Data Reification without Explicit Abstraction Functions

T Clement

Adelard**,
Coborn House,
3, Coborn Road,
LONDON E3 2DA,
U.K
e-mail: tpc@adtclem.demon.co.uk

Abstract. Data reification normally involves the explicit positing of an abstraction function with certain properties. However, the condition for one definition to reify another only requires that a function with such properties should exist. This suggests that it may be possible to carry through a data reification without giving an explicit definition of the abstraction function at all. This paper explores this possibility and compares it with the more conventional approach.

1 Introduction

An abstract type is a type name equipped with some operations, usually with parameters or results of other, previously defined, types. The *signature* of the abstract type gives the names of the operations and their types: for example, the abstract type Set, representing sets of integers, might have signature

$empty: \rightarrow Set$
$add: \mathbb{Z} \times Set \rightarrow Set$
$elem: \mathbb{Z} \times Set \rightarrow \mathbb{B}$

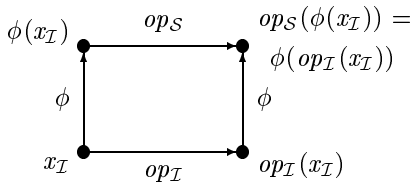
Because *empty* and *add* have results of type Set, they are said to be *constructors* of the type. The function *elem* is an *observer* because its result is of a predefined type. For technical reasons, it is convenient to consider only signatures with results of single types rather than tuples, so all operations are either constructors or observers.

In the model based approaches to formal specification, such as VDM or Z, an abstract type is defined by defining a set of values for the type name and defining the operations as (possibly partial) functions on this set and others corresponding to the predefined types. The signature is usually a part of these definitions rather than separate as above. It is possible to give many definitions of the same signature (which is why it will be convenient to separate it out here), and these will be distinguished by subscripting the type name and operation names. In

** This work was carried out while at the University of Manchester

particular, when discussing reifications, the definitions of the specification will be identified by subscript $\mathcal{S}$ and those of the implementation by subscript $\mathcal{I}$.

One definition of abstract type T will implement another by data reification if the value of every term of the signature of type other than the abstract type (of *visible* type) is the same in both specification and implementation whenever it is defined by the specification. This will be the case if there exists a (possibly partial) function ϕ: $T_\mathcal{I} \to T_\mathcal{S}$ (an *abstraction function*) such that for all operations op: $T \to T$ in the signature the square



commutes whenever the upper path is defined. (A justification will be found in Clement(1993)). Using $\delta(t)$ to signify that term t is defined, we can write this *reification condition* for an operation formally as

$$\forall x \in T_\mathcal{I} \cdot \delta(op_\mathcal{S}(\phi(x))) \;\Rightarrow\; \phi(op_\mathcal{I}(x)) = op_\mathcal{S}(\phi(x))$$

If there is a parameter of a predefined type, this becomes

$$\forall x \in T_\mathcal{I}, y \in Y \cdot \delta(op_\mathcal{S}(y, \phi(x))) \;\Rightarrow\; \phi(op_\mathcal{I}(y, x)) = op_\mathcal{S}(\phi(x))$$

while for observations (with no parameters) it reduces to

$$\forall x \in T_\mathcal{I} \cdot \delta(op_\mathcal{S}(\phi(x))) \;\Rightarrow\; op_\mathcal{I}(x) = op_\mathcal{S}(\phi(x))$$

The traditional approach to data reification is to posit an implementation representation of the abstract type and definitions of the operations of the signature on it, and then to posit a function and prove that it has the desired properties. In the alternative, calculational, approach (Darlington (1984); Morgan & Gardiner (1990)), the reification conditions are used as the defining properties of the operations. A more algorithmic definition working directly on the new representation can then be derived using conventional calculational techniques. However, it is still necessary to pick a set of values for the type and posit an abstraction function. Both are thus constructive approaches to showing the existence of an abstraction function. This raises the question of whether a non-constructive approach is feasible: if so, it may offer the advantage of not having to demonstrate that a posited function satisfies the reification conditions. In this paper we shall look at such an approach and assess its benefits and limitations.

In the next section, we shall present the basic idea for total operations. The techniques will be refined in Sect. 3 to deal with partial operations and abstraction functions. An extended example of the approach will be given in Sect. 4, and this will be compared with a more conventional approach to the same problem in Sect. 5. The last section will summarize and draw some conclusions. The

paper assumes a familiarity with the basic definition of categories (this will be found in Arbib & Manes (1975)) but more specialized standard results will be summarized here for completeness.

2 Constructing Categories

We can construct an implementation of an abstract type with signature Σ in the following way. We first fix interpretations for all the visible types of the signature as sets: intuitively, we can see these as the interpretations fixed by the prior specifications of these types (or the semantics of VDM). We can then construct a *heterogeneous algebra* (Goguen, Thatcher & Wagner (1978)) for the signature by taking these sets as the carriers of their types and some arbitary set as the carrier of the abstract type (we shall write $T_{\mathcal{A}}$ for the carrier of type T in algebra $\mathcal{A}$), and associating each constructor *op* in the signature with a function $op_{\mathcal{A}}$ from the appropriate carrier types to the carrier of the abstract type. (The observer functions will be considered separately later.) The specification of the abstract type can be identified with one such algebra, where the abstract type carrier is the set corresponding to the model chosen in the specification, and the functions meet the specifications of the operations.

We define a morphism from algebra $\mathcal{I}$ to algebra $\mathcal{S}$ to be an indexed collection of functions

$$\phi = \{\phi_T\colon T_{\mathcal{B}} \rightarrow T_{\mathcal{A}} \mid T \text{ is a type in the signature}\}$$

For the visible types, we fix each function as the appropriate identity function: for the abstract type, any function satisfying the reification conditions on the constructors is allowed.

The algebras and morphisms together form a category. (Clearly, the identity function on the abstract type provides an identity morphism, and identities compose to give identities while composition preserves satisfaction of the reification condition.) The category is a subcategory of one which is well known in algebraic specification (see Ehrig & Mahr (1985), for example), where there is a free choice of carrier sets for all types in the signature, and for each function *op*: $T_1 \times \ldots T_n \rightarrow T$ the morphisms satisfy the condition

$$\phi_T(op_{\mathcal{I}}(x_1, \ldots, x_n)) = op_{\mathcal{S}}(\phi_{T_1}(x_1), \ldots, \phi_{T_n}(x_n))$$

(This is easily seen to specialize to the reification condition if all the ϕ_T functions for visible T are identities.)

The full category has *initial objects*: that is, objects with a (unique) morphism to every object of the category. One initial object (the *term algebra* $\mathcal{T}_\Sigma$, or $\mathcal{T}$ when the signature is clear from the context) is constructed by taking the set of terms of each type as the carrier of that type (the signatures that arise in algebraic specification have constructors of all types) and defining the function corresponding to an operation to map terms to the application of the operation to those terms. The morphism to any algebra $\mathcal{A}$ (to be written $\_^{\mathcal{A}}$) uses the functions of $\mathcal{A}$ to evaluate the term from the term algebra, so for any constant c in

the signature $c^{\mathcal{A}} = c_{\mathcal{A}}$ and for any term $op(t)$, $(op(t))^{\mathcal{A}} = op_{\mathcal{A}}(t^{\mathcal{A}})$. This clearly satisfies the condition on morphisms. Any other object which is isomorphic to this is also initial.

The category we have just constructed also has initial objects. One can be produced using the construction above for the carrier of the abstract type, but using values from the other carriers rather than terms of the type as parameters. We shall call this the *relative term algebra*, and also denote it by $\mathcal{T}$ and the morphism to any other algebra by $\_^{\mathcal{A}}$. Since initial objects have a morphism to all objects, and the specification is one of the objects of the category, the relative term algebra defines an implementation of the type and its constructors. (Intuitively, an abstract type can be implemented whatever its specification by remembering in its values the values and operations used in their construction, since this gives enough information to define any observation.) In general, algebras isomorphic to this (and hence also initial) will give more interesting implementations.

For example, consider implementing sets with the signature of Sect. 1. Since they are predefined in VDM, the specification is trivial

$$empty_{\mathcal{S}} = \{\,\}$$

$$add_{\mathcal{S}}(n, s) \triangleq s \cup \{n\}$$

$$elem_{\mathcal{S}}(n, s) \triangleq n \in s$$

The carrier of the abstract type in $\mathcal{T}$ is terms of the form

$$add(n_1, add(n_2, \ldots add(n_n, empty)))$$

(which we can abbreviate as $add^n(n_1, \ldots, n_n, empty)$). The following definition describes an algebra which is more useful as an implementation.

$$Set_{\mathcal{I}} = X^*$$
$$empty_{\mathcal{I}} = [\,]$$
$$add_{\mathcal{I}}(x, s) \triangleq \mathbf{cons}(x, s)$$

(It is easy to see that it is isomorphic to the relative term algebra, but this could be verified by exhibiting a formal definition of the isomorphism.) This is a well-known implementation, of course, but the justification given by this approach is by appeal to abstract mathematics rather than by detailed calculation, and of course the abstraction function has not been made explicit.

We still need to derive an implementations of the observation *elem*. Following the calculational style of Darlington (1984), we use the reification condition to state an equality the implementation must satisfy

$$elem_{\mathcal{I}}(x, s) = elem_{\mathcal{S}}(x, \phi(s))$$

To find a definition of $elem_{\mathcal{I}}$ with no uses of ϕ satisfying this equation, we can consider cases of the structure of the set representation:

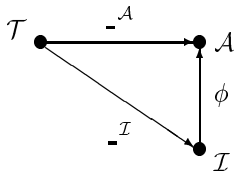
$$
\begin{aligned}
elem_{\mathcal{I}}(x, [\,]) &= elem_{\mathcal{I}}(x, empty_{\mathcal{I}}) \\
&= elem_{\mathcal{S}}(x, \phi(empty_{\mathcal{I}})) \\
&= x \in empty_{\mathcal{S}} \\
&= \mathbf{false} \\
elem_{\mathcal{I}}(x, \mathbf{cons}(y, s)) &= elem_{\mathcal{I}}(x, add_{\mathcal{I}}(y, s)) \\
&= elem_{\mathcal{S}}(x, add_{\mathcal{S}}(y, \phi(s))) \\
&= x \in (\{y\} \cup \phi(s)) \\
&= x = y \vee x \in \phi(s) \\
&= x = y \vee elem_{\mathcal{I}}(x, s)
\end{aligned}
$$

The strategy is to express each value of the type as a term of the signature. (This is always possible, because the values are reachable by construction: the algebra has *no junk*.) We then apply the reification condition for the constructors, and use the definitions from the specification to deduce a simpler form of the equality. Recursion is introduced by using the defining equation again. The resulting equations clearly hold of the following VDM function definition, which expresses the expected implementation.

$$
\begin{aligned}
elem_{\mathcal{I}}(x, s) \quad \triangleq \quad & \mathbf{cases}\ s\ \mathbf{of} \\
& \qquad [\,] \rightarrow \mathbf{false} \\
& \mathbf{cons}(y, s) \rightarrow x = y \vee elem_{\mathcal{I}}(x, s) \\
& \mathbf{end}
\end{aligned}
$$

There are of course many implementations of specifications with signature Σ which are not isomorphic to T_Σ. To calculate more of them, we need to define other categories with definitions (including the specification) as objects and functions satisfying the reification conditions as morphisms, and with initial objects. The theory of algebraic specification tells us that the category of all algebras satisfying a given set of equations is known to have initial algebras, and this remains true when we restrict the category by fixing carriers of visible types. The equations are then only needed to restrict the possible choices for the abstract type, and should therefore only involve terms of that type. Any choice of equations satisfied by the specification will define an implementation, although some of these implementations will be of more interest than others.

The carrier of the abstract type in an initial algebra $\mathcal{I}_{\Sigma,E}$ in the category generated by signature Σ and set of equations E can be constructed from T_Σ by partitioning it into sets which are provably equal given the equations and the usual properties of equality. This clearly satisfies the equations, but "only just": no terms which do not have to be equal are equal. We say that there is *no confusion*. If we write $[t]$ for the set of terms equal to t, the operations are defined by $op_{\mathcal{I}}([t]) = [op(t)]$. This is well defined because equality is a congruence. We shall call the result the *quotient algebra* of Σ and E. It is initial because a morphism ϕ can be defined to any other algebra $\mathcal{A}$ satisfying E by taking $\phi([t]) = t^{\mathcal{A}}$. This is well defined because there is no confusion in the quotient algebra. It is obvious that for any term t, $t^{\mathcal{A}} = \phi(t^{\mathcal{I}})$: that is, that



commutes.

As an example of calculating an implementation using equations, we may take sets with the following signature:

$empty: \rightarrow Set$
$unit: X \rightarrow Set$
$union: Set \times Set \rightarrow Set$
$elem: X \times Set \rightarrow \mathbb{B}$

with specification

$Set_{\mathcal{S}} = X\textbf{-set}$

$empty_{\mathcal{S}} = \{\,\}$

$unit(x) \triangleq \{x\}$

$union_{\mathcal{S}}(s_1, s_2) \triangleq s_1 \cup s_2$

$elem_{\mathcal{S}}(x, s) \triangleq x \in s$

The constructor terms from this signature are $empty$, $unit(x)$ for all x in the carrier, $union(empty, unit(x))$, $union(unit(x_1), unit(x_2))$ and so on: they are isomorphic to the more familiar structure of binary trees with the values in the leaves only, *unit* corresponding to the leaf constructor and *union* to the binary tree constructor. To arrive at the sequence representation from this signature and set of definitions, we show that the equations

$union(empty, s) = s$
$union(s, empty) = s$
$union(s_1, union(s_2, s_3)) = union(union(s_1, s_2), s_3)$

hold of the specification (trivially in this case, as properties of $\cup$) and consider the category of algebras where these equations hold of the *union* function. The equations for sets equate terms containing *empty* with terms which do not (except for *empty* itself, which is in a partition of its own), and equate terms where the same elements appear in the same order as arguments to *unit* to each other irrespective of how these are distributed as arguments to *union*. In looking for

isomorphic models, it is often helpful to select a canonical term from each partition and look at its form. The equations will rewrite any term to one of the form

$$union(unit(x_1), union(unit(x_2), \ldots, union(unit(x_n, empty))))$$

(deliberately putting back one *empty*) and these are readily seen to be isomorphic to sequences $[x_1, \ldots, x_n]$, giving the definition

$$Set_{\mathcal{I}} = X^*$$

$$empty_{\mathcal{I}} = [\,]$$

$$unit_{\mathcal{I}}(x) \triangleq [x]$$

$$union_{\mathcal{I}}(s_1, s_2) \triangleq append(s_1, s_2)$$

To derive the expected definition of $elem_{\mathcal{I}}$, we may begin again with the definition derived from the reification condition.

$$elem_{\mathcal{I}}(x, s) \triangleq elem_{\mathcal{S}}(x, \phi(s))$$

Considering cases of the value of s again, the derivation of $elem_{\mathcal{I}}(x, [\,]) = \mathbf{false}$ goes through as before and

$$\begin{aligned} elem_{\mathcal{I}}(x, \mathbf{cons}(y, s)) &= elem_{\mathcal{I}}(x, union_{\mathcal{I}}(unit_{\mathcal{I}}(y), s)) \\ &= elem_{\mathcal{S}}(x, \phi(union_{\mathcal{I}}(unit_{\mathcal{I}}(y), s))) \\ &= elem_{\mathcal{S}}(x, union_{\mathcal{S}}(unit_{\mathcal{S}}(y), \phi(s)))) \\ &= elem_{\mathcal{S}}(x, union_{\mathcal{S}}(unit_{\mathcal{S}}(y), \phi(s)))) \\ &= elem_{\mathcal{S}}(x, unit_{\mathcal{S}}(y)) \vee elem_{\mathcal{S}}(x, \phi(s)) \\ &= x = y \vee elem_{\mathcal{I}}(x, s) \end{aligned}$$

giving the same definition as above.

3 Partial Operations

Most abstract types have operations which are not defined everywhere on the type. For example, in the signature

empty: *Stack*
push: $\mathbb{Z} \times Stack \to Stack$
pop: $Stack \to Stack$
top: $Stack \to \mathbb{Z}$

we do not expect *pop* and *top* to be defined on *empty*. In VDM, such operations are defined as potentially partial functions, so abstract type definitions are *partial* (heterogeneous) *algebras* (Broy & Wirsing (1982)). The reification condition of Sect. 1 allows for a partial function to be implemented by one which is more defined. (This is reasonable in circumstances where all uses of a function are

proved to lie in its domain, as they are in VDM.) It also allows the abstraction function itself to be partial.

As before, abstraction functions can be extended to provide abstraction morphisms on partial algebras by adding (total) identity functions for the carriers of the predefined types. The class of all partial algebras for a given signature and interpretation of the predefined types together with the abstraction morphisms again forms a category. Unlike the categories built on total algebras, there are in general no initial objects. However, the relative term algebra, which is of course total, is privileged in the sense that it has at least one morphism to every other algebra $\mathcal{A}$ and hence remains an implementation. (Once again, because it remembers everything about the construction of the value, it is not surprising that anything can be implemented using it.) We can construct this morphism as before by giving terms which are defined in $\mathcal{A}$ their value there. Those which are not are left unmapped. This is easily seen to satisfy the reification condition, and is the morphism we shall denote by $\_^{\mathcal{A}}$. However, it may also be possible to map undefined terms to some value in $\mathcal{A}$ while still satisfying the reification condition (one not in the domain of any operation in $\mathcal{A}$ will do), which is why there is not always an initial object. Similarly, when the category of algebras is restricted to one where the operations on the new type satisfy given equations, the (total) quotient algebra has a morphism ϕ to every other algebra $\mathcal{A}$ in the category: again, it maps the value $[x]$ to $x^{\mathcal{A}}$ (where $x^{\mathcal{A}}$ is defined) and again, this is well defined because equality is a congruence and the equations are satisfied by all the algebras. There is an issue here of what is meant by an algebra satisfying an equation when some terms may be undefined. We shall take the *strong* interpretation of equality: an instance of an equation will hold if the two sides are defined and have the same value or if both are undefined.

Equations were introduced in Sect. 2 to define restricted categories of algebras with new initial objects and hence generate more possible implementations. Now we want to introduce some further properties of algebras to generate implementations where some of the functions are partial. This will be done by using axioms of the form $U(t)$ to say that term t is undefined. For example, we might give $U(pop(empty))$ as an axiom on stacks. As with equations, the relative term algebra provides the basis for constructing an algebra which just meets any undefinedness requirement. The carrier for the abstract type is constructed by deleting from the set of terms all those which are demonstrably undefined: that is, those containing terms which are explicitly stated to be undefined. (Functions are strict in VDM.) For example, $push(x, pop(empty))$ and similar terms will be removed from the carrier as well as $pop(empty)$. As usual, the operations just map values to the term which is the application of the operation to those values: where this is an undefined term the operation is undefined. There is a morphism ϕ to any algebra satisfying the undefinedness axioms, and thus with at least the same terms undefined. It is essentially that constructed from the relative term algebra, the difference being that the necessarily undefined terms do not appear in the carrier, rather than being left unmapped by the morphism. The algebra

just constructed is thus an implementation of any other algebra with the same undefinedness properties.

Equations and undefinedness axioms can be combined. An initial algebra in the category satisfying all the axioms can be produced by taking the quotient algebra and deleting all sets of terms containing an undefined term from the carrier. Where a function would have given such a set as a result, it becomes undefined. The morphism to any algebra $\mathcal{A}$ satisfying all the axioms is defined as usual to map $[t]$ to $t^{\mathcal{A}}$ when these are defined. Such morphism definitions have the property that $t^{\mathcal{A}} = \phi(t^{\mathcal{I}})$, and hence ϕ satisfies the stronger reification condition $\phi(op_{\mathcal{I}}(x)) = op_{\mathcal{S}}(\phi(x))$ (where the equality is strong in each case). As before, we would usually look for a model isomorphic to the constructed one but based on more familiar structures to provide the actual implementation.

As an example, this construction can be applied to stacks. The stack operations are known to satisfy the equation

$$pop(push(x, s)) = s$$

This partitions the terms into two kinds of sets: those containing terms where no subterm has more *pop*s than *push*es, which each have a canonical term of form $push^n(x_1, \ldots, x_n, empty)$; and the rest, each containing a canonical term of the form $push^n(x_1, \ldots, x_n, pop^m(empty))$ (for $m > 0$ and $n \geq 0$). (The canonical terms are achieved by using the equation to cancel as many *pop*s as possible.) The sets of the second kind clearly contain terms which are undefined as a consequence of the U axiom above, and are deleted. The carrier is thus the sets of the first kind, and their canonical terms clearly show them to be isomorphic to the sequences $[x_1, \ldots, x_n]$. The empty sequence and **cons** operation are easily seen to correspond to *empty* and *push*. A legal *pop* on a canonical term can be cancelled with the preceding *push* to get a new canonical term with the first *push* of the old one removed, while $pop(empty)$ is undefined, so *pop* corresponds to **tl**. This is the usual choice of model for stacks in VDM, constructed here from an algebraic specification.

There is a technical issue that arises when VDM is used to specify the abstract type as a partial algebra. The domain of a function is characterized by a precondition, and the semantics of the VDM standard (ISO (1993)) states that the function must be defined when the precondition holds, but is not necessarily undefined when it does not. However, the standard VDM reification condition is

$$\forall x \in T_{\mathcal{I}} \cdot \delta(\phi(x)) \wedge \textbf{pre-}op_{\mathcal{S}}(\phi(x)) \;\Rightarrow\; \phi(op_{\mathcal{I}}(x)) = op_{\mathcal{S}}(\phi(x))$$

which is also weaker and will be satisfied by any implementation constructed as described.

4 An Example

A more interesting example than those used above to illustrate the approach arises in the implementation of substitutions for the efficient unification algo-

rithm described in Boyer & Moore (1972). The signature of the operations on substitutions needed by this algorithm is

$$\begin{array}{l} \mathit{empty}\colon \mathit{Substitution} \\ \mathit{update}\colon \mathit{VarSymbol} \times \mathit{Term} \times \mathit{Substitution} \to \mathit{Substitution} \\ \_ \bullet \_\colon \mathit{Term} \times \mathit{Substitution} \to \mathit{Term} \end{array}$$

where the constant *empty* is a substitution having no effect on any term, *update* adds a new binding of a term to a variable, and $\bullet$ applies a substitution to a term. The predefined type *VarSymbol* is an arbitrary set, while *Term* may be taken to be defined by

$$\begin{array}{lll} CT :: & \mathit{op} & : \mathit{FnSymbol} \\ & \mathit{args} & : \mathit{Term}^* \end{array}$$

$$Var :: v : \mathit{VarSymbol}$$

$$\mathit{Term} = \mathit{Var} \mid CT$$

We assume that a function $\mathit{vars}(t)$ yielding the set of variables in term t is predefined on *Term*.

Substitutions may be specified as maps which bind variables to the terms which will replace them when the substitution is applied.

$$\mathit{Substitution}_{\mathcal{S}} = \mathit{VarSymbol} \xrightarrow{m} \mathit{Term}$$

where

$$\mathit{inv\text{-}Substitution}_{\mathcal{S}}(\theta) \triangleq \forall v \in \mathbf{dom}\,\theta \cdot \theta(v) \neq \mathbf{mk\text{-}}\mathit{Var}(v)$$

The invariant ensures a unique representation for substitutions, a property that is usually desirable in abstract type specifications. The operations can then be specified by

$$\mathit{empty}_{\mathcal{S}} = \{\,\}$$

$$\_ \bullet_{\mathcal{S}} \_ : \mathit{Term} \times \mathit{Substitution} \to \mathit{Term}$$

$$\begin{array}{lll} t \bullet_{\mathcal{S}} \theta & \triangleq & \mathbf{cases}\ t\ \mathbf{of} \\ & & \quad \mathit{mk\text{-}Var}(v) \to \mathbf{if}\ v \in \mathbf{dom}\,\theta\ \mathbf{then}\ \theta(v)\ \mathbf{else}\ t \\ & & \mathit{mk\text{-}CT}(f, \mathit{args}) \to \mathit{mk\text{-}CT}(f, \{i \mapsto \mathit{args}(i) \bullet_{\mathcal{S}} \theta \mid i \in \mathbf{inds}\ \mathit{args}\}) \\ & & \mathbf{end} \end{array}$$

$$\begin{array}{l} \_ \circ \_\ (\theta_1\colon \mathit{Substitution}, \theta_2\colon \mathit{Substitution})\ \theta\colon \mathit{Substitution} \\ \mathbf{post}\ \forall t \in \mathit{Term} \cdot t \bullet_{\mathcal{S}} \theta = t \bullet_{\mathcal{S}} \theta_1 \bullet_{\mathcal{S}} \theta_2 \end{array}$$

$$\mathit{update}_{\mathcal{S}} : \mathit{Variable} \times \mathit{Term} \times \mathit{Substitution} \to \mathit{Substitution}$$

$$\mathit{update}_{\mathcal{S}}(v, t, \theta) \triangleq \theta \circ \{v \mapsto t \bullet_{\mathcal{S}} \theta\}$$

$$\mathbf{pre}\ v \notin \mathbf{dom}\,\theta \wedge v \notin \mathit{vars}(t \bullet_{\mathcal{S}} \theta)$$

The composition operator $\_ \circ \_$ does not appear in the signature. It is defined only to assist in the definition of $update_{\mathcal{S}}$ and so is not subscripted. The strange definition of the function $update_{\mathcal{S}}$ reflects the needs of the unification algorithm, and is is partial in a way which means that substitutions are always idempotent. As a consequence, not all values of the type will be constructible. This is unusual (and in some ways undesirable) but will cause no particular problems in this development. The invariant needed to restrict $Substitution_{\mathcal{S}}$ to idempotent substitutions just adds complexity to the definitions.

If we can find some equations that the operations satisfy and determine which terms should be undefined, we shall have the basis for deriving a range of implementations. It turns out that if the precondition is taken to characterize undefinedness exactly, $update_{\mathcal{S}}$ is commutative.

$$update_{\mathcal{S}}(v_1, t_1, update_{\mathcal{S}}(v_2, t_2, \theta)) = update_{\mathcal{S}}(v_2, t_2, update_{\mathcal{S}}(v_1, t_1, \theta))$$

To prove this, it helps to name the intermediate results of the two orders of updating.

$$\theta' = update_{\mathcal{S}}(v_1, t_1, \theta) = \theta \circ \{v_1 \mapsto t_1 \bullet_{\mathcal{S}} \theta\}$$
$$\theta'' = update_{\mathcal{S}}(v_2, t_2, \theta) = \theta \circ \{v_2 \mapsto t_2 \bullet_{\mathcal{S}} \theta\}$$

We can then establish some useful lemmas characterizing the domain of the operation

Lemma 1.

$$\textbf{pre-}update_{\mathcal{S}}(v_1, t_1, \theta) \wedge \textbf{pre-}update_{\mathcal{S}}(v_2, t_2, \theta') \;\Rightarrow\; v_2 \notin \textbf{dom}\,\theta \wedge v_1 \neq v_2$$

Proof. *Obvious.*

Lemma 2.

$$v_1 \neq v_2 \wedge \textbf{pre-}update_{\mathcal{S}}(v_1, t_1, \theta) \wedge \textbf{pre-}update_{\mathcal{S}}(v_2, t_2, \theta')$$
$$\Rightarrow\; v_2 \notin vars(t_2 \bullet_{\mathcal{S}} \theta)$$

Proof. *If $v_2 \in vars(t_2 \bullet_{\mathcal{S}} \theta)$, then it will also be in $vars(t_2 \bullet_{\mathcal{S}} \theta \bullet_{\mathcal{S}} \{v_1 \mapsto t_1 \bullet_{\mathcal{S}} \theta\})$. But this is contrary to the second precondition.*

Lemma 3.

$$v_1 \neq v_2 \wedge \textbf{pre-}update_{\mathcal{S}}(v_1, t_1, \theta) \wedge \textbf{pre-}update_{\mathcal{S}}(v_2, t_2, \theta') \;\Rightarrow$$
$$v_1 \notin vars(t_2 \bullet_{\mathcal{S}} \theta) \vee v_2 \notin vars(t_1 \bullet_{\mathcal{S}} \theta)$$

Proof. *If both $v_1 \in vars(t_2 \bullet_{\mathcal{S}} \theta)$ and $v_2 \in vars(t_1 \bullet_{\mathcal{S}} \theta)$ then the first substitution will leave occurrences of v_1 which will be replaced by terms containing v_2 by the second substitution. But this is contrary to the second precondition, which requires*

$$v_2 \notin vars(t_2 \bullet_{\mathcal{S}} \theta') = vars(t_2 \bullet_{\mathcal{S}} \theta \bullet_{\mathcal{S}} \{v_1 \mapsto t_1 \bullet_{\mathcal{S}} \theta\})\ .$$

Lemma 4.

$$v_1 \neq v_2 \wedge (v_1 \notin vars(t_2 \bullet_{\mathcal{S}} \theta) \vee v_2 \notin vars(t_1 \bullet_{\mathcal{S}} \theta)) \wedge$$
$$v_2 \notin \mathbf{dom}\,\theta \wedge v_2 \notin vars(t_2 \bullet \theta) \;\Rightarrow$$
$$\quad \mathbf{pre}\text{-}update_{\mathcal{S}}(v_2, t_2, \theta')$$

Proof. *It is obvious that v_2 is not in the domain of θ', so the first conjunct of* $\mathbf{pre}\text{-}update_{\mathcal{S}}(v_2, t_2, \theta')$ *is satisfied. If $v_1 \notin vars(t_2 \bullet_{\mathcal{S}} \theta)$, then*

$$t_2 \bullet_{\mathcal{S}} \theta \bullet_{\mathcal{S}} \{v_1 \mapsto t_1 \bullet_{\mathcal{S}} \theta\} = t_2 \bullet_{\mathcal{S}} \theta$$

and so $v_2 \notin vars(t_2 \bullet_{\mathcal{S}} \theta')$. If $v_2 \notin vars(t_1 \bullet_{\mathcal{S}} \theta))$, we observe that

$$vars(t_2 \bullet_{\mathcal{S}} \theta \bullet_{\mathcal{S}} \{v_1 \mapsto t_1 \bullet_{\mathcal{S}} \theta\}) \subseteq vars(t_2 \bullet_{\mathcal{S}} \theta) \cup vars(t_1 \bullet_{\mathcal{S}} \theta)$$

so again $v_2 \notin vars(t_2 \bullet_{\mathcal{S}} \theta')$. As a consequence, the second part of the precondition is also satisfied.

We can then establish that if one term is defined, the other is

Theorem 1.

$$\mathbf{pre}\text{-}update_{\mathcal{S}}(v_1, t_1, \theta) \wedge \mathbf{pre}\text{-}update_{\mathcal{S}}(v_2, t_2, \theta') \;\Leftrightarrow$$
$$\mathbf{pre}\text{-}update_{\mathcal{S}}(v_2, t_2, \theta) \wedge \mathbf{pre}\text{-}update_{\mathcal{S}}(v_1, t_1, \theta'') \;\Leftrightarrow$$
$$v_1 \neq v_2 \wedge (v_1 \notin vars(t_2 \bullet_{\mathcal{S}} \theta) \vee v_2 \notin vars(t_1 \bullet_{\mathcal{S}} \theta)) \wedge v_1 \notin \mathbf{dom}\,\theta \wedge v_2 \notin \mathbf{dom}\,\theta \wedge$$
$$\quad v_1 \notin vars(t_1 \bullet \theta) \wedge v_2 \notin vars(t_2 \bullet \theta)$$

Proof. *The final formula is a conjunct of the first precondition and properties which have been shown equivalent to the second precondition in the lemmas. It is symmetric in its use of v_1 and v_2, so it is equivalent to the second formula too.*

We can complete the proof of commutativity by showing that when defined the terms are equal. The proof is by considering the values of the two substitutions at all variables in their common domain.

Theorem 2.

$$\mathbf{pre}\text{-}update_{\mathcal{S}}(v_1, t_1, \theta) \wedge \mathbf{pre}\text{-}update_{\mathcal{S}}(v_2, t_2, \theta') \;\Rightarrow$$
$$\quad (update_{\mathcal{S}}(v_2, t_2, \theta')(v)) = (update_{\mathcal{S}}(v_1, t_1, \theta'')(v))$$

Proof.

We have

$$\begin{aligned} update_{\mathcal{S}}(v_2, t_2, \theta') &= \theta' \circ \{v_2 \mapsto t_2 \bullet_{\mathcal{S}} \theta'\} \\ &= \theta \circ \{v_1 \mapsto t_1 \bullet_{\mathcal{S}} \theta\} \circ \{v_2 \mapsto t_2 \bullet_{\mathcal{S}} \theta \bullet_{\mathcal{S}} \{v_1 \mapsto t_1 \bullet_{\mathcal{S}} \theta\}\} \\ &= \theta \circ \{v_1 \mapsto t_1 \bullet_{\mathcal{S}} \theta \bullet_{\mathcal{S}} \{v_2 \mapsto t_2 \bullet_{\mathcal{S}} \theta \bullet_{\mathcal{S}} \{v_1 \mapsto t_1 \bullet_{\mathcal{S}} \theta\}\}, \\ &\qquad v_2 \mapsto t_2 \bullet_{\mathcal{S}} \theta \bullet_{\mathcal{S}} \{v_1 \mapsto t_1 \bullet_{\mathcal{S}} \theta\}\} \end{aligned}$$

Similarly

$$\begin{aligned} update_{\mathcal{S}}(v_1, t_1, \theta'') = \theta \circ \{&v_2 \mapsto t_2 \bullet_{\mathcal{S}} \theta \bullet_{\mathcal{S}} \{v_1 \mapsto t_1 \bullet_{\mathcal{S}} \theta \bullet_{\mathcal{S}} \{v_2 \mapsto t_2 \bullet_{\mathcal{S}} \theta\}\}, \\ &v_1 \mapsto t_1 \bullet_{\mathcal{S}} \theta \bullet_{\mathcal{S}} \{v_2 \mapsto t_2 \bullet_{\mathcal{S}} \theta\}\} \end{aligned}$$

It is enough, then, to show that the two substitutions

$$\begin{aligned}\theta_1 = \{&v_1 \mapsto t_1 \bullet_{\mathcal{S}} \theta \bullet_{\mathcal{S}} \{v_2 \mapsto t_2 \bullet_{\mathcal{S}} \theta \bullet_{\mathcal{S}} \{v_1 \mapsto t_1 \bullet_{\mathcal{S}} \theta\}\},\\ &v_2 \mapsto t_2 \bullet_{\mathcal{S}} \theta \bullet_{\mathcal{S}} \{v_1 \mapsto t_1 \bullet_{\mathcal{S}} \theta\}\}\\ \theta_2 = \{&v_2 \mapsto t_2 \bullet_{\mathcal{S}} \theta \bullet_{\mathcal{S}} \{v_1 \mapsto t_1 \bullet_{\mathcal{S}} \theta \bullet_{\mathcal{S}} \{v_2 \mapsto t_2 \bullet_{\mathcal{S}} \theta\}\},\\ &v_1 \mapsto t_1 \bullet_{\mathcal{S}} \theta \bullet_{\mathcal{S}} \{v_2 \mapsto t_2 \bullet_{\mathcal{S}} \theta\}\}\end{aligned}$$

are equal when the updates are defined, which can be done by considering their values at v_1 and v_2. We have

$$\theta_1(v_1) = t_1 \bullet_{\mathcal{S}} \theta \bullet_{\mathcal{S}} \{v_2 \mapsto t_2 \bullet_{\mathcal{S}} \theta \bullet_{\mathcal{S}} \{v_1 \mapsto t_1 \bullet_{\mathcal{S}} \theta\}\}$$

Since the preconditions hold, $v_1 \notin vars(t_2 \bullet \theta)$ or $v_1 \notin vars(t_2 \bullet \theta)$. If $v_1 \notin vars(t_2 \bullet \theta)$, then

$$t_1 \bullet_{\mathcal{S}} \theta \bullet_{\mathcal{S}} \{v_2 \mapsto t_2 \bullet_{\mathcal{S}} \theta \bullet_{\mathcal{S}} \{v_1 \mapsto t_1 \bullet_{\mathcal{S}} \theta\}\} = t_1 \bullet_{\mathcal{S}} \theta \bullet_{\mathcal{S}} \{v_2 \mapsto t_2 \bullet_{\mathcal{S}} \theta\}$$

because the second substitution makes no difference. Similarly, if $v_1 \notin vars(t_2 \bullet \theta)$ then

$$t_1 \bullet_{\mathcal{S}} \theta \bullet_{\mathcal{S}} \{v_2 \mapsto t_2 \bullet_{\mathcal{S}} \theta \bullet_{\mathcal{S}} \{v_1 \mapsto t_1 \bullet_{\mathcal{S}} \theta\}\} = t_1 \bullet_{\mathcal{S}} \theta \bullet_{\mathcal{S}} \{v_2 \mapsto t_2 \bullet_{\mathcal{S}} \theta\}$$

still holds because the first substitution has no effect, and so its form is immaterial. In either case, we have $\theta_1(v_1) = \theta_2(v_1)$. The argument for $\theta_1(v_2) = \theta_2(v_2)$ is the same, with the rôles of v_1 and v_2 reversed.

The quotient model satisfying this equation groups all terms which have the same arguments to the series of *update*s, irrespective of order (since any ordering can be transformed into any other). This is isomorphic to the bag of (v_i, t_i) pairs.

Partial implementations turn out to be more interesting. Theorem 1 shows that we cannot bind the same variable more than once. In conjunction with commutativity, this can be defined by

$$U(update(v, t_1, update(v, t_2, \theta)))$$

The effect is to remove from the quotient model all sets containing terms with repeated variables. The remaining sets associate at most one term with any given variable, and their terms exhibit the bindings in all possible orders. They are isomorphic to partial functions from variables to terms. This is the same model as the specification, but the operations are defined differently. In the relative term algebra, updating θ by binding v to t gives $update(v, t, \theta)$. In the quotient model, the equivalence class of terms with the same bindings is mapped to the class including the new binding: in the isomorphic partial function model this corresponds naturally to adding a new pair to the function. The term *empty* is equivalent only to itself, and can be made to correspond to $\{\}$

$$empty_{\mathcal{I}} = \{\}$$

$$update_{\mathcal{I}}(v, t, \theta) \quad \triangleq \quad \theta \cup \{v \mapsto t\}$$

The $update_{\mathcal{I}}$ function is undefined if $v \in \mathbf{dom}\,\theta$: in the quotient algebra this corresponds to an operation with a deleted set as a result. We could make the domain explicit by writing

$$\mathbf{pre}\text{-}update_{\mathcal{I}}(v, t, \theta) \quad \underline{\triangle} \quad v \notin \mathbf{dom}\,\theta$$

To arrive at the definition for substitution application based on this new representation of substitutions, we use the reification condition as usual.

$$t \bullet_{\mathcal{I}} \theta = t \bullet_{\mathcal{S}} \phi(\theta)$$

(Recall that this is now a strong equality.) We need to find a definition of $\bullet_{\mathcal{I}}$ with this property. Considering cases, applying the reification condition and unfolding gives

$$\begin{aligned}\mathbf{mk}\text{-}CT(f, args) \bullet_{\mathcal{I}} \theta &= \mathbf{mk}\text{-}CT(f, args) \bullet_{\mathcal{S}} \phi(\theta) \\ &= \mathbf{mk}\text{-}CT(f, \{i \mapsto args(i) \bullet_{\mathcal{S}} \phi(\theta) \mid i \in \mathbf{inds}\ args\}) \\ &= \mathbf{mk}\text{-}CT(f, \{i \mapsto args(i) \bullet_{\mathcal{I}} \theta \mid i \in \mathbf{inds}\ args\})\end{aligned}$$

$$\begin{aligned}\mathbf{mk}\text{-}Var(v) \bullet_{\mathcal{I}} \theta &= \mathbf{mk}\text{-}Var(v) \bullet_{\mathcal{S}} \phi(\theta) \\ &= \mathbf{if}\ v \in \mathbf{dom}\,(\phi(\theta))\ \mathbf{then}\ (\phi(\theta))(v)\ \mathbf{else}\ \mathbf{mk}\text{-}Var(v)\end{aligned}$$

The first equation immediately suggests a definition of $\bullet_{\mathcal{I}}$ for compound terms, but the second needs work to remove the uses of ϕ. That in the condition can be removed using

Theorem 3. $\delta(\phi(\theta)) \ \Rightarrow\ \mathbf{dom}\,\theta = \mathbf{dom}\,(\phi(\theta))$

Proof. *By induction over θ. The induction principle for maps given in Jones (1990) is essentially*

$$\frac{P(\{\}) \quad x \notin \mathbf{dom}\,m; P(m) \vdash P(\{x \mapsto y\} \cup m)}{\forall m \cdot P(m)}$$

The base case is established by

$$\begin{aligned}\mathbf{dom}\,(\phi(\{\})) &= \mathbf{dom}\,(\phi(empty_{\mathcal{I}})) \\ &= \mathbf{dom}\,(empty_{\mathcal{S}}) \\ &= \mathbf{dom}\,\{\}\end{aligned}$$

while the inductive case is proved by

$$\begin{aligned}\mathbf{dom}\,(\phi(\{v \mapsto t\} \cup \theta)) &= \mathbf{dom}\,(\phi(update_{\mathcal{I}}(v, t, \theta))) \\ &= \mathbf{dom}\,(update_{\mathcal{S}}(v, t, \phi(\theta))) \\ &= \mathbf{dom}\,(\phi(\theta) \circ \{v \mapsto t \bullet_{\mathcal{S}} \phi(\theta)\}) \\ &= \mathbf{dom}\,(\phi(\theta)) \cup \{v\} \\ &= \mathbf{dom}\,\theta \cup \{v\} \\ &= \mathbf{dom}\,(\{v \mapsto t\} \cup \theta)\end{aligned}$$

The proof strategy is to rewrite the values as applications of constructors and use the reification condition and specification definitions to simplify just like the synthesis of $elem_{\mathcal{I}}$ in Sect. 2. It depends heavily on the stronger reification condition established in Sect. 3.

To simplify the expression in the **then** arm, observe that if v is in the domain of θ, it was put there by an *update*, and since the order of updates is unimportant, there is some θ' and t such that $\theta = update_{\mathcal{I}}(v, t, \theta')$. We may then show

Theorem 4.

$$\delta(\phi(update_{\mathcal{I}}(v,t,\theta))) \Rightarrow \\ \phi(update_{\mathcal{I}}(v,t,\theta))(v) = (update_{\mathcal{I}}(v,t,\theta))(v) \bullet_{\mathcal{I}} (update_{\mathcal{I}}(v,t,\theta))$$

Proof.
Applying the stronger reification condition and unfolding, we have

$$\begin{aligned} \phi(update_{\mathcal{I}}(v,t,\theta))(v) &= update_{\mathcal{S}}(v,t,\phi(\theta))(v) \\ &= (\phi(\theta) \circ \{v \mapsto t \bullet_{\mathcal{S}} \phi(\theta)\})(v) \\ &= (\phi(\theta))(v) \bullet_{\mathcal{S}} \{v \mapsto t \bullet_{\mathcal{S}} \phi(\theta)\} \\ &= t \bullet_{\mathcal{S}} \phi(\theta) \end{aligned}$$

If $\phi(update_{\mathcal{I}}(v,t,\theta))$ is defined, we must have **pre**-$update_{\mathcal{S}}(v,t,\phi(\theta))$, *and so $v \notin vars(t \bullet_{\mathcal{S}} \phi(\theta))$. Hence*

$$\begin{aligned} t \bullet_{\mathcal{S}} \phi(\theta) &= t \bullet_{\mathcal{S}} \phi(\theta) \bullet_{\mathcal{S}} \{v \mapsto t \bullet_{\mathcal{S}} \phi(\theta)\} \\ &= t \bullet_{\mathcal{S}} update_{\mathcal{S}}(v,t,\phi(\theta)) \\ &= (update_{\mathcal{I}}(v,t,\theta))(v) \bullet_{\mathcal{I}} (update_{\mathcal{I}}(v,t,\theta)) \end{aligned}$$

Now the body of the variable case reduces to

$$\textbf{if } v \in \textbf{dom}\,\theta \textbf{ then } (\theta(v)) \bullet_{\mathcal{I}} \theta \textbf{ else } t$$

and there is a convenient definition with these properties to use as an implementation

$$\begin{aligned} t \bullet_{\mathcal{I}} \theta \;\; \triangleq \;\; & \textbf{cases } t \textbf{ of} \\ & \quad mk\text{-}Var(v) \rightarrow \textbf{if } v \in \textbf{dom}\,\theta \textbf{ then } (\theta(v)) \bullet_{\mathcal{I}} \theta \textbf{ else } t \\ & mk\text{-}CT(f, args) \rightarrow mk\text{-}CT(f, \{i \mapsto args(i) \bullet_{\mathcal{I}} \theta \mid i \in \textbf{inds } args\}) \\ & \textbf{end} \end{aligned}$$

5 Comparison

A more conventional development, positing implementation definitions and an abstraction function and then showing that the reification condition used here holds, is presented in Clement (1994) as a contrast with the usual VDM approach to reification: the relationship between the two is explored in more detail there. Here we want to compare that posit-and-prove development with the calculational development given here.

The most important practical consideration is the difficulty of constructing the proofs needed to justify the implementations. In the posit-and-prove approach, establishing that the reification condition holds involves a large amount of detailed reasoning using our intuition for how the implementation and the abstraction function work. In this particular development, the abstraction function is an iterated composition of the representation, and the arguments are based on how many iterations will be needed to guarantee that further iterations produce no change. Because the definitions involve substitutions, properties of the substitution operations are used widely throughout the proofs: these properties are taken as obvious in Clement (1994) but could be stated explicitly as lemmas and even proved from the specification.

In contrast, the approach using calculation presented here begins by establishing properties of the specification, independent of any implementation. The proof of commutativity is quite long, but only because the expanded formulae are quite large: the actual reasoning is exclusively concerned with the same kind of properties of substitutions that were used in the posit-and-prove approach (and which are again assumed). It was certainly easier to develop. The result of the proof is a theorem about the specification, which is potentially useful to definitions making use of the abstract type. This cannot be said of the reification condition proofs of the posit-and-prove approach (although useful lemmas may appear). The undefinedness property follows from one of the lemmas of the commutativity proof.

Once the properties have been determined, the term model has to be constructed and an isomorphic algebra found. This process has been presented informally, but the steps are formally defined and it would at least be possible to present a function connecting values in the quotient algebra with those in the implementation and show that it is an isomorphism. However, commutativity is well known to define bags and the effect of the undefinedness is reasonably clear so the less formal development here should be sufficiently convincing. In general, it seems more productive to exploit established results rather than prove everything ourselves. The final step is the synthesis of the observations: the proof here is very similar to that of the posit-and-prove development although it expresses a calculation rather than justifying a posited definition.

In practical terms, then, the calculational approach seems to have an advantage. This stems from avoiding an explicit definition of an abstraction function, since if one were to be posited as in the usual calculi, the calculation of the implementation of $update_{\mathcal{I}}$ would once again involve the details of how the implementation works.

A more philosphical question is to compare the amount and nature of the inspiration that was required to arrive at the implementation. This is, after all, what the calculational approach is meant to reduce. In the posit-and-prove approach, it leads to the posited implementation and abstraction function. In this case, the first informal idea is that recording the arguments to *update* will certainly be enough. Applying a substitution then means looking up each variable, and substituting the resulting term in the same way using the rest of the bind-

ings. (This is already independent of the order in which the bindings are made.) A simplification of this process can be justified by observing that substituting using the whole substitution makes no difference, because the precondition means that the term cannot contain the variable it is bound to. It is this that suggests the definition of the abstraction function as an n-fold iteration.

The inspiration in the calculational approach is in the choice of properties. The undefinedness is an obvious consequence of the precondition (although its usefulness is less obvious at the start). Given that we need to be able to identify the implementation from the quotient algebra, it helps to restrict attention to simple algebraic properties like commutativity, but that still leaves a range of possibilities, some of which may be easier to discount than others. This is typical of calculational approaches: they define a spectrum of possible ways to proceed but some intuition for the expected final result of the calculation is necessary to decide which way to go. In this case, the commutativity might be suggested by the unification algorithm which motivates these definitions: it generates a series of updates, but could do so in any order. (It has to be said that in practice (Manna & Waldinger (1981); Clement (1991)) the algorithm has been derived for a single order of updates rather than making this observation.) Failing that, the intuition for the posit-and-prove approach sketched above also tells us that the implementation of *update* is commutative and thus its specification should be, so at least the intuition needed for calculation is no harder to come by than that for posit-and-prove, even if there is limited evidence that it is easier.

On balance, then, the calculational approach seems to have advantages over posit-and-prove when it can be applied. In principle, it always can be, since any desired implementation can be given an algebraic definition, and it can then be confirmed that the properties hold of the specification. (These properties can be expressed in any language which allows us to construct objects with guaranteed morphisms to all others: a number are described in Goguen & Burstall (1992).) In practice, these algebraic definitions may need operations in their signatures which were not in the original specification. They could be added to the specification to allow their properties to be checked, but this extra work makes the approach look less attractive. It would be worth investigating examples of this kind, but for the moment the method looks more appropriate to cases where the implementations involve types with simple algebraic properties (although as we have seen the specifications can be quite complex).

6 Summary

We have presented a way of constructing implementations from specifications. They satisfy the VDM criterion that the implementation behaves like the specification whenever the specification is defined, but unlike the usual VDM approach do not require the explicit statement of an abstraction function. Instead, the approach makes use of properties of the specification, drawing heavily on the constructive aspects of the theory of algebraic specification. In the special case where operations and abstraction function are total, all the results presented are well

known in the algebraic setting, and only the application to model based specification is novel. Partial algebras have also received some previous attention. The robust morphism introduced in Broy (1985) corresponds to the reification condition of Sect. 1 and the stronger condition satisfied by the calculated abstraction function corresponds to the weak morphisms of Broy & Wirsing (1982). (It is an accident of nomenclature that the weak morphism is stronger than the robust one!) However, their particular interest was in the identification of morphisms leading to algebras with initial or terminal algebras for specification rather than any application to implementation. (Both weak and robust morphisms give rise to categories with terminal algebras.) The morphisms with initial algebras that they define have the empty algebra as initial model in the absence of axioms saying that particular terms must be defined: this is in contrast to our approach where the quotient algebras are total unless terms are explicitly declared undefined. By applying algebraic techniques in a model based setting we have avoided the principal problem of algebraic specification, which is the need to give enough axioms to characterize the specificand uniquely: all we need here are enough properties to characterize the wanted implementation.

Another approach to the calculation rather than positing of abstraction functions is the *SETS* calculus of Oliveira (1992). There, the emphasis is on deriving abstraction functions (and invariants on the implementation) for structured types given abstraction functions and invariants for the component types. There seems to be scope for combining the two methods to construct complex specifications without writing down abstraction functions: it should be possible to combine their properties in much the same way that the functions themselves are combined in the *SETS* calculus. However, technical differences in the details of the reification conditions used would have to be resolved.

It is worth saying that none of the approaches to data reification is mechanical: each has some elements of intuition guiding the choice of definitions or properties and if the intuition is wrong then the formal development will fail to go through. In the case of posit-and-prove, it will be some reification condition that will fail to hold, and in the usual calculational approach, the defining property of an observation based on the reification condition may not lead to a definition with no use of ϕ. In this approach, a given set of properties may not lead to interesting models or convenient definitions of the operations.

How relevant is this work to industrial practice? Categories are thought of as abstract concepts even by mathematicians, and the computer industry does not usually rush to take up abstract mathematics. This is a mistake if the goal is to deliver more reliable software at lower cost. Reliability requires some kind of proof, and proof is expensive and becomes more so as proofs become larger. For this reason, the proofs in this development and in that described in Clement (1994) are rigorous rather than formal: only a top-level view of the proof is provided using phrases such as "by induction", and the reader is left to fill in the details. Not only are the proofs here shorter, but because they apply abstract ideas, some of the gaps can be filled in by reference to the standard literature

rather than by detailed reasoning in the specific area of the application. They should thus be more convincing as well as less expensive.

References

M. A. Arbib and E. G. Manes. *Arrows, Structures, and Functors: The Categorical Imperative*. Academic Press, 1975.

R. S. Boyer and J. S. Moore. The sharing of structure in theorem-proving programs. In *Machine Intelligence 7*, pages 101–116. Edinburgh University Press, 1972.

M. Broy. Extensional behaviour of concurrent, nondeterministic, communicating systems. In *Control Flow and Data Flow: Concepts of Distributed Programming*, pages 229–276. Springer-Verlag, 1985.

M. Broy and M. Wirsing. Partial abstract types. *Acta Informatica*, 18:47–64, 1982.

T. Clement. Combining transformation and posit-and-prove in a VDM development. In *VDM'91: Formal Software Development Methods*, pages 63–92. Springer Verlag, 1991.

T. Clement. Notes on data reification. In *FME'93 tutorial material*, pages 151–190, 1993.

T. Clement. Comparing approaches to data reification. In *FME'94: Industrial Benefits of Formal Methods (LNCS 873)*, pages 118–133. Springer Verlag, 1994.

J. Darlington. The design of efficient data representations. In *Automatic Program Construction Techniques*, chapter 7, pages 139–156. Macmillan, 1984.

H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification*, volume 6 of *EATCS Monographs*. Springer-Verlag, 1985.

J. A. Goguen and R. M. Burstall. Institutions: Abstract model theory for specification and programming. *Journal of the ACM*, pages 95–146, 1992.

J. A. Goguen, J. W. Thatcher, and E. G. Wagner. An initial algebra approach to the specification, correctness and implementation of abstract data types. In *Current Trends in Programming Methodology*, volume 4, 1978.

International Standards Organisation. *Information Technology Programming Languages – VDM-SL First Committee Draft Standard CD 13817-1*, November 1993. Document Number ISO/IEC JTC1/SC22/WG19/N-20.

C. B. Jones. *Systematic Software Development Using VDM*. Prentice-Hall International, 2nd edition, 1990.

C. C. Morgan and P. H. B. Gardiner. Data refinement by calculation. *Acta Informatica*, 27:481–503, 1990.

Z. Manna and R. Waldinger. Deductive synthesis of the unification algorithm. *Science of Computer Programming*, 1:5–48, 1981.

J. N. Oliveira. Software reification using the SETS calculus. In *Proceedings of the 5th Refinement Workshop*, Workshops in Computing, pages 140–171. Springer-Verlag, 1992.