_1' = k_2') \wedge \\ \quad \pi_1[t'] \subseteq \pi_1[t] \wedge \\ \quad aux(k, t, t', \emptyset) \end{array} \tag{89}$$

where aux is the recursive auxiliary predicate given by (88).

5.3 Operation Reification

From the operations which are expected over type $DecTree$ we select the following,

$$\begin{array}{l} decide : DecTree \times Answer \to DecTree \\ decide(\sigma, a) \stackrel{\mathrm{def}}{=} \ \begin{array}{ll} let & m = dom(\pi_2(\sigma)) \\ in & \left\{ \begin{array}{lcl} \sigma & \Leftarrow & a \notin m \\ (\pi_2(\sigma))(a) & \Leftarrow & a \in m \end{array} \right. \end{array} \end{array} \tag{90}$$

which specifies the action of picking a particular answer a available at the root menu of decision tree σ and selecting the corresponding sub-tree (if a is a valid answer).

We want to calculate the reification of *decide* at reification-level 4, that is *wrt.* $DecTree_4$ (82). Following [7, 33], we start by building the corresponding (commutative) reification diagram:

$$\begin{tikzcd} DecTree \times Answer \arrow[r, "decide"] & DecTree \\ DecTree_4 \times Answer \arrow[u, "f \times 1_{Answer}"] \arrow[r, "decide_4"'] & DecTree_4 \arrow[u, "f"'] \end{tikzcd}$$

which leads to equation

$$f(decide_4(k, \langle t, t' \rangle, a)) = decide(f(k, \langle t, t' \rangle), a) \tag{91}$$

where $decide_4$ is regarded as the unknown and f is abstraction map (86). Substituting (90) in (91) we obtain

$$\begin{array}{rcl} f(decide_4(k, \langle t, t' \rangle, a)) & = & decide(f(k, \langle t, t' \rangle), a) \\ & = & let \quad m = dom(\pi_2(f(k, \langle t, t' \rangle))) \\ & & in \quad \left\{ \begin{array}{lcl} f(k, \langle t, t' \rangle) & \Leftarrow & a \notin m \\ (\pi_2(f(k, \langle t, t' \rangle)))(a) & \Leftarrow & a \in m \end{array} \right. \end{array}$$

Assuming the following definitions, which specify usual *projection/ selection* relational operators,

$$\begin{array}{l} proj : \{1, \ldots, n\} \times 2^{A_1 \times \ldots \times A_n} \longrightarrow \bigcup_{i=1}^{n} 2^{A_i} \\ proj(i, t) \stackrel{\mathrm{def}}{=} \{\pi_i(r) | \; r \in t\} = \pi_i[t] \end{array}$$

and

$$\begin{array}{l} sel : (\sum_{i=1}^{n} A_i) \times 2^{A_1 \times \ldots \times A_n} \longrightarrow 2^{A_1 \times \ldots \times A_n} \\ sel(\langle i, a \rangle, t) \stackrel{\mathrm{def}}{=} \{r \in t | \; a = \pi_i(r)\} \end{array}$$

it is easy to show that

$$dom(\pi_2(f(k, \langle t, t' \rangle))) = proj(2, sel(\langle 1, k \rangle, t'))$$

and that

$$\begin{array}{rll} (\pi_2(f(k, \langle t, t' \rangle)))(a) = & let & t'' = sel(\langle 1, k \rangle, t') \\ & & t''' = sel(\langle 2, a \rangle, t'') \\ & & k' = the(proj(3, t''')) \\ & in & f(k', \langle t, t' \rangle) \end{array}$$

Then

$$\begin{array}{l} f(decide_4(k, \langle t, t' \rangle, a)) = \\ \qquad let \quad m = proj(2, sel(\langle 1, k \rangle, t')) \\ \qquad in \quad f(\left\{ \begin{array}{lcl} \langle k, \langle t, t' \rangle \rangle & \Leftarrow & a \notin m \\ let \quad k' = the(proj(3, sel(\langle 2, a \rangle, sel(\langle 1, k \rangle, t')))) & \Leftarrow & a \in m \\ in \quad \langle k', \langle t, t' \rangle \rangle & & \end{array} \right.) \end{array} \tag{92}$$

Removing f from both sides of (92) we obtain

$$
\begin{array}{l}
decide_4(k, \langle t, t' \rangle, a) \stackrel{\text{def}}{=} \\
\qquad \begin{array}{ll}
let & m = proj(2, sel(\langle 1, k \rangle, t')) \\
 & k' = \left\{ \begin{array}{lcl} k & \Leftarrow & a \notin m \\ the(proj(3, sel(\langle 2, a \rangle, sel(\langle 1, k \rangle, t')))) & \Leftarrow & a \in m \end{array} \right. \\
in & \langle k', \langle t, t' \rangle \rangle
\end{array}
\end{array}
$$

which implements the expected "pointer-handling" operation in a relational database programming style. It should be noted that $decide_4$ is not a unique solution to (91) because f (86) is non-injective. Other valid solutions might "garbage collect" t and t' by removing all entries accessed by k (which will not be revisited, *cf.* invariant (89)).

6 Conclusions

The research described in this paper extends the work on the Sets calculus reported in [34, 35]. The $\trianglelefteq$-subcalculus is studied in much more detail and includes generic results for refining recursive data models.

Many laws of Sets are trivial from a strict mathematical viewpoint. What appears to be relevant is their use in the context of model-oriented reification. The synthesis of arbitrarily complex abstraction maps and data-type invariants induced by data refinement, resorts to a "functorial" approach whose compactness pays off the initial effort in understanding its categorial foundation. However, it is apparent from the final example of the paper that "pen and pencil" inference of such maps and invariants may become fairly laborious tasks in sizeable, real-life reifications. Resorting to a functional transformation tool (*e.g.* a system such as NUPRL [6] or ERIL [8]) may be of some help in this respect.

Section 5 illustrates how dramatically the complexity of data-type invariants may increase throughout reification. This provides good evidence of how insecure informal designs may become if invariants are either ignored or misrecorded.

7 Future Work

The Sets calculus is still in its infancy and there are several lines along which it may be developed.

7.1 Foundations

The categorial foundations of Sets need further investigation, particularly with respect to its naive basis on set-theoretical *cardinality*. Apparently, one is free to choose an abstraction map (surjection) $f : B \to A$ between two domains A and B such that $A \trianglelefteq B$. But this freedom appears to be available only at very basic data domains (*e.g.* $elems$ from sequences to sets) and is denied by the *functorial* strategy adopted in this paper for structural composition of data refinements. In fact, it would be "unnatural" to choose an abstraction map

$$g : \mathcal{F}(B) \to \mathcal{F}(A)$$

other than $\mathcal{F}(f)$ if f is the "natural" abstraction map between B and A, *cf.* theorem 2. Such a *functorial* naturalness of some abstraction maps (and corresponding invariants) is even more evident wherever recursion is around, *cf.* section 4.

This brings to mind Wand's *final algebra* approach to specification [45] [11]:

> [...] for any object A of K, there is only one morphism in K from A to W. (There is only one "reasonable" abstraction map for each data type representation A, *i.e.*, each "concrete" value in A may reasonably represent only one "abstract" value in W.) [...] that is: *an abstract data type is a final object in the category of its representations* [...]

In this *final* approach to refinement, epimorphism (2) will become *unique* and reifications will have to form a *comma category* [45].

Alternatively, some kind of "ordering" is required on multiple reification. In fact, there seems to be no "best" way in which a given data domain A may be reified by some other domain B. Which of the following "limits" is better from a calculation viewpoint: is it to have as much redundancy as necessary to avoid invariants on B? or is it to introduce as many invariants as necessary in order for A to be isomorphic to the corresponding subset of B? The latter alternative provides unique solutions to algorithmic refinements (recall section 5.3), but invariants grow in complexity. Moreover, this problem has model-theoretical implications with non-determinism, observational equivalence, implementation bias *etc.* [32, 38].

Other aspects which may require better foundations are "context-sensitive" transformations [35], "ad hoc" invariants [39] and recursion handling [12] [22]. The main result of the paper should be extended from polynomial to co-continuous functors, in a category theoretical setting.

7.2 Operation Refinement

SETS claims no originality on operation refinement. Section 5.3 illustrated how "program transformation"-like reasoning can be used to calculate functional refinements. But other techniques could have been used, *e.g.* the "Oxford calculus" [30, 29]. In fact, it is straightforward to combine this calculus with SETS, as is apparent from the following quotation from [30]:

> An *abstraction invariant AI* is chosen which links the abstract variables a and the concrete variables c. [...] In many cases, the abstraction invariant is *functional* [...] and can always be written as a conjunction
>
> $$(a = AF(c)) \wedge CI(c)$$
>
> where AF we call the *abstraction function* and CI the *concrete invariant.*

[11] In the following quotation from [45], W is an "abstract data type" and K is the category of representations of W.

[12] Note that some interesting reification steps actually *introduce* (rather than remove) recursion [7]. This kind of reification is particularly relevant *wrt.* parallel runtime environments.

in which one immediately identifies a $\trianglelefteq^{CI}_{AF}$-refinement step which may be calculated using SETS rather than being simply "chosen".

7.3 "Reverse" Calculation

Another direction of future research is (formal) *reverse specification*. Some experimentation carried out by the author suggests that SETS may become useful in reversing pre-existing code, by performing calculations in the reverse direction, that is $\trianglerighteq$-calculations instead of $\trianglelefteq$-calculations (from implementation to specification). As a side effect, such pieces of code are likely to be improved since code-level invariants may be unveiled which were never recorded in the documentation.

7.4 Horizontal Refinement and Classification

Yet another potential application of the calculus concerns *horizontal refinement* [13], that is the decomposition of large software systems in terms of re-usable *components* [41]. Many laws of *Sets* work in this direction, particularly the ones which "push the $\times$ construct outwards" — *e.g.* (38), (39), (42), (45), (60) *etc.* — because they help in factorizing complex state structures of monolithic designs into collections of simpler structures which may have been dealt with already, and may thus be re-used.

The hierarchical *classification* and retrieval of re-usable abstract models is suggested above in section 3.4.1 and has been further developed in [36] and [37], where a *repository* of such models is hierarchically organized in terms of the subterm instantiation ordering on the *Sets* expressions which denote the classes of their internal states. The same repository records $\trianglelefteq$-reification relationships among components.

Acknowledgements

The author wishes to thank Isabel Jourdan (Dept. of Mathematics, University of Coimbra) for her comments on earlier drafts of this paper. This work has been partly supported by the JNICT council under PMCT contracts nrs. 87/63 and 87/66.

References

[1] Arbib M.A., Kfoury A., Mool R.N. *A Basis for Theoretical Computer Science.* Texts and Monographs in Computer Science (D.Gries Ed.), Springer-Verlag, 1981.

[2] Aceto L., Hennessy M. *Towards Action-refinement in Process Algebras.* Computer Science Report 3/880, University of Sussex.

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

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

[5] Burstall R.M., Goguen J.A. *An Informal Introduction to Specifications Using Clear.* In *The Correctness Problem in Computer Science* (Boyer R.S., Moore J.S.—Eds.), 185–213, Academic Press, 1981.

[6] Constable R.L. *et al. Implementing Mathematics in the Nuprl Proof Development System.* Prentice-Hall International, 1986.

[7] Darlington J. *Program Transformation.* In *Funct. Prog. and Its Applications: An Advanced Course*, Univ. of Newcastle. Cambridge Univ. Press, 1982.

[8] Dick J.J. *ERIL User's Manual v. R1.6a.*, RAL-88-055, September, 1988.

[9] Dijkstra E.W. *A Discipline of Programming.* Prentice-Hall, 1976.

[10] Fielding E. *The Specification of Abstract Mappings and Their Implementation as B^+-Trees.* PRG-18, Oxford Univ., Sep. 1980.

[11] Goguen J.A., Thatcher J.W., Wagner E.W., Wright J.B. *Initial Algebra Semantics and Continuous Algebras.* JACM 24:1, 68-95, January 1977.

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

[13] Goguen J.A. *Reusing and Interconnecting Software Components.* IEEE Computer, 19(2), 16–28, February 1986.

[14] Guttag J.V., Horning J.J. *The Algebraic Specification of Abstract Data Types.* Acta Informatica 10, 27-52, 1978.

[15] Hennessy M. *Algebraic Theory of Processes.* MIT Press Series in the Foundations of Computing, 1988.

[16] Hennessy M., Milner R. *Algebraic Laws for Nondeterminism and Concurrency.* JACM 32(1):137–161, Jan. 1985.

[17] Hoare C.A.R., He J.F., Sanders J.W. *Prespecification in Data Refinement.* Inf. Proc. Letters 25, 71-76, 1987.

[18] Jifeng H. *Process Refinement.* In *The Theory and Practice of Refinement*, McDermid (Ed.), Butterworths, 1989.

[19] Jones C.B. *Software Development — A Rigorous Approach.* Prentice-Hall, 1980.

[20] Jones C.B. *Systematic Software Development Using VDM.* Prentice-Hall, 1986.

[21] Jones C.B. *Interference Resumed.* Technical Report UMCS-91-5-1, University of Manchester, 1991.

[22] Jourdan I.S. *Reificação de Tipos Abstractos de Dados: Uma Abordagem Matemática.* M.Sc. thesis, University of Coimbra, 1991 (in Portuguese).

[23] Knuth D. *Structured Programming with Goto Statements.* ACM Comp. Surveys, 6:(4), Dec. 1974.

[24] Manes E.G., Arbib M.A. *Algebraic Approaches to Program Semantics.* Springer-Verlag, 1986.

[25] Milner R. *Flowgraphs and Flow Algebras.* JACM 26(4):794–818, Oct. 1979.

[26] Milner R. *Interpreting One Concurrent Calculus in Another.* Proc. of the Int. Conference on Fifth Generation Computer Systems, 321–326, 1988.

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

[28] Möller B. (Ed.) *A Survey of the Project CIP: Computer-aided, Intuition-Guided Programming Wide Spectrum Language and Program Transformations.* Report TUM-I8406, T. Universität München, July 1984.

[29] Morgan C. *Programming from Specification.* Prentice-Hall International, 1990.

[30] Morgan C., Gardiner P.H.B. *Data Refinement by Calculation.* Acta Informatica 27, 481-503, 1990, Springer-Verlag.

[31] Mosses P. *Abstract Semantic Algebras!* Report DAIMI PB-145, Aarhus University, July 1982.

[32] Nipkow T. *Nondeterministic Data Types: Models and Implementations.* Acta Informatica 22, 629-661, Springer-Verlag, 1986.

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

[34] Oliveira J.N. *Refinamento Transformacional de Especificações (Terminais).* Actas das XII *Jornadas Luso-Espanholas de Matemática*, Vol.II, 412-417, 4-8 Maio 1987, Braga, Portugal (in Portuguese).

[35] Oliveira J.N. *A Reification Calculus for Model-Oriented Software Specification.* Formal Aspects of Computing, Vol.2, 1-23, 1990, Springer-Verlag.

[36] Oliveira J.N. *In CASE You Use Formal Methods ... or Formal Methods Just In CASE!* Invited communication, CASE'91 Symposium, Braga, Portugal, 10–11 May 1991 (in Portuguese).

[37] Oliveira J.N. *Especificação Formal de Programas.* Lecture Notes for M.Sc. Course in Computing, University of Minho, 1991 (in Portuguese).

[38] Oliveira J.N. *A Note on $\trianglelefteq$-Calculation, Nondeterminism and Implementation Bias.* DI/INESC Technical Report (in preparation).

[39] Oliveira J.N. *Hash Tables: A Case Study in $\trianglelefteq$-Calculation.* DI/INESC Technical Report (in preparation).

[40] Pacheco O. *Concepção de um Sistema Pericial para o Registo Predial.* Relatório de Estágio, LESI, U.Minho, Braga 1990 (in Portuguese).

[41] ROSE Consortium — E.S.F. *The ROSE Subproject.* Presentation at the Workshop on Reuse of Software Development Components, Joint Organization CEC-ESF, Berlin, 25-26 October 1989.

[42] Shaw R.C., Jones C.B. *Case Studies in Systematic Software Development.* Prentice-Hall, 1990.

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

[44] Vogler W. *Bisimulation and Action Refinement.* Technical Report INFO-05-90-120-480/1.-FMI, T. Universität München, 1990.

[45] Wand M. *Final Algebraic Semantics and Data Type Extensions.* JCSS 19, 27-44, 1979.

[46] Wirth N. *Algorithms + Data Structures = Programs.* Prentice-Hall Int., 1976.

[47] Wolczko M. *Semantics of Object-oriented Languages.* Tech. Report UMCS-88-6-1, U.Manchester, 1988.

[48] Wolczko M. *Garbage Collection.* in [42], 211-233.