The Role of Data Reification in Program Refinement: Origins, Synthesis and Appraisal

T. Clement
Department of Computer Science, University of Manchester, Manchester M13 9PL

The formal development of implementations from specifications requires that we should be able to justify the replacement of definitions which are clear but inefficient by those which are efficient but unclear. We look at ways to provide such justifications, and see that they depend on the form of the definitions and the ways in which we propose to exploit them. We develop conditions which are feasible to establish and meet the needs of the most common situations, and provide pointers to related work in this area.

Some familiarity with set theory will help with the formal definitions, although these can be skipped without loss of continuity.

Received June 1992

I. A MOTIVATING EXAMPLE

The process of software development begins with the requirements for the system to be developed. For example, imagine a system for advising on interchangeable parts. This may be required to have an operation for asserting that two parts are interchangeable, and another for determining if they are. Two parts will be interchangeable if they have been explicitly asserted to be so, or if each is interchangeable with some third part. The requirements must also say which parts are to be interchangeable initially.

If a formal development methodology is being used, these requirements will then be expressed formally in a specification. The first step is to describe the kinds of information (referred to as sorts below) that the operations act on and produce. This is the signature of the system. Thus the assert operation will take two parts and the state of the system and yield a new state, the test operation will take two parts and a state and yield a Boolean (true or false) result, and the initialisation will yield a state. To formalise this we invent names for the sorts and operations, and write

\[
\text{init} : \rightarrow \text{InterchangeableParts}
\]

\[
\text{test} : \text{Part} \times \text{Part} \times \text{InterchangeableParts} \rightarrow \mathbb{B}
\]

\[
\text{assert} : \text{Part} \times \text{Part} \times \text{InterchangeableParts} \rightarrow \text{InterchangeableParts}
\]

separating results from parameters by \(\rightarrow\), and individual parameters (and results) by \(\times\). In general, a signature defines a set of sorts \(\mathcal{S}\), and a (finite) map \(\Omega\) from operation names to their parameter and result sorts. We shall want operations to return values of more than one sort (\(\text{pop}\) on a stack, for example, returns an element and a new stack), so we shall write elements of \(\Omega\) as \(\text{op} : W_1 \rightarrow W_2\), where \(W_1\) and \(W_2\) are sequences of sorts.

We then define the operations. One way to represent the state of the system is as a set of sets of parts. Each part in a set is interchangeable with any other part in that set. When two parts are asserted to be interchangeable, we merge their sets together (and the other sets remain unchanged). Initially, no parts are interchangeable (except with themselves), so each is in its own set. We can formalise this in the definitions

\[
\text{InterchangeableParts}^0 = (\text{Part} \rightarrow \text{set}) - \text{set}
\]

\[
\text{init}^0() \triangleq \{(p)| p \in \text{Part}\}
\]

\[
\text{test}^0(p_1,p_2,\text{ss}) \triangleq \exists s \in \text{ss} \cdot \{p_1,p_2\} \subseteq s
\]

\[
\text{assert}^0(p_1,p_2,\text{ss}) \triangleq \{s | s \in \text{ss} \land (p_1 \in s \lor p_2 \in s)\} \cup \{s | s \in \text{ss} \land \neg (p_1 \in s \lor p_2 \in s)\}
\]

(We shall use superscripts to distinguish different definitions of objects from the same signature.) The formal version of assert looks more complex than it really is. The first set of sets contains a single set, which is the result of merging the sets containing the two parts into one, while the second set of sets is all the others from the original state, just as in the informal definition.

The formal description gives a clear and precise picture of what the operations are supposed to do, but viewed as a program it has two defects. First, the data types it manipulates are sets, and sets and their operations are not primitive data types in most languages. Second, even if the operations were available, the resulting program would be inefficient because both test and assert involve operations on the entire state. In other cases (when the definition tests all members of an infinite set, perhaps) the description may not be interpretable as a program at all.

An alternative approach to representing the state is to have an array indexed by and containing part identifiers. We begin with each element of the array containing itself.

To assert that two parts are equal, we follow the chains from each part and see if they end up in the same place. We then make one of these point to the other. (The effect is to build up a forest of trees of interchangeable parts, each one represented by links from each child to its parent.) Testing them just means following the chains from each part and seeing if they end up in the same place. Formally, we represent the array as a map, and the updating (in assert\(^*\)) as the overriding (\(\triangleright\)) of one map by another defined at the one point where we want to change it.

\[
\text{InterchangeableParts}^* = \text{Part} \rightarrow \text{Part}
\]

\[
\text{init}^*() \triangleq (p \rightarrow p) | p \in \text{Part}
\]

\[
\text{test}^* (p_1,p_2,\text{mpp}) \triangleq \text{root}(p_1,\text{mpp}) = \text{root}(p_2,\text{mpp})
\]
assert(e(p1, p2, mpp)) ≜
mpp ↑ {root(p1, mpp) → root(p2, mpp)}
root : Part × InterchangeableParts ↝ Part
root(p, mpp) ≜
if mpp(p) = p then p else root(mpp(p))

(Subsidiary functions such as root, which do not appear in the signature, are invisible to the user.) The work in both assert and test is in finding the ends of the chains from each part, which should be reasonably efficient (depending on the number of iterations needed to reach an array element containing its own index). It would be a suitable implementation of the requirements if we could be sure that it did in fact meet them.

One way to show that it does is to prove that it does the same thing as the first definition, in the sense that the user who initialises the system, does some sequence of assert operations and then a test operation will always find that the same parts are interchangeable. This may be difficult, but because proofs can be checked mechanically for their validity, we can be confident that it is correct. We can be as sure that the obscure but efficient definition meets the requirements as we are that the clear but inefficient one does. In that case, we say that the implementation reifies the specification: we have implemented one data type in terms of another and altered the operations to suit.

In the rest of this paper, we shall look at some approaches to showing that one definition reifies another. In the next section, we shall see how this can be done for specifications like those above, which define a unique result for each choice of inputs. We shall see that we need to make precise what we mean by one definition doing the same thing as another. There are many plausible choices here. We shall use the one which is most applicable to the functional correctness of sequential programs: some of the work that has been done to meet other requirements will be indicated in Section 7. In Section 3, we shall extend this to cases where operations are not always defined, and in Section 5 we shall look at the reification of loose specifications, where operations can have a range of possible results. Section 4 goes beyond defining an implementation as anything that behaves in the same way to capture an idea of specifications being closer to the requirements while implementations are closer to the machine, and in Section 6 we look at this again for loose specifications. Section 8 summarises the development and provides some practical suggestions for choosing an approach.

2. DEFINING REIFICATION

In this section, we shall make more precise the idea of reification that we sketched above for the example, and look at practical approaches to testing it. We must begin with a few definitions. The terms of a given signature are the things we can write using the operations of the signature and respecting the sorts of the arguments. Thus init and test(p1, p2, assert(p1, p2, init)) (where p1 and p2 are parts) are both terms of the signature of the example. The sort of a term is the result sort of its outermost operation (InterchangeableParts in the first case, and B in the second). We can extend the set of terms by introducing variables. These are treated like constants in forming terms, so p1 and p2 could be taken as variables in the example above. In formal definitions, we shall write terms as t: if we want to emphasise that a term contains a particular variable, we shall write t[x]. Terms which contain no variables are said to be ground.

A definition like those of the example defines sorts as sets (their carriers), and operations as functions (or values for constants). Technically, this is known as an algebra. We can evaluate a term using the definitions by applying the functions to the values of their arguments in the usual way. Variables are given values by an assignment to an element of the appropriate carrier. Thus to evaluate the second term using the first set of definitions, p1 and p2 would be given values from the set Part. We have not defined Part, but that does not matter because we can see that whatever their values, the value of the whole expression will be true. Formally, the value of term t using definitions a will be written t[a].

Now we can make more precise the idea that one definition implements another. The thought experiment of the introduction corresponds to evaluating ground terms using the two definitions and getting the same answer. (The user expects, say, test(p1, p2, assert(p1, p2, assert(p1, p2, init))) to be true in each case.) But not all terms have results that can be observed: for example, the user has no way to see the value of assert(p1, p2, assert(p1, p2, init)). It was this that gave us the freedom to change the definition of InterchangeableParts. In other words, we only expect the values of some terms to be the same, and need to say which. The distinction between these two is that the first has sort B, which had a counterpart in the requirements, while the second has sort InterchangeableParts, which does not. We can formalise the distinction by identifying a subset of visible sorts in the signature (we shall refer to this as V). In the example, these are Part and B. The other sorts (just InterchangeableParts in the example) are called hidden sorts. Our first condition for implementation is that ground terms with visible sorts as results must have the same value in each definition: we shall refer to this as a term condition.

It is not easy to verify that a term condition holds between two definitions, since there is an infinite number of terms. We should prefer an equivalent condition which looks at the sets and functions of the definitions only: this we shall call a reification condition. Informally, we know that definition a in the example behaves in the same way as definition b because there is a correspondence between the two definitions of InterchangeableParts. The nodes of a tree in a value from InterchangeableParts are the elements of a set in the corresponding value from InterchangeableParts. Clearly, corresponding values will agree on which parts are interchangeable, because in one case we check that they are in the same tree and in the other that they are in the same set. The two definitions begin in corresponding states, because init[a] puts each part in its own tree and init[b] puts each part in its own set. They stay in corresponding states because assert[a] merges trees to produce a new tree and assert[b] merges sets. Thus the same parts are interchangeable after any series of assertions using either definition.

This gives a general way to show that one definition reifies another by looking at the definition alone. a reifies b (which we shall write formally as a ≤ b) if we can find a family of correspondences, one for each sort in
their common signature (formally, $\approx$ for each $S \in \mathcal{S}$), where the correspondence on each of the visible sorts is equality, and where corresponding inputs to every operation give corresponding outputs. (Constants are treated as operations with no input.) Operations with visible results must give the same values in each definition because the correspondence is equality. Operations with hidden results preserve the correspondence as in the example. This will hold even when operations have parameters and results of both kinds and there may be more than one hidden sort. Formally, the inputs and results of functions are tuples (from the product of the sets which are the definitions of the sorts of the signature of the operation), so the input to $\text{test}^w$ and $\text{assert}^w$ are tuples $(p_1, p_2, p_{\text{ss}})$ (in $\text{Part} \times \text{Part} \times (\text{Part} \times \mathcal{S}) = \text{set} \text{-set}$), and one tuple corresponds to another (via a relation $\approx_w$, where $W$ is a sequence of sorts) if the first elements correspond, the second elements correspond, and so on. We can then write the reification condition formally as

$$\forall \approx \in \mathcal{A} \quad \exists \{ s \in S^w \leftrightarrow S^d | S \in \mathcal{S} \land (S \in \mathcal{V} \Rightarrow \approx_s \text{ is equality}) \}.$$ 

$$\forall \text{op}: W^1 \rightarrow W^2 \in \Omega, \forall a \in W^1, c \in W^2, \forall \approx_w, \approx^w, \approx^a.$$ 

$$(X \rightarrow Y) \text{ is the set of relations from } X \text{ to } Y.$$ 

We can formalize the correspondence on $\text{InterchangeableParts}$ described informally above as

$$\text{ms} \approx \text{pss} \Rightarrow s \in \text{Pss} = \{ \{ p' \in \text{Part} | \text{root}(p', ms) = \text{root}(p, ms) \} | p \in \text{Part} \}$$

and verify (at some length) that the reification condition is satisfied by the definitions of the introduction and this choice.

The reification condition guarantees that all ground terms have corresponding values in each definition, so ground terms of visible sort have the same values in each definition. It is thus a sound alternative to the term condition. Conversely, we can construct a family of correspondences by relating the value of any terms in one definition to the value of the same term in the other definition. If the definitions are related by the term condition, all ground terms of visible sort have the same values in the two definitions, and the correspondences will verify the reification condition. The reification is thus a complete replacement for the term condition. Soundness and completeness together tell us that the conditions are equivalent.

In general the ‘user’ of the specification will be another (exploiting) definition. (The interchangeable parts system might be used in a much larger specification for a computer-aided manufacturing system, to allow parts to be substituted automatically during the assembly of a product.) The details of the definitions of the hidden sorts will be hidden from the exploiting definition in the way that the state of a system is hidden from its user if it can only use terms of hidden sort in operations from the signature, while values of visible sort can be used in any way the specification language allows. For example, we could write $\text{if test}(p_1, p_2, \text{init}) \ldots$, or even $\text{let } s = \text{assert}(p_1, p_2, \text{init}) \in \text{test}(p_1, p_2, s)$, but not $\text{let } \text{init} = \{ \{ p \} | p \in \text{Part} \} \ldots$. This restriction on the use of a sort is familiar from the abstract data type declarations of programming languages like ML and Modula-2, where we have the same motivation of being able to change the representation of the type without affecting the program which uses it. However, we can see that these terms, unlike those of our user’s experiments, contain variables.

In constructs like $\text{let } x = t \in \{ x \}$, we want $t'[x]$ to have the same value in each definition if it is of visible sort, although if $t$ has hidden sort, $x$ will be given different values in each case. We must extend the term condition to say that terms of visible sort containing variables must have the same values in each definition, when the variables are assigned the values of the same term in each definition. The new reification condition is actually equivalent to this one, too, so we have not in fact said anything new. This is because we can replace the $\text{let }$ construct above by the term $t'$ with every occurrence of $x$ replaced by $t$ to give a ground term.

On the other hand, consider the formula

$$\forall s \in \text{InterchangeablePart} \cdot \text{test}(p_1, p_3, s) \land \text{test}(p_2, p_3, s) \Rightarrow \text{test}(p_1, p_3, s)$$

in the context of the example. It is false in the specification, but true in the implementation. We have two choices: we can say that exploiting definitions should not use quantified formulae (just as we limited the way they used terms of hidden sort), or we can strengthen the term and reification conditions. The problem is that an assignment of values to variables in one algebra does not necessarily have a counterpart in the other. If we insist that it should in the term condition, we can make the reification condition sound again by insisting that all correspondences are total and onto. However, there are no such correspondences between the definitions of the example: the specification allows sets of overlapping sets in $\text{InterchangeableParts}^w$ which have no counterpart in $\text{InterchangeableParts}$.

In this case, we can argue that such values cannot be produced by any ground term (they are junk), and have no place in a proper specification. We shall see, though, that there are junk-free implementations of junk-free specifications which seem intuitively reasonable but where there is still no total and onto correspondence.

The key phrase here is ‘intuitively reasonable’. Ultimately, we are looking for a condition that allows only (and preferably all) such implementations. This is what we are trying to formalise by saying that certain terms have the same values in each definition, and we should be guided in our choice of terms by considering the implementations it allows. We must then live with any restrictions on the ways we can exploit definitions. It is perhaps not surprising, given that our intuition about implementation comes from programming, that we end up allowing the program-like $\text{let }$ and disallowing the less program-like $\forall$.

3. REIFYING PARTIAL DEFINITIONS

There is an issue which did not arise in the example. It is often the case that the requirements do not completely establish the behaviour of an operation. This may be because the requirements are incomplete and need to be extended, or it may be that it is known that the operation will never be used where it is not defined. For example, consider a stack of natural numbers ($\mathbb{N}$), with signature

$\text{init} : \text{Stack} \rightarrow \mathbb{N} \times \text{Stack}$

$\text{push} : \mathbb{N} \times \text{Stack} \rightarrow \text{Stack}$

$\text{pop} : \text{Stack} \rightarrow \mathbb{N} \times \text{Stack}$

$\text{isempty} : \text{Stack} \rightarrow \mathbb{N}$
(where Stack is hidden). A natural way to define this is to make the stacks sequences of numbers, where push adds the new number to the front of the sequence and pop removes it. The initial (empty) stack is the empty sequence. It is not clear what values result from popping the empty stack, so we shall say that it is undefined. Formally, we record the domain of definition as a precondition on the inputs.

\[
Stack^d = \mathbb{N}^* \quad \text{init}^d() \triangleq [\]
\]
\[
\text{push}^d(x, s) \triangleq \text{cons}(x, s)
\]
\[
\text{pop}^d(s) \triangleq (\text{hds}, s, t) s
\]
\[
\text{isempty}^d(s) \triangleq s = [\]
\]

If we make the rule that an operation should not be used in the exploiting definition where it is not defined in the specification, we can modify our term condition to say that only those terms which have values defined by the specification need have the same visible values in an implementation. As a result, an implementation may choose to define functions in cases where the specification does not. We can formalise this by introducing the predicate \( D(\ldots) \), which will be true if the evaluation of a term using a definition involves no function application to arguments not satisfying its precondition. Then

\[
\forall \theta \subseteq \mathcal{A}, D \triangleq
\]
\[
\exists \{ s \in S^d : s = [ \} \wedge (s \in \mathcal{S} \Rightarrow s \text{ is equality})
\]
\[
\forall \theta : W_t \rightarrow W_s \in \Omega, \forall a \in W_t, c \in W_s
\]
\[
\forall \theta : W_t \rightarrow W_s \in \Omega, \forall a \in W_t, c \in W_s
\]
\[
c \approx w, a \wedge D\left(\text{op}^d(a)\right) \Rightarrow
\]
\[
D\left(\text{op}^d(c)\right) \wedge \text{op}^d(c) \approx w, \text{op}^d(a)
\]

When the operations are all total, \( D \) is always true, and the new reification condition reduces to the previous one. The equivalence arguments go through as before, except that when constructing the correspondences we consider only terms \( t \) where \( D \) is defined. We observe that the restriction on exploiting definitions rules out constructs like \( \theta(x) \triangleq \text{if } D(f(x)) \text{ then } f'(x) \text{ else } x \) which we might be tempted to use to provide error recovery in an operation. Such constructs would make the reification condition unsound: in the implementation, \( f' \) might be defined at some \( x \) where it was not previously, but yield a value other than \( x \), so the value of \( f(x) \) would change.

There is nothing to stop us using a term of visible sort from the signature to test the argument (such as \( \text{isempty} \) for stacks), since that will continue to return the same result in the implementation.

We can use the reification condition to show that a definition of the stack operations which introduces a specific 'broken' state where the stack goes when we try to pop an empty one, and subsequently stays, implements the specification (as we would expect). Formally,

\[
\text{Stack}^d = \mathbb{N}^* \cup \{\text{broken}\}
\]
\[
\text{init}^d() \triangleq [\]
\]
\[
\text{push}^d(x, s) \triangleq \text{cons}(x, s)
\]
\[
\text{pop}^d(s) \triangleq (\text{hds}, s, t) s
\]
\[
\text{isempty}^d(s) \triangleq s = [\]
\]

is justified by the correspondence suggested by the completeness argument:

\[
sc \approx \text{Stack}^d \Rightarrow sc = sa
\]

It is onto, but not total, and no total relation will satisfy the reification condition, despite the fact that there is no junk in the implementation or the specification. This reflects the fact that some terms have values in the implementation but not the specification.

4. SPECIFICATION AND IMPLEMENTATION

The extension of the reification condition to partial operations makes it asymmetric, and imposes a direction of implementation: we can choose to replace a partial operation by a less partial one, but not vice versa. It is now starting to make more sense to talk about specifications and implementations rather than just a pair of definitions.

We can observe a much more significant indicator of the direction of implementation in the example from Section 1. The relationship between the set of trees and the set of lists is that each tree represents a particular set rather than corresponding to perhaps many sets. What we have done is to impose some tree structure on each set in the implementation. The arrangement of the tree is arbitrary, so there are many possible tree representations of each set. This multiplicity of representations contributes to the efficiency of \( \text{assert} \): were we to insist on single representation (such as height 1 trees with the least element as the root), \( \text{assert} \) would have to locate and update all the nodes of one of the merged trees instead of just the root. (We note, though, that just the addition of the structure is sufficient to improve \( \text{test} \).) Because the correspondence is many-to-one rather than many-to-many, it is functional rather than being a general relation, and we can rewrite the previous formal definition as

\[
a : \text{InterchangeablePart}^d \rightarrow \text{InterchangeablePart}^d
\]
\[
\text{assert}(\text{mzs}) \triangleq \{ p \in \text{Part} \wedge \text{root}(p) = \text{root}(p') \} \wedge p' \in \text{Part}
\]

At this point, we can take a philosophical position that what we mean by implementation is a combination of increasing definedness and (possibly multiple) representation. In that case, we should replace the relations in our formal reification condition by functions to give a new reification condition

\[
\mathcal{G} \subseteq \mathcal{A}, D \triangleq
\]
\[
\exists \{ a_s, S^d \rightarrow S^d : s \in \mathcal{S} \wedge (s \in \mathcal{S} \Rightarrow a_s \text{ is identity}) \}
\]
\[
\forall \theta : W_t \rightarrow W_s \in \Omega, \forall a \in W_t, c \in W_s
\]
\[
\forall \theta : W_t \rightarrow W_s \in \Omega, \forall a \in W_t, c \in W_s
\]
\[
d(\text{op}^d(a_s))(c) = \text{op}^d(a_s(c))
\]

(As before, we could strengthen this to total and onto functions.) We can think of the specifications as more abstract than the implementations because they suppress differences between values that are not significant in understanding the operations (such as the difference in structure between trees with the same set of nodes). The functions are known as abstraction functions or retrieval functions because they recover the abstraction from the more concrete implementation. This new reification condition is clearly sound with respect to the term condition of the previous section, since it is just a restriction of the previous reification condition. It is equally clearly not complete. The two definitions of the interchangeable parts system are themselves interchange-
able (either can be taken as the specification and the other as the implementation), but this condition only allows the second to implement the first.

However, out of the set of definitions for a given set of requirements (which agree on all visible terms, and on which terms are defined) there is always one which has all the definitions implementing those requirements related to it by \( \leq_s \). Intuitively, this is the definition of where there is no junk in the carriers, and where each value of each sort is visibly different from all other values of the sort. This is the smallest multiplicity of representation that we can get away with (1 when we need the value and 0 when we do not), so any other definition needs at least as many values. We say that it is fully abstract. In one sense at least, the fully abstract definition is the best specification of the requirements, because the others are all derived from it in a way we regard as 'real' implementation. We can strengthen our philosophical position to say that this is therefore the specification we should use.

This standpoint has some advantages when it comes to constructing proofs. Functions have traditionally been seen as easier to work with than relations, although this advantage is reduced in theorem-proving assistants which do not support equational reasoning. It is also likely that an implementation will be easier to prove from a more abstract specification than from some arbitrarily chosen definition. For this reason, the fully abstract definition is said to be free from bias towards any implementation. There is a technical advantage when it comes to stating properties of definitions: these will often refer to identical visible behaviour of two states of a system, but in a fully abstract definition this can be expressed as the equality of the two states, which is easier to reason about. There is also an aesthetic aspect of economy of means in the use of the smallest sets possible to interpret the hidden sorts.

Consider, though, a system to log some measurements (real numbers, \( \mathbb{R} \)) and provide an average on demand. Its signature is

\[
\text{init}: \rightarrow \text{Measurements} \\
\text{log}: \mathbb{R} \times \text{Measurements} \rightarrow \text{Measurements} \\
\text{average}: \text{Measurements} \rightarrow \mathbb{R}
\]

where \( \text{Measurements} \) is hidden. The natural specification keeps the sequence of measurements so far, logging each new one by adding it to the front, and computes the averages from the measurements when needed. There must, of course, be at least one to average.

\[
\text{Measurements}^d = \mathbb{R}^* \\
\text{init}^d() \triangleq [ ] \\
\text{log}^d(v, m) \triangleq \text{cons}(v, m) \\
\text{average}^d(m) \triangleq \text{sum}(m)/\text{len}(m) \\
\text{pre} \text{len}(m) > 0 \\
\text{sum}(s) \triangleq \begin{cases} 0 & \text{if } s = [ ] \\ \text{hd}(s) + \text{sum}(\text{tl}(s)) & \text{otherwise} \end{cases}
\]

A reasonable implementation just keeps a running total and sum of the measurements, and divides one by the other to get the average.

The first definition may seem like the better specification, being more like the current requirements and easier to adapt as they change (to compute a standard deviation as well as an average, perhaps). Unfortunately, the second is the fully abstract one, and the first definition is an implementation in the sense that we can abstract \( \text{Measurements}^z \) from \( \text{Measurements}^d \).

\[
\begin{align*}
\text{a}_{\text{Measurements}} : & \text{Measurements}^d \rightarrow \text{Measurements}^z \\
\text{a}_{\text{Measurements}}(m) & \triangleq (1: \text{len}(m), \text{sum}(m))
\end{align*}
\]

Examples like this suggest that we do not always want to use \( \leq_s \) as the reification condition rather than \( \leq \).

5. REIFYING LOOSE SPECIFICATIONS

In many cases, the requirements do not determine a unique result for each operation, but rather describe properties that the result must have. An obvious example is a storage-management system, where each call of the allocation operation must give an address that was not previously allocated (and remember that it has been allocated), but the actual address returned is unimportant. If we take addresses as natural numbers (\( \mathbb{N} \)), the signature of such a system might be

\[
\text{allocate}: \text{Allocations} \rightarrow \mathbb{N} \times \text{Allocations} \\
\text{init}: \rightarrow \text{Allocations}
\]

where \( \text{Allocations} \) is hidden.

To record these requirements exactly, without imposing any arbitrary storage allocation policy, we need loose specifications which capture just the defining properties of the result rather than giving a particular answer. The obvious approach is to make the state the set of addresses allocated so far (initially the empty set) and just say that the allocated address is not in it. (It is of course added immediately.) Formally, we can write the property as a postcondition on the inputs and results

\[
\begin{align*}
\text{allocations}^d & = \mathbb{N} \times \text{Allocations} \\
\text{allocate}^d(a, as) & = a \land \text{as} = as \cup \{a\} \\
\text{init}^d & \triangleq \{ \}
\end{align*}
\]

All such a specification promises to an exploiting definition is that the value of a term of visible sort will satisfy the operation definitions. An implementation may thus define the value of a term more precisely than (be more deterministic than) the specification without any effect on an exploiting definition. In many cases, the final implementation will be completely deterministic, although there may be several intermediate stages. This provides another sense in which implementations are closer to the machine. As before, it may also define a result where none is defined by the specification, and give a different interpretation to the hidden sorts.

In order that the exploiting definition may use constructs like \( \text{let } x = t \text{ in } t' \) with \( t' \) of visible sort, we require any value of \( t' \) for any assignment of \( x \) to a value of \( t \) in the implementation to be a value of \( t' \) for some assignment of \( x \) to a value of \( t' \) in the specification. This is the counterpart to the term condition that we arrived at in Section 3. Note, though, that we can no longer think of the \( \text{let} \) expression as a shorthand for \( t' \) with all occurrences of \( x \) replaced by \( t \), and so including terms with variables in the term condition now makes a
difference. For example, if we imagine tossing a coin, which we can formalise as

\[
\begin{align*}
\text{toss} & \rightarrow \{\text{Heads}, \text{Tails}\} \\
\text{toss}^\omega(r) & \text{post} \rightarrow \{\text{Heads}, \text{Tails}\}
\end{align*}
\]

(with \{Heads, Tails\} visible) tossing it twice may give different results (so, formally, the term \text{toss} = \text{toss} can evaluate to \text{true} or \text{false}), but the value of one toss is fixed (so, formally, the term \text{let } x = \text{toss} in x = x evaluates to \text{true}). Variables serve to 'freeze' the nondeterminism, reducing the set of possible outcomes.

As in Section 2, the term condition is inconvenient to test, and we should prefer an equivalent test on the operation definitions alone. Again, we can guarantee the term condition by finding correspondences between the carriers of the definitions, using equality as the correspondence for all visible sorts, where any result from an operation on some values in the implementation corresponds to a possible result from the operation on the corresponding values in the specification. We can write this formally as a relation \(\subseteq\) between definitions, using the postconditions to define a relationship between inputs and possible outputs and taking care of possible undefinedness of the operations:

\[
\begin{align*}
\forall s, s' \in S. & (s \subseteq s') \land (s \in S) \land (s' \in S) \\
\forall a \in W. & (a \in W') \land (a' \in W'_1) \\
\forall c \approx_{w, a} & (D(op^\omega(c)) \land c' \approx_{w, a'} \lor \exists a' \in W'. \text{post-op}^\omega(c, a') \land c' \approx_{w, a'})
\end{align*}
\]

\((D(t)\text{ here means that } t \text{ evaluates to at least one value.})\)

The correspondences are again not necessarily total or onto, even when operations are total and carriers are free from junk, since with this reification condition only some of the specification values need to correspond to values in an implementation of increased determinism. The reification condition also appears to be complete with respect to the term condition for practical purposes, although the correspondences are much more difficult to derive, and some technical restrictions have to be imposed on the definitions (for example, the specification must not generate hidden values which cannot be used by any term with visible results).

Intuitively, a possible implementation of the storage-management system could allocate the addresses in order (hence deterministically). In that case, we need only keep a record of the highest address allocated.

\[
\begin{align*}
\text{Allocations}^\omega & \triangleq \mathbb{N} \\
\text{init}^\omega() & \triangleq 0 \\
\text{allocate}^\omega(\text{lim}) & \triangleq (\text{lim} + 1, \text{lim} + 1)
\end{align*}
\]

We can justify this formally as a reification using the correspondence

\[
\begin{align*}
n \approx_{\text{Allocations}} & \triangleq as = \{i | i \in 1, \ldots, n\}
\end{align*}
\]

6. SPECIFICATION AND IMPLEMENTATION REVISITED

When definitions are deterministic, we saw that it was possible to find a fully abstract specification to which all implementations were related by functions, and that in many cases this was the most appropriate one to use. We should see whether a similar situation applies for loose specifications. The corresponding reification condition will be

\[
\begin{align*}
\forall s, s' \in S. & (s \subseteq s') \land (S \subseteq S') \\
\forall a \in W. & (a \in W') \land (a' \in W'_1) \\
\forall c \approx_{w, a} & (D(op^\omega(c)) \land c' \approx_{w, a'} \lor \exists a' \in W'. \text{post-op}^\omega(c, a') \land c' \approx_{w, a'})
\end{align*}
\]

We note that the correspondence above is functional, so we could also justify our proposed implementation using this reification condition.

Our idea of a fully abstract definition was one where every value of each sort was necessary because it could both be generated and be distinguished from all the others. It was essentially unique (we can define it in many ways, but there will be a 1:1 correspondence between all of them). Unfortunately, this is no longer the case with loose specifications. For example, while the definition of \text{Allocations}^\omega is fully abstract (we can distinguish all the states by looking at the addresses they can allocate), so is the definition of \text{Allocations}^\omega. While we might argue that the first of these captures the requirements, the second clearly imposes some sort of allocation policy.

This arises because there is interaction between the old aspect of implementation and the new one: increasing the determinacy means that we need fewer values to represent the hidden sorts while retaining one or more representations for each term value. As a result, we cannot always find a specification from which all the others can be derived using abstraction functions only. Consider the following example, where we can toss a coin, turn it over, and look at the result. The signature is

\[
\begin{align*}
\text{toss} & \rightarrow \text{Result} \\
\text{flip} \rightarrow \text{Result} \\
\text{look} & \rightarrow \text{Result} \rightarrow \mathbb{N}
\end{align*}
\]

(where \text{Result} is hidden). We specify the operations to use a conventional coin, with heads and tails. Tossing it may give either result, flipping it turns it over, and looking at it can return 0 or 1 for 'heads', and 1 or 2 for 'tails'. (Like many such counterexamples, this may seem an unlikely set of requirements!) Formally, we define

\[
\begin{align*}
\text{Result}^\omega & \triangleq \{\text{Heads}, \text{Tails}\} \\
\text{toss}^\omega(r) & \text{post} \rightarrow \{\text{Heads}, \text{Tails}\} \\
\text{flip}^\omega(r) & \triangleq \text{if } r = \text{Heads then Heads else Heads} \\
\text{look}^\omega(r,n) & \text{post} (r = \text{Heads} \land n \in \{0, 1\}) \lor (r = \text{Tails} \land n \in \{1, 2\})
\end{align*}
\]

The overlap of the two results allows us always to return 1 in an implementation of \text{look}, in which case we might as well use a 1-sided coin.

\[
\begin{align*}
\text{Result}^\omega & \triangleq \{\text{HT}\} \\
\text{toss}^\omega() & \triangleq \text{HT} \\
\text{flip}^\omega(r) & \triangleq \text{HT} \\
\text{look}^\omega(r) & \triangleq 1
\end{align*}
\]

Imagine that we could find a definition \(\mathcal{B}\) which behaves identically to \(\mathcal{A}\) (and hence \(\mathcal{A} \subseteq \mathcal{B}\) and \(\mathcal{B} \subseteq \mathcal{A}\)), and from which \(\mathcal{A}\) could be derived with abstraction function
g. We must have \( g(HT) \approx \text{Heads} \) and \( g(HT) \approx \text{Tails} \) by considering the implementation of \( \text{toss}^g \) and \( \text{flip}^g \) by \( \text{toss}^f \) and \( \text{flip}^f \). But that means that for \( \text{look}^g \) and \( \text{look}^f \) to mutually implement each other, we must have \( \text{look}^g(\text{g(HT)}) \leq \text{look}^f(\text{Heads}) \cup \text{look}^f(\text{Tails}), \text{look}^f(\text{Heads}) \leq \text{look}^g(\text{g(HT)}) \) and \( \text{look}^f(\text{Tails}) \leq \text{look}^g(\text{g(HT)}) \), which is not possible. In general, we cannot find a specification which uses fewer values than an implementation but achieves some particular less deterministic behaviour.

However, a reasonable strategy for arriving at a specification is to consider how much non-determinacy the requirements allow, and then choose the definition of the sorts to support it with one value for each distinct state. This has the spirit of the principle which led us to the fully abstract specification in Section 4, if not the letter of its definition. For example, we can consider increasing the nondeterminacy of the original specification of the storage allocation system by not only allowing any free address to be returned, but also allowing the allocation process to mark as in use any superset of the addresses actually returned so far. Such a specification can be given formally with the states represented as sets as before:

\[
\text{Allocations}^g = \text{N-set} \\
\text{INIT}^g(\cdot) \triangleq \{\} \\
\text{allocate}^g(\cdot)(a, a^\prime) \\
\text{POST} \cdot a \notin \cdot \land \cdot a \subseteq \cdot a^\prime \land a \in a^\prime
\]

This makes explicit the possibility of implementations where some overhead is used for managing the storage (such as might be required if we had a means of disposing of allocated storage), which the original definition did not. All the other definitions of the storage system above can be shown to implement this one using the \( \equiv \) definition.

### 7. HISTORY AND BIBLIOGRAPHY

The literature on data type definitions and their implementation is very extensive, and the bibliography here can only provide an introduction to it. It is also a very diverse literature. Differences in formalisms account for some of this diversity. In particular, the definitions here are model-based, \(^{16} \) while much of the research on data type refinement has been done in the property-based (more specifically, equational or algebraic) setting, \(^{7} \), where sorts and operations are defined more implicitly. Authors also use different names for essentially the same conditions. However, as we have seen, the reification conditions we define depend heavily on what we think of as an implementation, and this gives rise to many definitions that are different in nature. We can only compare them when we have understood the context for which they are intended. Work that has looked at a range of notions of refinement includes Refs 2 and 28 as well as 29, while Ref. 11 establishes a categorical framework for studying the interaction of language constructs and reification conditions.

Of the reification conditions in this paper, \( \equiv \) was developed in Schoett \(^{27} \) for modular algebraic specifications, as a generalisation of the, historically earlier, \( \equiv \) condition from Hoare, \(^{14} \) developed in a programming language setting. This in turn was a specialisation of the correspondences between program states used in Ref. 21 to show programs equivalent. The idea of identifying hidden and visible sorts when characterising the behaviour of a specification comes originally from Ref. 8, and the proof that there is always a fully abstract specification of a deterministic set of requirements is given in Nipkow. \(^{24} \) The implementation of interchangeable parts in the introduction uses the Fischer–Galler algorithm, which has been a popular example for reification techniques, \(^{14} \) and the idea of characterising reification conditions by the terms whose values they preserve is due to Sannella and Wirsing. \(^{28} \)

The \( \equiv \) condition comes from Refs 22 and 23, where it is known as weak simulation, and is developed in the context of a careful design of languages for exploiting definitions (seen as programs). The only difficulty is that these programs cannot detect all the differences that a term condition can (essentially, because a term may have more than one value). As a result, completeness cannot be established due to the existence of some uninformative ‘implementations’, which cannot be justified by \( \equiv \). Another reification condition, called simulation, is shown to be more appropriate than \( \equiv \) for programs with parallelism. The \( \equiv \) condition is almost the verification condition used by VDM. \(^{18} \) The inability to find a specification from which all others can be derived using \( \equiv \) is established in Ref. 24. The VDM condition insists that the abstraction function are total and onto (adequate, in VDM terminology). The argument for this is that it encourages good specifications, and makes verification of the reification condition, easier, but we have seen that it involves a further sacrifice of completeness.

The \( \equiv \) condition was developed independently by Hoare and others, \(^{10, \ 13} \) where it is known as downward simulation. Their work provides another illustration of the importance of how we exploit a definition in establishing the soundness and completeness of a reification condition. A value of a hidden sort is regarded as a private initial state rather than as the result of an operation delivered to an exploiting definition. (A similar view is taken by VDM for its operations.) As a result, it is not possible to binds a hidden value to a variable. This weakens the term condition, and so \( \equiv \) is no longer complete. For example, in a system which generates a single (visible) random number and then stops

\[
\text{INIT}^g = \text{N} \cup \{\text{DONE}\} \\
\text{INIT}^g(\cdot) \triangleq \{\} \\
\text{allocate}^g(\cdot)(a, a^\prime) \\
\text{POST} \cdot a \notin \cdot \land \cdot a \subseteq \cdot a^\prime \land a \in a^\prime
\]

the specification may pick the number in advance

\[
\text{Stat}^g = \text{N} \cup \{\text{DONE}\} \\
\text{INIT}^g(\cdot) \triangleq \{\} \\
\text{allocate}^g(\cdot)(a, a^\prime) \\
\text{POST} \cdot a \notin \cdot \land \cdot a \subseteq \cdot a^\prime \land a \in a^\prime
\]

while a plausible implementation might wait until the number is requests.

\[
\text{Stat}^g = \{\text{READY, DONE}\} \\
\text{INIT}^g(\cdot) \triangleq \text{READY} \\
\text{allocate}^g(\cdot)(a, a^\prime) \\
\text{POST} \cdot a \notin \cdot \land \cdot a \subseteq \cdot a^\prime \land a \in a^\prime
\]

These behave the same (delivering a \( \text{random integer} \) unless we can freeze the initial state in a \( \text{STATE} \) and see
that the specification always returns the same number while the implementation does not. As a result, there is no correspondence \( \approx_{\text{state}} \) validating the reification condition. He and Hoare\(^2\) give an upwards simulation condition which is complete in conjunction with downwards simulation: if \( \mathcal{C} \) implements \( \mathcal{A} \), there is a third definition \( \mathcal{B} \) which upwards simulates \( \mathcal{A} \) such that \( \mathcal{C} \subseteq \mathcal{B} \). They can however show that downwards simulation alone is complete for a subset of definitions, the canonical specifications where the state after each operation is determined by the inputs and visible results of that operation. Thus even with this further restriction on exploiting definitions \( \mathcal{E} \) is nearly always enough in practice.

Another area where changing the way in which specifications can be exploited affects the soundness of reification conditions is that of secure systems.\(^15\) Increasing the determinacy of an operation may allow the user to deduce more about the circumstances in which the operation is applied, and thereby obtain information to which they are not entitled and probably could not obtain using the operations as specified.

When all the operations are total and there is only one (hidden) sort, the \( \approx_{r} \) reification condition corresponds to the standard mathematical notion of homomorphism between algebras.\(^8\) Considerable effort has been devoted to extending this to many sorts and partial operations.\(^3\) We are particularly interested in defining homomorphisms so that there is a final algebra: that is, one to which all the others are related by a homomorphism to it. This corresponds to the fully abstract specification of Section 4, and is the one that gives the most natural interpretation of property-based specifications for further implementation.\(^19\)\(^20\) One area of particular concern in property-oriented specifications has been that of data type parameterisation.\(^9\)\(^11\)\(^12\) In the example from the introduction, we never say what \textit{Part} is, assuming that the specification will meet the specification for any choice (at least as long as we can tell when two parts are equal). More generally, we can specify what properties a parameter sort must have. For example, we can specify ordered binary trees if the sort of the elements has a comparison operation. We expect that implementations of the specification will work with any definition of the parameters with the necessary properties. More recently, Sannella and Tarlecki have returned to this issue.\(^28\)

We have assumed that the implementation as well as the specification is given, and that the reification condition is then verified \textit{post hoc}. In practice, we usually have an informal idea of the correspondence with the specification as we construct the implementation, and the verification justifies the intuition. An alternative approach is to synthesise the implementation from the specification by a series of steps, each of which is known in advance to be correct. Darlington\(^4\) describes an approach to deriving implementations of operations given an abstraction function, while Oliveira presents a calculus for building correct implementations of data types based on set constructions.\(^25\)\(^26\) Burstall and Landin give an early example of data type reification, using categorical ideas to construct abstraction functions.\(^1\) Morgan and Gardiner provide an extension to their refinement calculus for introducing modules implementing data type abstractions.\(^10\)

Finally, the term ‘data reification’ itself is introduced by Jones,\(^17\) in response to a comment from Michael Jackson that the more usual term ‘data refinement’ is hardly appropriate to the replacement of an elegant mathematical abstraction by some collection of words and pointers.

8. SUMMARY

We can see the development of reification conditions as an exercise in formal development itself. We had to begin by deciding what we mean by a definition. We could then establish a connection between these meanings, based on our requirements for what we expected an exploiting definition to be able to depend on. We assumed that the values of terms of hidden types were of no interest except as ways of passing values from one operation of the specified system to another, that the exploiting definition could only make use of operations where they were defined by the specification, and that when operations are loosely defined, any result meeting the postcondition will be satisfactory. We formalised this by saying that values of defined terms of visible sort in the implementation had to have values they could have according to the specification, provided values assigned to variables could actually be produced by evaluating some term. This last restriction corresponds to our intuition that these are the only values that arise while executing programs, but prevents us from using arbitrary quantified expressions in the exploiting definition.

This is fine as a specification of when one definition implements another, but is not much use for determining whether we have an implementation in any particular case. We can see the reification conditions as the (meta)implementation of the (meta)specification, giving a practical test for being an implementation (although we may still need some insight to construct the \( \approx_{r} \) relations). That means, of course, that we must establish a connection between the metaspecification and the reification conditions: we want them to be sound and ideally complete.

What the definitions \( \approx_{i} \) and \( \subseteq \) lacked was any notion that values in the implementations normally represent values in the specification. The restriction of the correspondences between carriers to be functional formalised this intuitive view of implementation, although we saw that not all implementations do represent values from the specification in this way, and that changes in representation of data could be confused with increasing determinacy of operations in the case of \( \subseteq_{F} \).

What do we conclude from all this theory? That the reification condition \( \subseteq_{r} \), which reduces to \( \approx_{i} \) in the case of deterministic specifications, is the appropriate one in nearly all cases. Because it imposes a further sense of direction from specification to implementation, it will tend to guide us to more abstract specifications. We should regard \( \subseteq \) like the \textit{repeat} loop in Pascal: there are cases where it is more appropriate, but they are sufficiently rare that we should check whether an alternative approach to the requirements allows us to use the more standard method. In particular, it may be useful to split developments from loose specifications into steps which increase determinacy and those which change representation when possible.

We should not necessarily require the abstraction functions (or correspondences, if we use \( \subseteq \)) to be total
and onto, since to do so will rule out some perfectly reasonable implementations. However, it is reasonable to define the specification to have no junk in its carriers (by use of a predicate on the carrier values, if necessary). For example, we should insist that the sets in InterchangeableParts* are disjoint and cover Part.

\[
\text{inv-InterchangeableParts : InterchangeableParts} \rightarrow \mathbb{B}
\]

\[
\text{inv-InterchangeableParts(pss)} \triangleq
(\forall s_1, s_2 \in \text{pss}. s_1 \neq s_2 \Rightarrow s_1 \cap s_2 = \{\}) \land \bigcup (\text{pss}) = \text{Part}
\]

This brings requirements and specification closer together, which is good, since the step from one to the other cannot be formally verified. Freedom from junk was also a requirement for full abstraction. In the deterministic case, at least, the correspondences will then be onto. Totality makes the functions easier to use in verifying the

reification condition, but if it is achieved by an invariant it is as easy to provide and check a precondition on the function. In any case, programming languages have no counterpart to invariants in type declarations, so using them in the implementation definition widens the gap between definition and program. Because this is another gap that is often bridged informally, this is something we want to avoid.

Finally, we should recognise that many implementations are derived in standard ways from specifications, and calculate them rather than positing them and proving that the reification condition holds. The theory of reification then serves just as a way to show that each step of the calculation gives an implementation that can replace the specification.

REFERENCES


THE COMPUTER JOURNAL, VOL. 35, NO. 5, 1992 459