Design Strategies for Algebraic Specifications

Sergio Antoy

TR 89-32

October 1989

Design Strategies for Algebraic Specifications

Sergio Antoy

Virginia Tech
Northern Virginia Graduate Center
2990 Telestar Ct.
Falls Church, VA 22042-1287
(703) 698-6088
antoy@vtopus.cs.vt.edu

Abstract: The algebraic specifications of abstract data types can be affected by serious flaws. Rather than attempting, as it is usually done, to detect defects in a specification *a posteriori*, i.e. after a specification has been designed, we propose two strategies for addressing this problem *a priori*, i.e. during the design phase of a specification. We investigate two common flaws of algebraic operations, under- and over-specification, determine sufficient and/or necessary conditions to avoid them, and show how to obtain these conditions in a constructive way. Our approach is based on the completeness and parsimony properties of sets of tuples of terms and on a recursive mechanism which extends primitive recursion from natural numbers to abstract data types. Finally, we attempt to assess the power and the limitations of our approach and relate the properties on which it is based to other similar properties which have appeared in the literature. We formally prove a number of results and illustrate their application to the design of specifications by means of examples.

Keywords: Algebraic specifications, Software design, Under- and over-specification, Completeness, Parsimony, Termination, Linearity, Binary choice procedure, Recursive reduction.

1. Introduction and Motivation

Software specifications are an early phase of software development which has profound repercussions on the whole software life-cycle. This consideration alone should suffice to claim that

This material is based upon work supported by the National Science Foundation Grant No. CCR-8908565. The Government has certain rights in this material. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation.

specifications should be designed in a systematic way and should be defect free. What sometimes is not so obvious is the extent of the consequences of flawed specifications. We support this point, by presenting an example which shows how the consequences of lack of a methodology in the design of specifications may affect code implementation and its verification.

The scenario of our example consists of a fragment of an annotated program and a specification. Both are extremely simple, but more realistic variations will be outlined in the discussion. The scenario presupposes the existence of a secure database and agents which either write or read data items. A reader is granted access to an item if its privilege (security-level) is higher than the privilege of the agent who wrote the item. Otherwise the program terminates with a suitable exit code. The annotations are shown in italics between braces. The program fragment which controls the access is:

$\{true\}$
if *reader-privilege* $\leq$ *writer-privilege*
  then exit(ACCESS_DENIED)
  else $\{$*reader-privilege* $\not\leq$ *write-privilege*$\}$
  ... grant access ...

The desired behavior of the fragment is that "no matter what" (this is the meaning of $\{true\}$ as a precondition) access to the database is granted only if the privilege of the reader of an item is not less than or equal to that of the item's writer. The second annotation is the crucial one, in fact it states the precondition for granting access to the item.

With appropriate specifications to give meaning to the annotations, the program fragment can be proved to be correct, since the program does what the annotation requires. The proof is simple enough to be carried on by an automatic prover if a suitable axiomatization is available. We will provide axioms only for the operation, denoted by the symbol "$\leq$" in the code, which compares two access privileges.

We assume that the privilege of an agent is expressed by a natural number, thus "$\leq$" is simply the "usual less than or equal operator". In a more realistic setting, access to an item should be granted to its writer, furthermore, the privilege may be a vector with components such as the rank of the agent, the organization and/or department to which s/he belongs, the project(s) to which s/he has been assigned, etc. Since comparing such objects is no longer "usual", a precise axiomatization of the comparison of privileges would become necessary. Here, we simplify the problem by specifying the comparison between natural numbers. Even in this simple case errors may occur. Our specification is provided in an algebraic framework.

The specification which a careless or unskilled designer may propose consists of the following

three equations. Capitalized letters at the end of the alphabet, i.e. X and Y denote implicitly universally quantified variables.

$$0 \leq 0 = true \tag{1.1}$$
$$succ(X) \leq 0 = false \tag{1.2}$$
$$succ(X) \leq succ(Y) = X \leq Y \tag{1.3}$$

The correctness of the program fragment is, in a strict sense, independent of this specification if the comparison operator implemented in the program and that denoted by the annotation are the same. We assume just so in our scenario.

Let us consider, however, what happens when, for example, the privilege of the reader is 2 and that of the writer is 3. Two applications of equation (1.3) reduce the initial situation to the comparison of 0 and 1. At this point, no equation can be further used and $0 \leq 1$ cannot be equated to either *true* or *false*. The specification leaves the implementor free to return anything s/he choses for this situation. If the implementation does not return any output, the program fragment does not terminate, a situation which is always undesirable. The implementation could return *false* for a number of reasons. This might increase the efficiency or simplify the code or it may "look" safer — it seems we prefer to say "No" in uncertain situations. The consequence is that access will be granted to the reader.

> It cannot be over-emphasized that the program fragment is correct with respect to its annotation, that the annotation states exactly what we desire in this case, and that the implementation of the specification of "$\leq$" is formally correct.

The problem, in the above scenario, stems from the fact that the axioms for the operation "$\leq$" are poorly designed (or are not enough.) The fact that algebraic specifications can be affected by serious flaws is well-known. This is not perceived as an intrinsic weakness of the formalism. Actually, algebraic specifications are the environment in which addressing and detecting design flaws seems easier and more successful than in other specification methods. We quote from [FUTA84] "The original motivation of this work [OBJ0 and OBJT] was the observation that algebraic specifications published in the literature where very often wrong, so that some way of testing them was needed." The motivation of our work is the same, but our approach is quite different. Rather than attempting to detect through testing the flaws of a specification *after* it has been designed, we explore the possibilities and the limitations of designing it in a way which avoids or minimizes the introduction of flaws. This approach is analogous to the principles of structured programming which aim at the same goal in the realm of programs rather than in that of specifications.

This paper is organized as follows. In Section 2 we introduce terminology and notation. In particular we define an algebraic specification as a 6-tuple and present a minimal language in which the specification's components can be precisely stated and easily perceived. In Section 3 we introduce the concepts of under- and over-specification, we show why these conditions should be avoided in a specification, we relate these concepts to two properties of sets of tuples of term, i.e. completeness and parsimony, and we propose a strategy for the design of complete and parsimonious sets. In Section 4 we introduce a strategy for the definition of recursive operations and we prove that operations designed through the combined use of this and the previous strategy are neither under- nor over-specified. In Section 5 we prove additional results concerning the design strategies. These results focus on what is possible and what is impossible to do through the design strategies and are an attempt to evaluate their power and the limitations. In Section 6 we compare our results to similar or related ideas which have appeared in the literature. The example presented in the discussion are selected by merit of simplicity. In the appendix we show some more significant examples.

2. Preliminary Concepts

The target of our investigation are specifications whose semantic model is called initial [GOGU78] and with which we assume some familiarity. Our approach follows this model with two minor deviations: the orientation of the specification axioms from left to right and the partition of the specification signature into constructors and operations. Both characteristics are implicitly present since early work on algebraic specifications [GOGU78, GUTT78a], but have been employed systematically only more recently. Through them we address a number of properties whose absence in a specification is often the consequence of poor understanding of the problem and causes difficulties in many specification-based operations, such as code implementation, verification, maintenance, and/or documentation.

At the core of any algebraic specification there is a fairly typical set of axioms. These axioms are equations such as (1.1) through (1.3) shown above. The equations we choose to consider are "oriented" [HUET80a] or, in other worlds, are rewrite rules. The introduction of a direction allows us to define a left and a right side in an axiom and is not a limitation, since we can still regard axioms as mere equations when it is convenient. In addition to orienting the axioms, we also choose to declare the constructors [HUET82] of each sort of our specifications. Again, the declaration of the constructors is not a limitation, since we may disregard any partition of the symbols of the signature.

A *specification* is a 6-tuple $\langle \mathcal{S}, \mathcal{C}, \mathcal{D}, \tau, \mathcal{Q}, \mathcal{A} \rangle$, where the various components are defined as follows. $\mathcal{S}$ is a set of symbols called *sorts*. $\mathcal{C}$ is a set of symbols called *constructors*. $\mathcal{D}$ is a set of symbols called (*defined*) *operations*. $\mathcal{C}$ and $\mathcal{D}$ are disjoint and their union is denoted with Σ, i.e. $\Sigma = \mathcal{C} \uplus \mathcal{D}$. A set of constructors and/or operations is almost always implicitly associated with the function τ defined below. With this association in mind, the set is referred to as a *signature*. τ is a mapping from Σ into $\mathcal{S}^+$ called *arity*, $\mathcal{S}^+$ denotes the set of non-null strings over $\mathcal{S}$. For any f in Σ, $\tau(f)$ is denoted either as s or as $s_1 \times \ldots \times s_n \rightarrow s$ when it is longer than 1. s and $s_1 \times \ldots \times s_n$ are respectively called *range* and *domain* of f. $\mathcal{Q}$ is a set of rewrite rules, called *quotient axioms*, in which there are no occurrences of defined operations. $\mathcal{A}$ is a set of rewrite rules, called *defining axioms*, in which the root of each axiom left side is an operation. $\mathcal{Q}$ and $\mathcal{A}$ are obviously disjoint and their union is denoted by $\mathcal{E}$, i.e. $\mathcal{E} = \mathcal{Q} \uplus \mathcal{A}$.

We recall that all the variables occurring in the right side of a rewrite rule must occur in the left side too. $T(\bullet)$ is the set of *terms* built over a signature denoted by the symbol $\bullet$. $T(\bullet)_s$ is the subset of $T(\bullet)$ of all the terms of sort s. $\mathcal{X}$ denotes a set of sorted *variables*. An element in $T(\mathcal{C} \cup \mathcal{X})$ is a *constructor term*. A term t of the form $f(x_1, \ldots, x_n)$, where f is an operation and the arguments are constructor terms is an *f-rooted constructor term* and f is the *root* of t. An axiom $\alpha \rightarrow \beta$ such that the root of α is an operation f is a *defining axiom* of f.

For the purpose of presenting our ideas and showing their applications to examples, we define a minimal language in which the various components of a specification can be precisely presented and more easily perceived. The language has only two statements: one for specifying sorts and the other for specifying operations. Statements are terminated by a period. Sort specifications consist in a declaration of the sort's symbol, followed by the enumeration of its constructors with their arities, followed by the enumeration of its quotient axioms, if any. The arity of a symbol of the signature follows the symbol declaration. A slight variation is appropriate for the constructors. Since the range of a constructor is obviously the sort in which the constructor is declared, only the domain of a constructor needs to be specified. Natural and conventional notations for the signature symbols are preserved by allowing the use of infix and "mixfix" [FUTA84, GUTT85] operators. The declaration of the domain of any such symbol supplies a template for these characteristics. We ignore problems of precedence, associativity, and parsing, since they are irrelevant to our discussion. Operation specifications consist in a declaration of the operation's symbol and arity followed by the enumeration of the operation's defining axioms. Variables in the axioms are denoted by capitalized identifiers and are implicitly universally quantified. Thus, referring to Figure 1, *0* is a constant

constructor of *nat* and *succ* is a constructor of *nat* which takes, between parentheses, one argument of sort *nat*. The symbol "+" denotes a binary, infix operation on *nat*, the familiar addition, and is specified by the two axioms at lines (7) and (8).

```
sort nat                                                    (1)
  constructors                                              (2)
    0;                                                      (3)
    succ(nat).                                              (4)
  operation nat + nat → nat                                 (5)
  axioms                                                    (6)
    0 + Y → Y;                                              (7)
    succ(X) + Y → succ(X + Y).                              (8)
```

Figure 1. Definition of the sort *nat*, modeling the natural numbers, and the addition.

The few remaining features available in the small language are presented in the next figure. Any variable which occurs in an axiom in only one occurrence (which must be in the left side) is called *anonymous* and may be denoted by the underscore symbol. Generic sorts are defined through the use of some parameter sorts. A sort s is a *parameter* of a specification when there are no occurrences of the constructors of s in $\mathcal{E}$. For example, the sort *list* (Figure 2) is parameterized by the sort *element* — the sort of the elements of a list. Finally, we need a notation for computations which are undefined for some arguments, such as the *head* of a null list. These computations are called *exceptional* and are denoted by the distinguished symbol "?" on the right side of an axiom. This is analogous to the Larch's clause *exempt* [GUTT85a,GUTT85b]. These exceptional axioms are part of our language but should not be considered as axioms of the specification. The specifications of partial functions are difficult to handle and have been approached with partial algebras, conditional rewriting, and order-sorted algebras. We limit our discussion to the initial model and need exceptional axioms in our language only to discriminate between exceptionality and incompleteness, i.e. omissions such as that presented in our introductory example and further elaborated later.

Our language, unlike similar ones, e.g. Larch [GUTT85], OBJ3 [GOGU88], lacks type declarations for variables. For reasons which will become clear shortly, the left sides of our axioms are always *linear* terms, i.e. their variables occur in only one occurrence. Since the scope of a variable is limited to the axiom in which it occurs, its sort is precisely and uniquely defined by its occurrence in the axiom's left side. In this situation, any explicit sort declaration for a variable outside its context is inappropriate, since it is a potential source of inconsistency and seems to imply a broader

sort *integer*
  constructors (1)
    0; (2)
    succ(*integer*); (3)
    pred(*integer*); (4)
  quotient (5)
    $succ(pred(X)) \rightarrow X$; (6)
    $pred(succ(X)) \rightarrow X$. (7)
sort *element* **parameter.** (8)
sort *list* (9)
  constructors (10)
    []; (11)
    [*element*|*list*]. (12)
operation $head(list) \rightarrow element$ (13)
  axioms (14)
    $head([]) \rightarrow ?$; (15)
    $head([X|\_]) \rightarrow X$. (16)
(17)

Figure 2. Additional features of the small language: quotient axioms, lines 6-8; parameter sorts, line 9; exceptional computations, line 16; anonymous variables, line 17. The constructors of *list* are denoted in standard Prolog notation. In our discussion, the symbols *0* and *succ* are unrelated to the symbols with the same name of Figure 1.

scope. Furthermore, a concept we will introduce in a following section greatly enhances the use of anonymous variables for which a sort declaration is inconvenient. The use of anonymous variables makes definitions more understandable by hiding irrelevant details.

3. Completeness and Parsimony — Binary Choices

In Figure 3 we present, more precisely, all the pieces of the faulty specification discussed in Section 1. The problem with this specification is that the operation "$\leq$" is under-specified, i.e. it is impossible to get a *Boolean* value from the application of "$\leq$" to certain pairs of natural numbers. This is an internal contradiction of the specification. In fact, lines (1) through (3) claim that any *Boolean* value is either one of the (constant) constructors *false* or *true*. Line (4) claims that the operation "$\leq$" applied to any pair of arguments of sort *nat* yields a *Boolean* value. However, the axioms at lines (6) through (8) fail to provide this condition, e.g., for $0 \leq succ(0)$. We formally characterize this situation in the next definition.

Definitions 1. An operation f is *under-specified* if there exists some ground f-rooted constructor term which cannot be rewritten to some ground constructor term. By extension, an *under-specified* specification is one containing an under-specified operation.

sort *Boolean* (1)
constructors (2)
$false; true.$ (3)
operation $nat \leq nat \rightarrow Boolean$ (4)
axioms (5)
$0 \leq 0 \rightarrow true;$ (6)
$succ(X) \leq 0 \rightarrow false;$ (7)
$succ(X) \leq succ(Y) \rightarrow X \leq Y.$ (8)

Figure 3. Specification of the sorts *Boolean* and the operation "$\leq$". The operation "$\leq$" is under-specified. The sort *nat* is specified in Figure 1.

In this section we introduce a technique which, in combination with a second technique introduced in a following section, allows us to design specifications which are not under-specified, hence to avoid the problems of Figure 3. The first technique we are going to describe takes care as well of over-specification, a problem somewhat opposite to under-specification. Over-specification arises when we say too much, rather than too little, about an operation and is sometimes the consequence of a naive approach to avoid under-specification. Over-specification, similar to under-specification, makes a specification either flawed or difficult to maintain.

operation $max(nat, nat) \rightarrow nat$ (1)
axioms (2)
$max(0, Y) \rightarrow Y;$ (3)
$max(X, 0) \rightarrow X;$ (4)
$max(succ(X), succ(Y)) \rightarrow max(X, Y).$ (5)

Figure 4. Specification of the operation *max* which computes the maximum of two natural numbers. The operation *max* is over-specified. The sort *nat* is specified in Figure 1.

In Figure 4 we present the operation *max* which computes the maximum of two natural numbers. The operation *max* is over-specified in the sense that the term $max(0, 0)$ can be reduced by both the axioms at lines (3) and (4). Since both axioms reduce $max(0, 0)$ to the same term, i.e. 0, there are no significant problems at this time. Over-specification *per se* is not a contradiction. However, despite its apparent adequacy, the situation resulting from the specification of *max* is dangerous. Later on, a careless maintainer of the specification could change the axiom at line (3) to, say:

$$max(0, Y) \rightarrow succ(succ(Y)) \tag{3.1}$$

The consequence would not only alter the definition of *max* — such a change could be intentional and appropriate if the meaning of *max* were to be modified — but also dramatically affect the

confluence of the specification. We will define this property and discuss it at length later. Here we simply show the implications of such a change. A consequence of (3.1) is that either $max(0,0)$ evaluates non-deterministically to two distinct values or that (3.1) implicitly implies that 0 is equal to $succ(succ(0))$. In the algebraic formalism this would not be an inconsistency. For example, this can be a way (a very bad one indeed!) to specify the integers modulo 2. A more appropriate specification for the integers modulo 2 should not entail any defined operation, but a quotient axiom such as:

$$succ(succ(0)) \rightarrow 0 \tag{3.2}$$

The above problem originates from the fact that we can choose between two axioms for reducing a term. Within the initial model, the only plausible discipline for dealing with choices of this kind seems to be non-determinism. In the following, we investigate a discipline for limiting the proliferation of situations in which such choices could arise. We show that such a discipline is available without altering the initial model or sacrificing a specification. This discipline is based on the observation that in the above example the axioms that we can chose have the same redex occurrence in a term and the root of this occurrence is a defined operation. This situation is characterized as over-specification which is formally defined below. The definitions of the concepts of *redex* and *match*, which is used in the next definition, can be found, e.g., in [HUET80a].

Definitions 2. An operation f is *over-specified* if there exists some ground f-rooted constructor term matched by two distinct defining axioms. By extension, an *over-specified* specification is one containing an over-specified operation.

We now introduce a few concepts which are useful to address under- and over-specification in a constructive way. We recall that for x and y in $T(\Sigma \cup \mathcal{X})$, y is an *instance* of x if there exists a substitution σ such that $\sigma(x) \equiv y$, i.e. $\sigma(x)$ and y are congruent [GOGU78, HUET80a] modulo the axioms of the specification. We say that the constructors of a specification are *free* when any two ground constructor terms are congruent if and only if they are syntactically equal.

Examples 3. Referring to Figure 1, 0 is an instance of both 0 and X with substitutions Φ (empty) and $\{0/X\}$ respectively. $succ(succ(X))$ is an instance of $succ(Y)$ with substitution $\{succ(X)/Y\}$. 0 is not an instance of $succ(X)$. However, referring to Figure 2, 0 is an instance of $succ(X)$ with substitution $\{pred(0)/X\}$.

We now introduce two key concepts of our discussion.

Definitions 4. A set R of k-tuples of constructor terms is *complete* if every k-tuple t of ground constructor terms is an instance of at least one element of R. R is *parsimonious* if t is an instance

of at most one element of R. By extension, we say that an operation f is *complete* or *parsimonious* if so is the set of tuples of terms consisting of the arguments of f in the left sides of f's defining axioms. Similarly, we say that a specification is *complete* or *parsimonious* if so are all its defined operations.

Examples 5. The set $\{\langle 0, X\rangle, \langle Y, succ(Z)\rangle\}$ is neither parsimonious nor complete, in fact, $\langle 0, succ(0)\rangle$ is an instance of both elements of the set and $\langle succ(0), 0\rangle$ is not an instance of either one. All the operations presented in the figures of this note are complete and parsimonious, unless otherwise explicitly stated in the caption.

The relationships between completeness and under-specification and between parsimony and over-specification are captured by the following lemmas, whose proofs are immediate.

Lemma 6. The completeness of a specification is a necessary condition to avoid under-specification.

Lemma 7. The parsimony of a specification is a sufficient condition to avoid over-specification.

The previous lemmas show that completeness and parsimony are relevant properties to consider during the design of algebraic specifications. Unfortunately, completeness alone is not enough to avoid under-specification. For an example, consider a binary operation $f : s \times s \rightarrow s$, for some sort s, whose only defining axiom is:

$$f(X, Y) \rightarrow f(Y, X) \tag{3.3}$$

f is clearly complete, but grossly under-specified if $T(\mathcal{C})_s$ is not empty. Furthermore, although parsimony takes care of over-specification, we will show later that also in specifications which are not over-specified there are situations with the same potential of "collapsing" a sort as done by the axiom (3.1). Sufficient conditions to avoid under-specification and/or to provide confluence need further work and will be discussed in the next sections. For the time being, we investigate how completeness and parsimony, which are necessary to avoid flaws, may be achieved in a specification.

Our definitions of completeness and parsimony are non-constructive, i.e. they do not suggest how to achieve the conditions or to test whether a specification satisfies them. Thus, we now describe a procedure for generating complete and, under some hypotheses to be discussed later, parsimonious sets of tuples. The procedure is non-deterministic, since it contains some indefinite selections and choices. Also, the termination of the procedure for arbitrary choices is not guaranteed. In practical applications, these selections and choices correspond to design decisions and the termination of the procedure is an expected consequence of the designer's strategy. $\Box$ is a distinguished symbol, called *place*, which is neither in Σ nor in $\mathcal{X}$. *Occurrences* of places, similar to variables, are sorted. In

the procedure's description, the sort of each occurrence of the initial tuple is assigned arbitrarily. In the applications of the procedure to the design of an operation, a place always occurs as an argument in a term t. If f is the root of t, then the sort of an occurrence is established by arity of f.

Binary Choice Procedure.

Input: $\mathcal{S}$: a set of sorts,
$\mathcal{C}$: a signature of constructors,
k: a non-negative integer.
Output: R: a set of k-tuples of terms in $T(\mathcal{C} \cup \mathcal{X})$.

1. [Initialize] Initialize R to a set consisting of a single k-tuple whose components are all places. Let s_i be the sort of the i-th place, for $1 \leq i \leq k$.
2. [Test] If there are no occurrences of places anywhere in R, then halt with output R.
3. [Select] Select a place in some tuple t of R. Let u be the occurrence of the selected place and s its sort.
4. [Chose] Chose either "variable" or "inductive" for the place at u.
5. [Update] If the binary choice is "variable", then replace in t the place at u with a fresh variable, else remove t from R and for each constructor c, of arity $s_1 \times \ldots \times s_n \rightarrow s$, for some $s_1, \ldots, s_n$ in $\mathcal{S}$, insert in R a tuple obtained by replacing in t the place at u with the term $c(\Box, \ldots, \Box)$ in which there are exactly n places and the sort of the i-th place, for $1 \leq i \leq n$, is s_i.
6. [Iterate] Go to step 2).

We use an implementation of the binary choice procedure to show its application to the design of the left sides of defining axioms. The procedure has been implemented, within the *GNU Emacs* editor [STAL87] in the style of [HALM88], as a template-driven interactive process. A simplified session of the design of the operation computing the length of a list is presented in Figure 5. For clarity, the definition of the right sides of the axioms is omitted from the description. This step will be completed later. The initial template is automatically generated from the domain of *length*. The constructors of *list* are derived from its declaration, see Figure 2. Our interest in the binary choice procedure is motivated by the next theorem. First, we discuss the termination of the procedure.

Lemma 8. If the number of "inductive" choices is bounded, the binary choice procedure terminates.

Proof. At every iteration of the procedure the number of occurrences of places in R is either decremented by 1 if the choice is "variable" or incremented by a finite amount if the choice is

The initial template, 1, results from the domain of *length*.

length(■) (1)

Templates 2 and 3 come from 1 with choice "inductive".

length([]) (2)

length([■|□]) (3)

Template 5 comes from 3 with choice "variable", 4 is 2.

length([]) (4)

length([X|■]) (5)

Template 7 comes from 5 with choice "variable", 6 is 4.

length([]) (6)

length([X|Y]) (7)

Figure 5. Coding of the left sides of the axioms defining the length of a list using an implementation of the *binary choice* procedure. At every iteration of the interactive process an occurrence of □, called *cursor* and denoted with ■, is selected and the user prompted for a choice. The implementation selects places in lexicographic order, but the user may override this strategy. The coding is completed in just three keystrokes and both the completeness and parsimony of the operation are guaranteed.

"inductive". If the number of "inductive" choice is bounded, eventually the number of occurrences of places in R will decrease to 0 and the procedure terminate.

Theorem 9. The output R of the binary choice procedure is complete. Furthermore, if the constructors of the specification are free, then R is also parsimonious.

Proof.

– Completeness: [HUET82] proposes an inductive, *operative* definition of completeness for linear tuples of constructor terms. R satisfies Huet's definition, hence his Lemma 3 proves that R also satisfies our definition of completeness.

– Parsimony: Let t_1 and t_2 be tuples of R with a common instance, i.e. there exist two substitutions σ_1 and σ_2 such that $\sigma_1(t_1) \equiv \sigma_2(t_2)$. If $t_1 \neq t_2$, then for some occurrence u, selected in step 3 of the binary choice procedure, the roots of t_1 and t_2 at u are different constructors, hence $\sigma_1(t_1) \neq \sigma_2(t_2)$. If the the constructors are free, $\sigma_1(t_1)$ and $\sigma_2(t_2)$ cannot be congruent, which contradicts the initial assumption. Thus, $t_1 = t_2$ and R is parsimonious.

4. Termination — Recursive Reductions

In the previous sections we have seen two substantially different cases of under-specification. In the first one, see the displays (1.1) through (1.3), some operation-rooted constructor term could not be rewritten. We have related this situation to the notion of completeness. In the second, see

the display (3.3), some operation-rooted constructor term is rewritable, but to a term which always contains a defined operation. In this section, we relate this situation to the notion of recursion.

Definitions 10. A sort s is *recursive* if there exists a constructor c such that $\tau(c) = s_1 \times \ldots \times s_n \rightarrow s$ and $s_i = s$ for some i such that $1 \leq i \leq n$. c and its i-th argument are called *recursive* too.

In other words, a sort is recursive when it is an element of the domain of one of its constructors.

Examples 11. The sorts *nat* and *list* previously presented are recursive — *succ* and the mixfix symbol "[|]" denote their respective recursive constructors. Only the second argument of "[|]" is recursive.

In order to employ recursion in the design of an axiom $\alpha \rightarrow \beta$ there must be an occurrence of a recursive constructor in α. Initially, for simplifying discussion and notation, we impose the additional condition that each recursive constructor has only one recursive argument. This restriction will be relaxed later.

Definition 12. We call *recursive reduction* the function $\rho : T(\mathcal{C} \cup \mathcal{X}) \rightarrow T(\mathcal{C} \cup \mathcal{X})$ defined as follows:

$$\rho(t) = \begin{cases} t, & \text{if } t \text{ is a variable or a constant constructor;} \\ \rho(t'), & \text{if } t = c(\ldots, t', \ldots), \text{ where} \\ & \quad c \text{ is a recursive constructor and} \\ & \quad t' \text{ is its recursive argument;} \\ c(\rho(t_1), \ldots, \rho(t_k)), & \text{if } t = c(t_1, \ldots, t_k) \text{ and } c \text{ is not recursive.} \end{cases} \tag{4.1}$$

The recursive reduction operation is not very interesting for ground terms. Intuitively, the recursive reduction of a term t "strips away" all occurences of recursive constructors in t. Each subterm of t whose root is a recursive constructor is (recursively) replaced by its recursive argument. The definition (4.1) is extended by (4.2) to include in its domain operation-rooted constructor terms, in particular, the left sides of defining axioms designed with the binary choice procedure.

$$\rho(f(x_1, \ldots, x_k)) = f(\rho(x_1), ..., \rho(x_k)) \tag{4.2}$$

We also extend with this concept the language we use for describing specifications. The symbol "!" on the right side of an axiom denotes the recursive reduction of the left side, see Figure 6.

$length([]) \rightarrow 0$	(1)
$length([X\|Y]) \rightarrow succ(!)$	(2)

Figure 6. Axioms for the operation *length* whose left sides were defined in Figure 5. The symbol "!" in line 2 denotes the *recursive reduction* of the left side and stands for $length(Y)$. The names of the variables of axiom 2 do not convey any information, since they occur in only one occurrence. Our notation makes X and Y anonymous, hence they should be replaced by an underscore symbol.

Loosely speaking, the concept of recursive reduction is complementary to that of completeness to avoid under-specification. To show this fact it is convenient to combine these two concepts in the following definitions.

Definitions 13. We call *semi-regular* any defined operation f such that: 1) f is complete, 2) any operation g different from f occuring in the right side of an axiom defining f is not under-specified, and 3) any f-rooted (sub)term of the right side of an axiom defining f is the recursive reduction of the corresponding left side. We call *regular* any parsimonious semi-regular operation. By extension, we say that a specification is *semi-regular* or *regular* if so are all its defined operations.

The following theorem is our main result. It identifies semi-regularity as a condition sufficient to avoid under-specification. The usefulness of this result stems from the fact that under-specification is undecidable (it will be proved later) and semi-regularity is automatically provided by our design strategies.

Theorem 14. Any semi-regular operation f is not under-specified.

Proof. The proof is based on structural [BURS69] or data type [GUTT78] induction on the "inductive" arguments of f. Proof sketch: the completeness of f implies that any f-rooted ground constructor term t is matched by a defining axiom of f, say $\alpha \rightarrow \beta$. There are two cases to consider. *Case 1*: There are no occurrences of f in β. Then t is rewritten to some term t' which, by the second condition of semi-regularity, is eventually rewritten to some ground constructor term. Notice that case 1 is necessarily satisfied when there are no occurrences of recursive constructors either in α or t or both. *Case 2*: There are occurrences of f in β. Then t is rewritten to a term t' in which there are some f-rooted subterms. If t'' is any such subterm, for those arguments corresponding to arguments of α whose root is a recursive constructor, the number n of nested applications of recursive constructors is less than the corresponding number for t. For all the other arguments of t'' the number of nested applications of recursive constructors remains unchanged with respect to t. t'' can be rewritten by $\alpha \rightarrow \beta$ at most n times. The resulting term, by the hypothesis of completeness, either will satisfy case 1 or will satisfy case 2 for some other arguments. Since the number of arguments of f is finite, eventually only case 1 will be applicable.

Corollary 15. A specification designed with the binary choice procedure and with recursive reductions as only recursion mechanism is neither under- nor over-specified.

Proof. By Theorem 9 the specification is complete and parsimonious, hence semi-regular. Thus, by Lemma 7 is not over-specified and by Theorem 14 is not under-specified.

When in a specification there are no quotient axioms, we believe that regularity is a formalization of the intuitive feeling that an operation is defect free. However, when there are quotient axioms this is not yet enough. In the next section we will show why and discuss what is still missing for claiming that the specification of an operation is defect free.

5. Power and Limitations of the Strategies

The design strategies we have presented in the previous sections suggest an "incremental" methodology for the design of software abstractions and their specifications. The initial step of this methodology consists in enumerating each sort of an abstraction, and for each sort its constructors and any mutual relationship existing between them. The sort statement of our small specification language, contains exactly this information. Subsequent steps of our methodology consist in enriching [GOGU78] an existing abstraction with defined operations in a "non-disruptive" way. The incrementality supported by our methodology is interesting from an engineering point of view, since at each increment we do not have to reconsider the whole abstraction. Although types are intimately bound to their operations, the addition of new operations to an existing type should not change its "essence". This consideration may be appreciated by those who feel, e.g., that instances of stacks and queues are essentially the same thing. According to our methodology, these different types are based on a single initial abstraction which captures the idea of a collection of elements in which repetitions are allowed and the order in which elements are inserted into a collection is relevant. This initial abstraction is subsequently enriched with stack- or queue-specific operations. A consequence of our incremental, non-disruptive enrichment is that if these characteristic operations are designed with our strategies, then the two abstractions could be merged together, e.g. to provide the type deque, with the guarantee that the operations of each abstraction do not need to be redesigned.

For non-trivial abstractions, the effort required for completing the initial step is generally small when it is compared with the total effort required for completing the whole abstraction. Many types involved in an abstraction are based on well-understood models, such as numbers, lists, trees, sets, etc. whose formalization is sometimes already available, e.g. consult [MANN85]. Our methodology is not concerned with the design of sorts, in particular with the quotient axioms. Notice that these axioms are a small fraction of the axioms of an abstraction and for them conditions of under- and over-specification are meaningless. However, certain characteristics of these axioms, hence of the initial specification, affect the characteristics of its enrichments; in the following paragraphs we investigate this issue. In this section, we consistently denote with $S = \langle \mathcal{S}, \mathcal{C}, \Phi, \tau, \mathcal{Q}, \Phi \rangle$ an initial

specification. In S the set of defined operations, and consequently the set of defining axioms, are both empty. An enrichment of S is denoted with $S' = \langle \mathcal{S}, \mathcal{C}, \mathcal{D}, \tau, \mathcal{Q}, \mathcal{A} \rangle$. S is extended only with defined operations and we assume that their defining axioms are designed using exclusively non-recursive functional composition and the binary choice and the recursive reduction strategies.

As a preliminary observation we notice that if $T(\mathcal{C})$ is empty, then any defined operation is neither under- nor over-specified, regardless of its defining axioms. In fact, in this circumstance, both conditions become vacuously true since there are no ground terms. This observation strengthens our belief that under- and over-specification are appropriate concepts for addressing in a constructive way the design of operations. We recall that a sort whose set of ground terms is non-empty is called *strict* and that strictness is the basis for defining *sensible* signatures [HUET80a]. Our methodology is not interesting for non-strict sorts which, on the other hand, have no applications in software engineering.

Rewriting, which is the definitional mechanism of the axioms, has two very important attributes: confluence and termination, which are briefly recalled below; for further details consult respectively, e.g., [HUET80b] and [DERS85]. A term rewriting system is *confluent* (or Church-Rosser) if any two terms with a common ancestor have a common descendant, and it is *terminating* (or Noetherian) if there are no infinite derivations. When both properties hold, the system is called *canonical* (or convergent, or complete) and has the fundamental property that there exists a unique normal form for each term.

In the following we discuss properties of S which are sufficient to guarantee either the confluence or the termination of S'. Both confluence and termination are undecidable properties, hence any attempt to determine if they hold for a given system is non-trivial. The results we are going to prove are valuable since they simplify these attempts in practical cases. In particular, the next corollary reduces the problem of the termination of a possibly large set of axioms to that of the termination of a smaller, or possibly empty, subset.

Corollary 16. If $\mathcal{Q}$ is terminating, then $\mathcal{E}$ is terminating.

Proof. By Corollary 15 the defined operations of S' are not under-specified, hence every ground term is rewritten by the defined axioms to a ground constructor term. Thus, the termination hypothesis of $\mathcal{Q}$ guarantees the termination of $\mathcal{E}$.

The termination property of the term rewriting system established by $\mathcal{E}$ should not be confused with the notion of termination of the functions that we would like to specify through the axioms. In fact, the term rewriting system resulting from Figure 3 is terminating, but we have shown that there

exists a correct implementation of this specification in which the function "$\leq$" does not terminate for some arguments. These different viewpoints are unified by the concept of under-specification. Occasionally, a design goal of an abstraction is to specify the values of some types through the terms in $T(\mathcal{C})$ and the functions manipulating values of these types through defined operations. According to this viewpoint, if an operation f is not under-specified, then it "returns" a value for any tuple of arguments, hence the function specified by f is terminating for any input in the classical sense, e.g. consult [KFOU82].

operation $integer + integer \rightarrow integer$
axioms (1)
$0 + Y \rightarrow Y$; (2)
$succ(\_)+\_ \rightarrow succ(!)$; (3)
$pred(\_)+\_ \rightarrow !$. (4)
(5)

Figure 7. Specification of a "flawed addition" for integer numbers. The sort *integer* has been defined in Figure 2. The symbol "+" is unrelated to the symbol of Figure 1 with the same name. The right side of the axiom at line (5) should be $pred(!)$.

We now turn our attention to confluence. In Figure 7 we show a non-confluent specification. The situation is analogous to that discussed in Figure 4 for the operation *max*, but does not originate from over-specification. The term $pred(succ(0)) + 0$ could be reduced in two different ways.

$$pred(succ(0)) + 0 \rightarrow \begin{cases} 0 + 0 \rightarrow 0 \\ succ(0) + 0 \rightarrow succ(0 + 0) \rightarrow succ(0) \end{cases} \tag{5.1}$$

The first derivation starts with the application of the axiom at line (8) of Figure 2. The second derivation employes only axioms of the specification presented in Figure 7. A consequence of this specification is that either the term $pred(succ(0))+0$ evaluates non-deterministically to two distinct values or $succ(0)$ must be equated to 0. Both options seem unacceptable, thus the specification should be considered flawed. Situations of this kind can be detected by a *superposition* algorithm which looks for *critical pairs* [KNUT70]. The next result relates the binary choice strategy to this concept.

Lemma 17. No critical pair originates from any two distinct defining axioms $\alpha_1 \rightarrow \beta_1$ and $\alpha_2 \rightarrow \beta_2$ of S'

Proof. If the two axioms define different operations, then the root of α_1 does not occur in α_2 and vice versa. If they define the same operation, α_1 could superimpose α_2 (and vice versa) only at the root, but the parsimony property prevents this from happening. Thus in both cases no critical pair originates from the axioms.

This result is interesting because it supports the following corollary and because it suggests a more efficient approach to detect the critical pairs and eventually determine the confluence of a system when the binary choice strategy is employed.

Corollary 18. If $\mathcal{Q} = \Phi$, then $\mathcal{E}$ is confluent.

Proof. Lemma 17 guarantees that no critical pair may originate from the left sides of two defining axioms. The absence of quotient axioms implies that there exist no critical pairs at all in the system. Thus, by Lemma 3.1 of [HUET80b] $\mathcal{E}$ is locally confluent. By Corollary 16 $\mathcal{E}$ is terminating. Thus, by Lemma 2.4 of [HUET80b] $\mathcal{E}$ is confluent.

The assumption that in a specification there are no quotient axioms, hence no quotient sorts, is sometimes too restrictive. In these situations it is necessary to check the specification for problems such as that of Figure 7. This is done by a superposition algorithm that looks for critical pairs. Let d denote the number of defining axioms of a specification and q the number of quotient axioms. If the defining axioms are designed without any restriction on their left sides, the superposition algorithm must check any pair of axioms, thus its computational complexity is $O((d+q)^2)$. However, if the defining axioms have been designed with the binary choice strategy, then Lemma 17 suggests than it is useless to check defining axioms against each other, thus the computational complexity is reduced to $O((d+q)q)$. Since d is usually much larger than q, the gain may be substantial.

In the previous paragraphs, we have discussed two fundamental properties of a specification. We have shown that if a specification S satisfies these properties, then any specification S' obtained extending S with the binary choice and recursive reduction strategies still satisfies these properties with the possible exception of confluence which, however, can be handled more efficiently. Since both these properties are undecidable, in particular we will show in the next section that under-specification is undecidable too, it is obvious that there are inherent limitations in our approach. In the remaining of this section we assess the extent of these limitations or in other words we determine what price is attached to the features we have gained.

A major feature absent in our methodology is the lack of a strategy for mutually recursive operations. Obviously, we can specify mutually recursive operations, but there will be no *automatic* guarantee that the specification is not under-specified or is terminating. We will show shortly that this is more of an inconvenience than a substantial limitation. When mutually recursive operation are specified we may resort to *ad hoc* stratagems to prove that the specification is not under-specified or is terminating. A proof of termination may be attempted, e.g., with one of the several methods discussed in [DERS85]. As far as both termination and under-specification are concerned, there

exist interesting methods and efficient procedures if the binary choice strategy is still employed in the design of the mutually recursive operations [ANTO90b].

Now we relate the notion of recursive reduction with that of primitive recursion. When recursive reductions are applied to complete operations the resulting mechanism is at least as powerful as primitive recursion. *Primitive recursion* is used to define functions such that the type of both their arguments and results is natural number. The template proposed in [MINS67, p. 175] to define primitive recursion can be trivially reformulated using our strategies. The notion of recursive reduction extends that of primitive recursion from natural numbers to abstract data types. Recursive reductions make specifications more easily understandable through the use of an elegant notation. Concerning the intrinsic limits of primitive recursion, hence the implication of losing mutually recursive operations, we quote from [GUTT78c] "In programming, the occasions when one has need of a non-primitive recursive function seem to be very rare indeed."

Our final consideration on the power and limits of the notion of recursive reduction concerns its application when there are constructors with multiple recursive arguments. In these situations the recursive reduction of a term is better described by a set of terms rather than by a single term. For an example and a plausible notation see Figure 8. The results we have proved in Section 4 still hold unchanged. The major problem is notational. As in a term t the numbers of occurrences of recursive constructors with multiple arguments increases, plausible notations for the elements of the recursive reduction of t lose part of the elegance of the single argument case.

sort *node* **parameter**; (1)
sort *bin_tree* (2)
  constructors (3)
    empty; (4)
    make(*node*, *bin_tree*, *bin_tree*). (5)
operation *leaf_count*(*bin_tree*) → *nat* (6)
  axioms (7)
    leaf_count(*empty*) → *succ*(*0*); (8)
    leaf_count(*make*(_, *L*, *R*)) → !*L*+!*R*. (9)

Figure 8. Specification of the sort modeling binary trees and the operation "count the number of leaves." The recursive constructor *make* has two arguments of sort *bin_tree*, a condition which violates an assumption made in the definition of *recursive reduction*. In this situation the left side of axiom 9 has two distinct recursive reductions. They are denoted with !L and !R which stand respectively for *leaf_count*(L) and *leaf_count*(R).

We now turn our attention to the binary choice strategy. To better understand its limits we show the existence of complete and parsimonious sets of tuples which cannot be generated by the

binary choice procedure. In order to prove our claim and to understand the characteristics of these sets a few preliminary results are needed.

Lemma 19. If $R = \{\langle x_{11}, \ldots x_{1k}\rangle, \ldots \langle x_{n1}, \ldots x_{nk}\rangle\}$, for some $k > 0$, is a set of k-tuples generated by the binary choice strategy, then either for some i, such that $1 \leq i \leq k$, and for all j, such that $1 \leq j \leq n$, the term x_{ji} is a variable, or for some i, such that $1 \leq i \leq k$, and for all j, such that $1 \leq j \leq n$, the term x_{ji} is not a variable.

Proof. The index i is the occurrence of the place selected in step 3 during the first iteration of the binary choice procedure.

We call i-th *column* of R the set of elements $\{x_{ji}\}_{1 \leq j \leq n}$. We call *pure* any column in which either all or none of the elements are variables. The previous lemma suggests then to look into sets of k-tuples without pure columns. One such set is provided by the arguments in the left sides of the axioms of the operation $\wedge$ specified in Figure 9.

operation $Boolean \wedge Boolean \rightarrow Boolean$
axioms
$true \wedge false \rightarrow false$; (1)
$false \wedge true \rightarrow false$; (2)
$X \wedge X \rightarrow X$. (3)
(4)
(5)

Figure 9. Specification of the operation $\wedge$ (logic conjunction). The sort *Boolean* is defined in Figure 3. The operation $\wedge$ has not been designed with the binary choice strategy.

Notice that the axiom at line (5) of Figure 9 is non-linear. The next result relates linearity to both completeness and parsimony and is important for understanding sets of tuples without pure columns. Through the remaining of this section we assume that sorts are always strict.

Theorem 20. If R is a complete set of k-tuples of terms in $T(\mathcal{C} \cup \mathcal{X})$, t is a non-linear element of R, and the sort of a non-linear variable of t is infinite, then R is not parsimonious.

Proof. To simplify the proof we prove the theorem for $k = 2$ and $\mathcal{S} = \{nat\}$ (see Figure 1) and address the problem of its generalization later. For any term x, the expression $succ^i(x)$, where i is a non-negative integer, stands for x when $i = 0$ and for $succ(succ^{i-1}(x))$ otherwise. Any instance of *nat* is representable by $succ^i(x)$ where either $x = 0$ or $x = X$ for some non-negative integer i and variable X. If R is complete, then, since *nat* is infinite, by simple case analysis, R must contain some element t' of the form $\langle succ^{v_1}(X), succ^{v_2}(Y)\rangle$ for some v_1 and v_2 and variables X and Y, with $X \neq Y$. If t is non-linear, then $t = \langle succ^{u_1}(X), succ^{u_2}(X)\rangle$ for some u_1, u_2, and X. The pair $\langle succ^{max(u_1,v_1)}(0), succ^{max(u_2,v_2)}(0)\rangle$ is an instance of both t and t', hence R is not parsimonious.

When the proof is generalized, the notation becomes rather cumbersome since the sort of a non-linear variable may have more than one recursive constructor and these may have in turn more than one argument, thus the simple notation $succ^i$ cannot be used. Apart from this, no other substantial difficulty arises. The hypothesis that the sort of one non-linear variable is infinite is a necessary one as Figure 9 proves.

The next display shows a set R of 3-tuples in which the sort of each component is *nat*. The presentation in matrix form facilitates the analysis of the columns of R. It is easy to prove by case analysis that R is complete and parsimonious. Since there are no pure columns, R cannot be generated by the binary choice procedure.

$$\begin{aligned} R = \{ & \langle\, succ(X)\,, \quad 0 \quad\,, succ(Z)\,\rangle \\ & \langle \quad 0 \quad\,, succ(Y)\,, \quad 0 \quad\,\rangle \\ & \langle \quad 0 \quad\,, \quad 0 \quad\,, \quad Z \quad\,\rangle \\ & \langle \quad X \quad\,, succ(Y)\,, succ(Z)\,\rangle \\ & \langle\, succ(X)\,, \quad Y \quad\,, \quad 0 \quad\,\rangle\,\} \end{aligned} \tag{5.2}$$

The fact that R has at least 3 columns is necessary. We elaborate on this in the next corollary. Notice that the hypothesis of linearity in the statement of the corollary is automatically implied by those of completeness and parsimony (Theorem 20) for infinite sorts.

Corollary 21. Any complete and parsimonious set R of linear k-tuples of terms in $T(\mathcal{C} \cup \mathcal{X})$, for $k = 1$ or $k = 2$ and any set of constructors $\mathcal{C}$, has a pure column.

Proof. The proof is trivial for $k = 1$, thus let $R = \{\langle x_{11}, x_{12}\rangle, \ldots \langle x_{n1}, x_{n2}\rangle\}$ and $n > 1$ (otherwise the claim is trivially satisfied). Let $\bar{x}_m$, for $1 \leq m \leq n$, denote the m-th 2-tuple of R. Suppose that neither column of R is pure. Then, there exist two indices, i and j, such that x_{i1} and x_{j2} are variables. If $i \neq j$, then $\langle x_{j1}, x_{i2}\rangle$ is an instance of both $\bar{x}_i$ and $\bar{x}_j$. If $i = j$, then, since $x_{i1} \neq x_{j2}$, any 2-tuple of R is an instance of $\bar{x}_i$. In any case R is not parsimonious, thus at least one column of R must be pure.

The previous corollary is the basis for proving, by case analysis, that when $k = 1$ or $k = 2$ any complete and parsimonious set of k-tuples of constructor terms of sort *nat* is generated by the binary choice procedure. A simple generalization of this statement to any sort is not possible. In fact, Corollary 21 considers only the "outer" structure of the set. Even in a set of 1-tuples we can recreate conditions similar to those of (5.2), if in the specification there is some constructor which takes 3 arguments of sort *nat*.

The above results do not support a definite conclusion on the limitations of the binary choice strategy. While in some simple situations this strategy is substantially the only approach to completeness and parsimony, in slightly more complex situations other alternatives seem available. We have not found any circumstance in which the binary choice procedure did not appear to be a natural and effective approach for designing the left sides of an operation's axioms. Our experience seems to indicate that in the situations in which the initial model is appropriate, the circumstances in which the binary choice procedure is inappropriate should be quite scarce.

6. Related work and concluding remarks

The notion of completeness is proposed in [HUET82] through an inductive definition for testing whether a set of linear tuples is complete for a set of constructors. Completeness is a condition supporting a principle of definition assumed by an "inductionless" method of proof by induction. [THIE84] and [KOUN85] extend this definition to sets of non-linear tuples. However, both papers seem unaware of the relationship between completeness, parsimony, and linearity that we stated in Theorem 20 and hence the limitation of this extension. These papers are focused on detecting lack of completeness and/or parsimony; non-parsimonious sets, called ambiguous in [THIE84] and redundant in [KOUN85].

The question of deciding whether a set of axioms indeed defines an operation has been repetedly addressed in the literature. [HUET82] proposes a *Principle of Definition* based on equations. [KAPU87] reformulates for rewrite rules a notion of *sufficient-completeness* originally proposed by [GUTT78c]. In the following, we compare both these conditions to under-specification. Our comparison is performed in the framework of rewriting. Accordingly, we interpret the congruence relation of the principle of definition to be the derivation relation established by $\mathcal{E}$, see [HUET82] section 2. The simplest characterization of under-specification, we believe, is offered by the following theorem.

Theorem 22. If a specification S satisfies Huet's Principle of Definition, then S is not under-specified, and if S is not under-specified, then S is sufficiently-complete.

Proof. The above conditions are all centered on how the terms of a specification are related to constructor terms. Let t be a term of the specification S. If S satisfies the principle of definition, then the normal form of t is a constructor term, hence S is not under-specified. If S is not under-specified, then for some constructor term c we have $t \xrightarrow{*} c$, hence $t \overset{*}{\leftrightarrow} c$ and S is sufficiently complete

Corollary 23. The under-specification property of an algebraic specification is undecidable.

Proof. [KAPU87] proves in Theorem 3.3 that the sufficient completeness property of a term rewriting system is undecidable. Hence, our corollary is an immediate consequence of the previous theorem.

The principle of definition is a global property of a specification, i.e. it involves simultaneously constructors and operations. Under-specification is a local property of a single operation, hence it is much simpler to handle and as a consequence it can be constructively addressed during the design phase of each operation. We do not think that sufficient completeness is an adequate property from a software engineering standpoint. Figure 10 rephrases an example presented in [KAPU87]. This specification is sufficiently complete, but clearly under-specified. An equivalent, but much simpler specification of the operation g consists of the following single axiom.

$$g(X) \rightarrow c \tag{6.1}$$

With this specification, the defining axioms at lines (4) and (5) of Figure 10 become theorems which are easily provable from (6.1).

sort s **constructors** c.
operation $g : s \rightarrow s$ (1)
axioms (2)
$g(g(X)) \rightarrow g(X)$; (3)
$g(g(X)) \rightarrow c$. (4)
(5)

Figure 10. The following specification [KAPU87] is sufficiently complete, but under-specified.

Assumptions on certain characteristics of algebraic specifications are found frequently in the literature. This occurs nearly always when specifications are involved in some process which requires the interpretation, compilation, or implementation of a specification. For example, consult recent efforts [EMDE87, TOGA87] directed toward the integration of functional and/or equational programs into logic programs. The conditions often required in an specification to be implemented originate from the desire of a precise separation between the concepts of value and computation, for a discussion see [ANTO90a], so that a computation can always return a value for any legal input. In particular, for the application of a translation scheme from axioms to logic programs [EMDE87] requires a term rewriting system to be canonical (or *strongly-canonical*) and [TOGA87] requires any computation to be *successfully terminating*. In both cases the question of whether a specification is indeed adequate for the application for which it is considered is left unanswered. Although quite disturbing, this is only mildly surprising since these questions are undecidable.

A more pragmatic approach is taken by the authors of OBJ which we quote below: "we have found that experienced programmers usually write rules that satisfy these properties [Church-Rosser and termination]" [FUTA85], and "We conjecture that users almost always write equations for abstract data types that are canonical, because they tend to think of equations as programs, and then write primitive recursive definitions for operations" [GOGU88].

Our work is an attempt to address this unsatisfying state of things on two fronts: 1) operations designed with our strategies satisfy each of the requirements mentioned above, thus our effort is somewhat complementary to that of other researchers, and 2) our approach improves the design of certain specifications by simultaneously increasing the reliability of the result and decreasing both the skill of the designer and the time for the design, three conditions which are usually pair-wise conflicting.

References

[ANTO89] Antoy S., "Systematic Design of Algebraic Specifications", 5th Int'l Workshop on Software Specification and Design, Pittsburg, PA, May 19-20, 1989, 278-280.

[ANTO90a] Antoy S., P. Forcheri, and M.T. Molfino, "Specification-based Code Generation", 23rd Hawaii Int'l Conf on System Sciences, Kona, Hawaii, Jan. 3-5, 1990 (to appear).

[ANTO90b] Antoy S., "A Decision Procedure for Proving Termination and Other Properties of Algebraic Operations", *Virginia Tech Technical Report*, 1990, in preparation.

[BURS69] Burstall L., "Proving Properties of Programs by Structural Induction", *Computer Journal*, **21**-1, 1969, 41-48.

[DERS85] Dershowitz N., "Termination", Proc. Rewriting Techniques and Applications, Dijon, France, Springer-Verlag, May 1985, 180-223.

[KFOU82] Kfoury A., R. Moll, and M. Arbib, A Programming Approach to Computability, Springer-Verlag, New York, NY, 1982.

[EMDE87] van Emden M. and K. Yukawa, "Logic Programming with Equations", *The Journal of Logic Prog.*, 4, 265-288, 1987.

[GOGU78] Goguen J.A., 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 4, 80-149 (Ed. R.T. Yeh), Prentice-Hall, Englewood Cliff, NJ, 1978.

[GOGU88] Goguen J.A. and T. Winkler, "Introducing OBJ3", *SRI-CSL-88-9*, SRI International, Menlo Park, CA, 1988

[GUTT78a] Guttag J.V., E. Horowitz, and D. Musser, "The Design of Data Type Specifications", in Current Trends in Programming Methodology 4, 60-79 (Ed. R.T. Yeh), Prentice-Hall, Englewood Cliff, NJ, 1978.

[GUTT78b] Guttag J.V., E. Horowitz, and D. Musser, "Abstract Data Types and Software Validation", *CACM*, **21**, 1978, 1048-1064.

[GUTT78c] Guttag J.V. and J.J. Horning, "The Algebraic Specifications of Abstract Data Types", *Acta Informatica*, **10**, 1978, 27-52.

[HALM88] Halme H. and J. Heilanen, "GNU Emacs as a Dynamically Extensible Programming Environment", *Software—Practice and Experience*, **18**-10, 1988, 999-1009.

[HOPC79] Hopcroft J.E. and J.D. Ullman, Introduction to Automata Theory, Languages, and Computations, Addison-Wesley, Reading, MA, 1979.

[HUET80a] Huet G. and D. Oppen, "Equations and rewrite rules: A survey", in Formal Language Theory (R. Book, ed.), Academic Press, 1980, 349-405.

[HUET80b] Huet G., "Confluent Reductions: Abstract Properties and Applications to Term-Rewriting Systems", *JACM*, **27**, 1980, 797-821.

[HUET82] Huet G. and J.-M. Hullot, "Proofs by Induction in Equational Theories with Constructors", *JCSS* **25**, 1982, 239-266.

[KAPU87] Kapur D., P. Narendran, and H. Zhang, "On Sufficient-Completeness and Related Properties of Term Rewriting Systems", *Acta Informatica*, **24**, 1987, 395-415.

[KNUT70] Knuth D.E. and P.B. Bendix, "Simple Word Problems in Universal Algebras", in Computational Problems in Abstract Algebras, (J. Leech, ed.), Pergamon Press, New York, 1970, 263-297.

[KOUN85] Kounalis E., "Completeness in Data Type Specifications", EUROCAL '85, LNCS **204**, 1985, 348-362.

[MANN85] Manna Z. and R. Waldinger, The Logical Basis for Computer Programming, Addison-Wesley, Reading, MA, 1985.

[MINS67] Minsky M. L., Computation: Finite and Infinite Machines, Prentice-Hall, Englewood Cliffs, NJ, 1967.

[STAL81] Stallman R., "Emacs: the Extensible, Customizable, Self-documenting Display Editor", Proc. ACM SIGPLA/SIGOA Symp. on Text Manipulation, Portland, OR, 1981.

[STAL87] Stallman R., *GNU Emacs Manual*, Sixth ed., Version 18, Free Software Foundation, Cambridge, MA, 1987.

[THIE84] Thiel J.J., "Stop Losing Sleep over Incomplete Data Type Specifications", in 11th Annual Symp. on Principles of Prog. Languages, 76-82, ACM, 1984.

[TOGA87] Togashi A. and S Noguchi, "A Program Transformation from Equational Programs into Logic Programs", *The Journal of Logic Prog.*, 4, 85-103, 1987.

Appendix

Example 24.

This example concerns the specification of the Ackermann's function. We show how to design the left sides of the specification axioms using the binary choice procedure. Since this function is not primitive recursive, recursive reductions are not appropriate for the axioms' right sides. The Ackermann's function has exactly two arguments of sort *nat*, thus any complete and parsimonious specification of this function has to be designed with the binary choice or an equivalent procedure. In the following axioms we abbreviate $succ^i(0)$ with i

The arity of the operation *acker* is $nat \times nat \rightarrow nat$. The initial template is

$acker(\blacksquare,\square)$ (1)

With choice "inductive" we obtain

$acker(0,\blacksquare)$

$acker(succ(\square),\square)$ (2)

Now we choose "variable" for the cursor, we define the left side, and we select the next place for the cursor. We obtain (3)

$acker(0,Y) \rightarrow 1$

$acker(succ(\square),\blacksquare)$ (4)

With choice "inductive" we obtain (5)

$acker(0,Y) \rightarrow 1$

$acker(succ(\blacksquare),0)$ (6)

$acker(succ(\square),succ(\square))$ (7)

Now we choose "inductive" and we define the left side. We obtain (8)

$acker(0,Y) \rightarrow 1$

$acker(1,0) \rightarrow 2$ (9)

$acker(succ(succ(\blacksquare)),0)$ (10)

$acker(succ(\square),succ(\square))$ (11)

All the remaining selections are in lexicographic order and the choice is "variable" in each case. As soon as there are no places in the left side of an axiom we define the right side. We obtain (12)

$acker(0,Y) \rightarrow 1$

$acker(1,0) \rightarrow 2$ (13)

$acker(succ(succ(X)),0) \rightarrow X+4$ (14)

$acker(succ(X),succ(Y)) \rightarrow acker(acker(X,succ(Y)),Y)$ (15)

This specification is substantially identical to a set of rules proposed in [HOPC79, p. 175] for the definition of the Ackermann's function. Notice that the completeness and parsimony of the original definition as well as of this specification are not immediately apparent. (16)

Example 25.

This example concerns the car pool of a car rental agency and the determination of the cars available in the pool. A car is specified by a tuple of attributes. To keep the example manageable we consider two attributes only: a generic *id*, which is left unspecified, and a *state*, which is either *avail*, *rented*, or *broken*. Cartesian product sorts, i.e. sorts whose elements are tuples, have only one constructor. By convention, we use the same identifier for the sort and its constructor, since they never occur in the same context.

sort *id* **parameter.**

sort *state* **constructors** *avail*; *rented*; *broken*. (1)

sort *car* **constructors** $car(id, state)$. (2)

(3)

The set of cars in the pool is described by the sort *pool*. The first quotient axiom specifies that the order in which cars are added to the pool is irrelevant. The second quotient axiom specifies that there are no duplicate cars in the pool. Notice that the quotient axioms of *pool* are non-terminating.

sort *pool*

constructors (4)

nocars; (5)

$add(pool, car)$; (6)

quotient (7)

$add(add(X,Y),Z) \rightarrow add(add(X,Z),Y)$; (8)

$add(add(X,Y),Y) \rightarrow add(X,Y)$. (9)

(10)

We imagine that, during time, cars are added to or removed from the pool and that their states are occasionally updated. Now we design an operation for counting how many available cars there are in the pool. The arity of *count* is $pool \rightarrow nat$. The initial template is

$count(\blacksquare)$ (11)

We choose "inductive" and define the first right side. We obtain

$count(nocars) \rightarrow 0$

$count(add(\blacksquare,\square)$ (12)

With choice "variable" we obtain (13)

$count(nocars) \rightarrow 0$

$count(add(X,\blacksquare)$ (14)

With choice "inductive" we obtain (15)

$count(nocars) \rightarrow 0$

$count(add(X, car(\blacksquare,\square))$ (16)

Since the sort *id* is a parameter, the choice is automatically "variable". (17)

$count(nocars) \rightarrow 0$

$count(add(X, car(Y,\blacksquare))$ (18)

(19)

With choice "inductive" we obtain

$$count(nocars) \rightarrow 0$$

$$count(add(X, car(Y, avail)) \tag{20}$$

$$count(add(X, car(Y, rented)) \tag{21}$$

$$count(add(X, car(Y, broken)) \tag{22}$$

We now complete the right sides using recursive reductions (23)

$$count(nocars) \rightarrow 0$$

$$count(add(X, car(Y, avail)) \rightarrow succ(!) \tag{24}$$

$$count(add(X, car(Y, rented)) \rightarrow ! \tag{25}$$

$$count(add(X, car(Y, broken)) \rightarrow ! \tag{26}$$

Finally, notice that the variable names occur in only one occurrence, thus they serve no purpose. (27)
All the variables could be as well anonymous.