Specification-based Code Generation

Sergio Antoy†
Virginia Tech
Northern Virginia Graduate Center
2990 Telestar Ct.
Falls Church, VA 22042

Paola Forcheri and Maria Teresa Molfino
Consiglio Nazionale delle Ricerche
Istituto per la Matematica Applicata
via L.B. Alberti 4
16132 Genova, Italy

Abstract: Algebraic specifications of the highest quality can be efficiently obtained when
design strategies are employed. An implementation of a specification of this kind can be
generated directly from a procedural interpretation of the specification’s axioms. The com-
bination resulting from design strategies and the direct implementation of specifications
provides a powerful tool for software construction and rapid prototyping. Code generated
in this way may be used as that traditionally implemented, and although it is occasionally
inefficient, it is obtained quickly and inexpensively, it is guaranteed to have all the nice
properties of the specification, and it is very modular and reusable. We recall strategies
appropriate for designing algebraic specifications to be directly implemented and discuss
problems and solutions concerning code generation. We focus on mapping the specifica-
tion to several hosting environments, representative of different models of computation,
and on integrating this implementation with the environment’s programming language.
Several examples are presented and key components of a prototypical implementation are
outlined.

1 Introduction

The traditional approach to specification-based software con-
struction follows, with some variations, a pattern consisting of
four major phases: specifications are first designed from the re-
quirements and then implemented in a target language, then the
efficiency of this implementation with respect to the specific-
ation is verified, either formally or informally, and finally the
implementation is executed to check the appropriateness of the
solution proposed by the specification to solve a given problem.
Typically, these four steps are iterated to take into account the
changes in the solution resulting from a better understanding of
both the problem and the consequences of a tentative solution.
Various attempts at improving the above process have been sug-
ggested. In [FUTA84] specifications can be executed in a stand-
alone or LISP-like environment so that certain defects in the
solution can be discovered early. In [GUTT85] specifications are
specified in a form suitable of a sophisticated, formal, static seman-
tic analysis so that certain flaws in a specification can be detected
without testing. Other efforts, including [ANTO87], [EMDE87],
[ALO87] and [KLA89], bypass the second and third phases
by directly implementing the specification, i.e. by automatically
generating from it executable source code in some conven-
national programming language (respectively Pascal, Prolog, C,
and Modula 2.) With this approach, the adequacy of a speci-
fication is investigated through the fourth phase and its efficient
implementation and the implementation verification are delayed
until sufficient confidence in the specification and its correspond-
ing solution has been accumulated.

Recent results [ANTO88a] have shown the existence of
strategies for producing algebraic specifications more efficiently
than previously done. Furthermore, abstractions designed using
these strategies have desirable characteristics which make them
easier to understand, maintain, implement, and reuse. These
strategies increase both productivity and product reliability, two
usually conflicting goals. Design strategies for algebraic specifica-
tions offer interesting possibilities for both software construction
and rapid prototyping. In this note we investigate how to exploit
some of these possibilities. Our approach, similar to [FUTA84]
and [GUTT85], emphasizes the importance of the validity of a
specification. However, our strategies allow us to obtain specifi-
cations which are valid a priori, i.e. through a design methodology,
rather than a posteriori, i.e. through a process of validation or
testing applied to specifications only after they have been de-
signed. A thorough investigation of the power and limitations of
these strategies can be found in [ANTO88c].

A non-negligible consequence of the use of the strategies is
that specifications so designed have a procedural interpretation
and consequently they can be directly translated into executable
code. In our investigation, we are concerned with mapping the
underlying model of computation of an algebraic specification to
various models of computations. Specifically, we investigate cer-
tain characteristics which unify both a number of problems arising in the direct implementation of algebraic specifications and the use of the corresponding code from different hosting environments. The hosting environments are provided by three programming languages: Common LISP [STEE84], Prolog [CLOCS81], and C [KERN78] respectively representative of a functional, a logic, and an imperative model of computation.

There are two significant aspects in our approach: design and mapping. Using an appropriate methodology, specifications of the highest quality can be designed with comparatively modest efforts and skills. Although specifications are inherently declarative, there is a model of computation—a term rewriting system [HUET80b]—in which specifications can be interpreted, rather naturally, as programs. Mapping this model to other models of computations offers the possibility of obtaining code which is easily embeddable into a variety of environments developed by a variety of means. This code supports the complete and natural access to the types and computations abstractly specified and although it is occasionally inefficient, it is obtained quickly and inexpensively, it conserves all the nice properties of the specification (e.g. computations are complete and terminating), it is easily maintained through the specifications from which it has been generated, it is reusable across a variety of environments, and it is highly modular, thus, individual components can be replaced with more efficient versions as the need commands.

This paper is organized as follows. First, we introduce a very small specification language that both supports the use of our design strategies and facilitates the perception and understanding of the various components of a specification. Then, we briefly recall the binary choice and the recursive reduction strategies which can be employed during the design of an algorithm expressed in our small language. These strategies are the basis for rapid and very reliable abstraction design. Finally, we propose a family of translation schemes for obtaining, from algebraic specifications which possess the properties achieved via our design strategies, executable code in one of several programming languages representative of different models of computations. Many aspects of our approach can be mechanized and we will outline prototypical implementations of key steps.

2. A small specification language

In this section we introduce a language for presenting specifications. The language simply provides syntactic sugar around the core components of a specification. We follow the semantic model of specification called initial [GOGU78] and we assume that the reader has some familiarity with it. The core components of this model are: a set of sort symbols, which roughly correspond to the types of an abstraction, a signature, i.e., a set of function symbols together with their arities, which roughly correspond to the routines of the abstraction, and a set of axioms which give meaning to the function symbols. With a leap of faith, the axioms correspond to the bodies of the functions of the abstraction. The signature is partitioned into constructors and defined operations. Although this partition is somewhat arbitrary, it has a significant influence on our approach. The rationale behind this partition and how it affects our treatment will be further discussed later.

The language has only two statements: one for specifying sorts and the other for specifying operations. Statements are terminated by a period. Sorts are specified through their constructors. The domain of a constructor is declared jointly with the constructor. The range of a constructor is obviously the sort in which it is declared and does not need to be specified. Referring to Figure 1, nil is a constant constructor of list and cons is a constructor of list which takes two arguments: the first of sort element (a parameter declared in line 1) and the second of sort list.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>sort</td>
<td>element parameter.</td>
</tr>
<tr>
<td>sort</td>
<td>list</td>
</tr>
<tr>
<td>constructors</td>
<td></td>
</tr>
<tr>
<td>nil;</td>
<td></td>
</tr>
<tr>
<td>cons: element × list.</td>
<td></td>
</tr>
<tr>
<td>operation</td>
<td>sis : list → list</td>
</tr>
<tr>
<td>axioms</td>
<td></td>
</tr>
<tr>
<td>sis(nil) → nil;</td>
<td></td>
</tr>
<tr>
<td>sis(cons(E, L)) → put.in.place(E, sis(L)).</td>
<td></td>
</tr>
</tbody>
</table>

Figure 1. Specification of the generic abstract data type list and the operation sis which defines a "straight insertion sorting" algorithm. The operation put.in.place, which is not shown here, inserts its first argument, an element, at the "right place" of its second argument, a list.

Operation specifications consist in a declaration of the operation arity, in standard mathematical notation, and a set of defining axioms. Referring to Figure 1, sis is an operation whose domain and range are both list. The keyword parameter, line 1, introduces a parameter sort. Parameter operations can be specified similarly as long as they do not change within the extent of an operation. Following Prolog's conventions, variables in the axioms are denoted by capitalized identifiers or the underscore symbol which stands for an anonymous variable. The small language is strongly typed, however, type declarations for variables are neither necessary nor convenient for reasons which will become clear later.

Our specification language has been restricted to the bare minimum necessary to present our ideas so that essential concepts and their implications can be investigated more easily. Additional features, marginal to the theme of the direct implementation, such as quotient sorts [GOGU78], exceptionally terminating computations, and infix and "mixfix" [FUTA84, GUTT85] notation for operations, are further discussed in [ANTO89c]. The direct implementation of algebraic specifications drawn in an extended language which includes the above features is substantially similar to the one described in the remainder of this paper.

3. Completeness and Parsimony — Binary Choices

In this section we introduce our first design strategy. Through this strategy, one side of the defining axioms, conventionally the left one, can be specified exclusively with a handful of binary choices. We have implemented this strategy as a template-driven interactive process. In this implementation a choice is made with just a single keystroke. The strategy not only speeds up and simplifies one significant aspect of the design of a specification, but also contributes to certain characteristics...
of completeness and parsimony without which a specification is under- and/or over-specified [ANT089a], and hence not easily implementable.

We call pattern a tuple of arguments in the left sides of an axiom defining an operation. Referring to Figure 1, the set of patterns of the operation \textit{sis} is \{\textit{nil}, \textit{cons}(E, L)\}. The binary choice procedure operates on these patterns. When an operation \textit{f} is applied to some tuple of ground arguments, the axioms which could be used to reduce the resulting term depend on these patterns. For the direct implementation of an operation \textit{f} it is crucial that one and only one axiom is applicable to any tuple of ground arguments. We say that an operation is under-specified when there is no axiom applicable to some tuple of ground arguments and we say that an operation is over-specified when more than one axiom is applicable. If an operation \textit{f} is under-specified, then the implementation of \textit{f} is undefined for some input. If \textit{f} is over-specified, then the implementation of \textit{f} is multiply defined, possibly with distinct values, for some inputs. The binary choice procedure removes these potential problems by generating, under the designer direction, a set of patterns which is guaranteed to make an operation complete and parsimonious [ANT089a].

The input of the binary choice procedure is the arity of a function \textit{f}. The output is the set \textit{S} of the left sides of the defining axioms of \textit{f}. It is also assumed that the definition of the sorts of the specification is accessible to the procedure. The binary choice procedure presented below is non-deterministic, since some selections and choices are left indefinite, and possibly it is non-terminating. In the application of the procedure to the design of an operation, these selections and choices correspond to design decisions, and the termination is guaranteed if the number of "inductive" choices in step 4 is bounded. \textit{S} is a distinguished, sort-overloaded symbol, called place, which is used as a placeholder for the choices that must be made. Occurrences of places are sorted, the sort of an occurrence being established by the arity of the symbol of which the occurrence is an argument.

### Binary Choice Procedure

1. \textbf{Initialize} Initialize \textit{S} to \{\textit{nil}, \ldots, \textit{cons}(\ldots)\}.
2. \textbf{[Halt if done]} If there are no occurrences of places anywhere in \textit{S}, then halt with output \textit{S}.
3. \textbf{Select} Select a place in some term \textit{t} of \textit{S}. Let \textit{u} be the occurrence of the selected place and \textit{u} its sort.
4. \textbf{Choose} Choose either "variable" or "inductive" for the place at \textit{u}.
5. \textbf{Update} If the binary choice is "variable", then replace \textit{t} the place at \textit{u} with a fresh variable, else remove \textit{t} from \textit{S} and for each constructor \textit{c} of arity \textit{a}_1 \times \ldots \times \textit{a}_n \rightarrow \textit{s}, insert in \textit{S} a tuple obtained by replacing \textit{t} the place at \textit{u} with the term \textit{c}(\textit{u}, \ldots, \textit{u}) in which there are exactly \textit{n} places.
6. \textbf{Iterate} Go to step 2.

The binary choice procedure has been implemented as a template-driven interactive process. The hosting environment is provided by the GNU Emacs [STAL87] editor. The procedure is used for designing the left sides of defining axioms. A simplified session of the design of the operation computing the length of a list is presented in Figure 2. For clarity, the definition of the right sides of the axioms is omitted from the description. The example will be completed later. The initial template is automatically generated from the domain of length. The constructors of list are derived from the definition of Figure 1. The backtracking capabilities of Emacs make extremely simple the recovery from wrong selections and choices.

The left sides of axioms designed with the binary choice procedure are linear terms, i.e. their variables occur in only one occurrence. This does not seem a practical limitation, since non-linearity is compatible with completeness and parsimony only for finite sorts [ANTOS85]. The scope of a variable is limited to the axiom in which it occurs, thus the sort of a variable is precisely and uniquely established by its occurrence in the left side of its axiom and does not need to be declared. Explicit type declarations for variables outside the axioms, such as those in Larch [GUTT85], Ob2 [FUTH84], or LIL [GOGU88], are not necessary in our specification language, are potential sources of inconsistencies, and seem to imply broader scopes for variables.

### 4. Termination — Recursive Reductions

In this section we introduce our second design strategy. Through this strategy, one major definitional device of algebraic specifications, i.e. recursion, can be easily denoted and safely applied. Safety is a concern since an incorrect use of recursion may cause either non-termination [DERS85] or under-specification and hence specifications which are not meaningful to implement.

A sort \textit{s} is recursive if at least one of its constructors has at least one argument of sort \textit{s}. For example, the symbol \textit{cons} of Figure 1 is the recursive constructor of list since its second argument is of sort list. Two conditions are necessary for applying recursion in an axiom defining an operation \textit{f}: 1) the sort of at least one argument of \textit{f} is recursive, and 2) in the left side of the axiom the leading symbol of some argument is a recursive constructor. For simplifying discussion and notation, we impose the additional condition that each recursive constructor of sort \textit{s} has only one argument of sort \textit{s}. This condition is informally relaxed in Figure 4.
The input of the recursive reduction procedure is the left side of an axiom. The output is a copy of the input in which any subterm with a recursive constructor as leading symbol is recursively replaced by its recursive argument. Again we assume that the definitions of the sorts of the specification are accessible to the procedure. The termination of the procedure is trivial to prove. The non-determinism of the selection in step 3 simplifies the description; the order in which recursive constructors are selected does not affect the output.

Recursive Reduction Procedure.
1. [Initialize] Initialize $t$ to $f(x_1, \ldots, x_n)$, the left side of an axiom.
2. [Halt if done] If there are no occurrences of recursive constructors in $t$, then halt with output $t$.
3. [Select] Select an occurrence $u$ in $t$ such that the subterm of $t$ at $u$ is $c(y_1, \ldots, y_m)$ and $c$ is a recursive constructor.
4. [Update] Replace the subterm of $t$ at $u$ with $y_i$, where $i$ is the recursive argument of $c$.
5. [Iterate] Go to step 2.

The recursive reduction of the left side of an axiom occurs frequently in specifications, thus we extend our notation with this concept. The symbol "$t$" on the right side of an axiom denotes the recursive reduction of the left side, see Figure 3. Both the notation and the properties of recursive reductions are very valuable. The notation makes design decisions explicit and specifications faster to write and read. More important, if the right sides of the axioms of a specification are designed using only non-recursive functional composition and recursive reductions, then all operations are guaranteed to terminate for any input [ANTO89a]. It is well-known that termination is an undecidable property, thus any attempt at its verification in practical cases is non-trivial.

| Figure 2. Axioms for the operation length whose left sides were defined in Figure 2. The symbol "$t$" in line 2 denotes the recursive reduction of the left side and stands for $\text{length}(Y)$. The names of the variables of axiom 2 do not convey any information, since they occur in only one occurrence, and could be omitted entirely. 0 and $\text{succ}$ are the constructors of the sort "natural number" which is not shown here. |
|---|---|
| $\text{length}(\text{nil}) \rightarrow 0$ | (1) |
| $\text{length}(\text{cons}(X,Y)) \rightarrow \text{succ}(!)$ | (2) |

The notation of recursive reduction occasionally makes naming variables unnecessary. For example, this happens for the recursive axiom of $\text{length}$, and for one variable in the recursive axiom of $\text{sin}$, see Figure 1, line 9. The use of anonymous variables not only speeds up defining specifications, but also makes them more readable, since it hides irrelevant information.

5. Translation schemes
In the previous sections we have shown how certain types and computations can be conveniently specified in a small specification language. In addition to the guarantee that specifications have certain desirable properties we have shown how two design strategies substantially reduce the effort, time, and skill required in the design phase of the specification. The contribution of this paper consists in taking advantage of these results by automatically generating an implementation of these specifications.

Our goal consists in providing the capability of using, in a simple and natural way, abstractly specified objects, i.e. sorts, sort instances, and operations, as concrete entities (respectively types, values, and subprograms) within programs developed by whatever means. An example of use is shown in Figure 5; sort symbols of the specification become types of the implementation and function symbols become function identifiers. Our approach consists in a source-to-source translation from specifications written in the small language to modules of a target programming language. These modules, which consist of type declarations and/or subprograms, can be processed, i.e. compiled, linked, included, etc., as manually written code would be. "Subprogram" is a generic name for a computing entity which, depending on the target language, corresponds to a function or a predicate or a procedure etc.

```c
#include "list"  
list x, y;  
  ;
  y = \text{cons}(n,x);  
  ;
  l = \text{length}(y);  
```

Figure 5. Use, from a C environment, of a directly implemented specification. $\text{list}$ and $\text{cons}$ are specified in Figure 1 and $\text{length}$, in Figures 2 and 3. The direct implementation consists of executable source code, which is compiled into a library to be loaded with the program, and an external interface which is to be included, see line 1, in the hosting source code. Declarations, construction of a list instance, and an operation on a list instance are shown in lines 2, 3 and 4 respectively. The use of this abstraction from other environments is similar, although, interface and declarations are unnecessary in LISP and Prolog environments and the syntax is obviously different.

The direct implementation of a specification is a process conceptually similar to the implementation traditionally done by a
programmer. The adjective "direct" means that this implementation is based on a \textit{procedural interpretation} of the algebraic axioms and can be obtained automatically. The procedural interpretation of the axioms is provided by the rewriting of instances of the left sides of axioms to their corresponding right sides. This model of computation is called \textit{term rewriting system}. Not every term rewriting system is suitable for direct implementation. The term rewriting systems we obtain through our design strategies are \textit{canonical}, i.e. Noetherian and \textit{confluent} [HUET80b]. In canonical term rewriting systems every term has a unique \textit{normal form}, i.e. a representation that cannot be further reduced by the specification axioms. In some sense, the normal form of a term $t$ is its value. According to this viewpoint, we regard a term $t$ which is not in normal form as the invocation of a computation whose result is the normal form of $t$.

As an example, consider again Figure 1 and assume that the parameter \textit{element} is bound to the integers and \textit{put.in.place} specifies the insertion, for ascending values, of integers in a list. $s(	ext{cons}(2, \text{cons}(1, \text{nil})))$ is in normal form. If it can be reduced, by axiom (9), to $s(	ext{cons}(2, \text{cons}(1, \text{nil})))$ and this can be further reduced. Thus, $s(	ext{cons}(2, \text{cons}(1, \text{nil})))$ is the invocation of a computation. The result of this computation, if $\text{put.in.place}$ is properly specified, is the value $\text{cons}(1, \text{cons}(2, \text{nil}))$. Thus, $s$ sorts its argument.

This relatively simple model of computation relies on the existence and uniqueness of a normal form for each term. Notice that both Noetheriánity and confluence, which guarantee this condition, are undecidable properties and we are able to obtain both of them through our design strategies.

5.1. Data representation

The preceding considerations strongly suggest that a Noetherian and confluent term rewriting system could be embedded in a programming language as follows. If a term $t$ is in normal form, then $t$ must be (associated to) a data object of the hosting environment, e.g. a literal or a value assignable to a variable. If a term $t$ is not in normal form, then the leading symbol of some subterm of $t$ must be (associated to) some computational device of the hosting environment, that we have previously referred as subprogram. In this subsection we propose a methodology which allows us to associate to any normal form of any abstract data type that we design with our strategies a data object of each of the three hosting environments we consider.

The normal forms of a specification, i.e. the terms we want to associate to data objects, are terms built with constructors only. This is easy to see. The binary choice strategy generates complete operations. This implies that any term whose leading symbol is an operation is reducible, hence it is not a normal form. The recursion reduction strategy generates terminating operations. This implies that we cannot keep reducing terms forever. Thus the terms that we cannot further rewrite must be built with constructors only. This resolves two key problems for the direct implementation of a specification: we have an operational definition of the concepts of values and computations and we understand the characteristics of the objects we need to represent in a hosting environment.

The problem of an adequate internal representation, in a hosting environment, for abstract values of a specification is interdependent with that of referencing these values through an appropriate notation available in the environment’s programming language. This referencing capability is a bridge from the hosting environment to the abstraction provided by the specification. The goal here is a notation compatible with the characteristics of the target language and as close as possible to the abstract notation. We call this notation \textit{concrete} as opposed to the abstract notation of abstract values as terms. In Prolog and C the syntax of the concrete notation is identical to the abstract one (see Figure 5, line 3). In LISP we adopt the LISP notation for terms. For example, the abstract \textit{bin.tree} value $\text{make}(\text{a, make}(b, \text{empty}, \text{empty}), \text{empty})$, where $a$ and $b$ denote values of sort \textit{node}, (see Figure 5) is denoted in LISP as \textbf{make} \textbf{a} (\textbf{make} \textbf{b} (\textbf{empty}) (\textbf{empty})) (\textbf{empty}). Notice the shifting of open parentheses to the left and the introduction of parentheses around the occurrences of the atom \textbf{empty}. These will be justified shortly.

In each hosting environment abstract values are internally represented using native types and data structures. Prolog offers the simplest solution. Since symbols and terms are primitive structures of the language, the Prolog interpreter generates the internal representation of an abstract value from its concrete notation without putting any burden on the direct implementation. In languages in which terms are interpreted as applications of functions to arguments, such as LISP and C, a function must be associated to each constructor. We call constructor \textit{functions} such functions. The overall computation performed by these constructor functions is the same for both LISP and C. Let $t = c(x_1,\ldots,x_n)$ be an abstract value, i.e. a term in which $c$ is a constructor and the $x_i$’s abstract values. The LISP concrete notation for $t$ is $(c \ x_1 \ldots \ x_n)$. The $c$ concrete and abstract notations are the same. In these languages the evaluation mechanism of nested terms is left-to-right, innermost-first, thus all the $x_i$’s are first evaluated. This is a recursive invocation of constructor functions which results in the concrete representation of the $n$ arguments of $c$ in $t$. Then, the function $c$ is applied to these objects. $c$ simply combines these objects and a token symbolizing the constructor $c$ all together. In LISP the structure representing this combination is obviously a list. The head of the list is an atom representing $c$, the leading symbol of $t$. The tail of the list consists of the representation of the arguments of $t$. For example, the functions associated to \textbf{empty} and \textbf{make} could be implemented as follows.

\begin{verbatim}
(defun empty () (list 'empty))
(defun make (node left right)
  (list 'make node left right))
\end{verbatim}

(5.1)

In our implementation, constructor functions are defined in a very uniform way through a simple macro whose argument is an atom, $c$, symbolizing a constructor, and whose expansion is the form:

\begin{verbatim}
(defun c (rest args) '(c ,args))
\end{verbatim}

(5.2)

Notice the difference between the quote symbol "'", in the definitions (5.1), and the backquote symbol "" used in (5.2). Through
this mechanism, the \textit{LISP} internal representation of an abstract value is obtained by evaluating its concrete notation. The only difference with Prolog is that in Prolog the constructor functions are not necessary. Finally, it is interesting to observe that abstract values in a specification are similar to literals in a programming language, i.e. they are objects that "stand by themselves". In \textit{LISP}, the concrete notation and the internal representation of an abstract value are both lists. Accordingly, the concrete notation of an abstract value evaluates to itself.

In the \textit{C} programming language, the internal representation of an abstract value is generated, similar to \textit{LISP}, by evaluating its concrete notation. In \textit{C}, and most other imperative languages, terms are conveniently represented by trees implemented as linked structures. Any such structure is based on the canonical binary tree representation \cite{KNUT73} of the ordered tree of which a term is an abstract notation \cite{GOGU78}. There are two kinds of nodes in these linked structures: data nodes and connection nodes. Connection nodes form chains which connect together all the arguments of a constructor symbol. Connection nodes facilitate multiple references to the same term. Without connection nodes the representation of some argument \( t \) would need a pointer to a brother argument, thus making the representation of \( t \) context dependent. Data nodes contain tree pieces of information: a token symbolizing a constructor, a reference counter for garbage collection, and a connection node. Figure 6 shows a pictorial representation of the internal \textit{C} representation of an abstract term. The function associated to a constructor \( c \), when invoked, simply grabs a new node, marks its with the token for \( c \), sets its reference counter, and links it through connection nodes to the data nodes representing its actual arguments.

![Figure 6. Diagram of the internal representation in \textit{C} of the abstract term \texttt{make(a, make(b, ...), empty)}, see Figure 4. \( a \) and \( b \) are generic objects of type \texttt{node}. Data and connection nodes are represented by large and small boxes respectively. A data node contains a token associated to a constructor, a reference counter, and a connection node. Connection nodes form chains which connect together all the arguments of a constructor symbol.](image)

Garbage collection requires user's intervention, if at all, only in the \textit{C} environment. Since no circular structures can originate from the specification's operations, a reference counter in each node is the basis for efficient garbage collection. User's code manipulating abstract values through functions non-directly implemented should deal with the problem. Since the structure of any node is independent of the abstract data type its represents, the management of a pool of nodes is accomplished, easily and efficiently, by a few routines supported by system calls.

5.2. Computations

As we did for values, our first concern in addressing the problem of directly implementing an abstract operation is finding, in the target language, a natural and convenient notation for the invocation of the operation's concrete counterpart. An abstract operation is implemented by a function in the \textit{LISP} and \textit{C} environments and by a predicate in the Prolog environment. The concrete notation for the invocation of an operation's direct implementation is thus suggested by these choices. In \textit{C} concrete and abstract notations coincide. In \textit{LISP} the concrete notation is the \textit{LISP} notation for terms previously discussed. In \textit{Prolog}, terms containing nested occurrences of operations become sequences of calls to predicates, e.g. see Figure 8, line 2. Since this approach appears to be neither convenient nor natural, we will improve the situation through an extension of the \textit{is} built-in predicate.

The direct implementation of operations in \textit{LISP} and \textit{C} is greatly simplified by the concrete notation we have established for values and computations. An operation is implemented by a function whose code performs three main steps. The first one consists in determining which axiom must be applied to an actual combination of arguments. This is achieved by analyzing the structure of the arguments. The hypothesis of completeness guarantees that one such axiom always exists for any combination of arguments. The hypothesis of parsimony\footnote{The weaker (in our framework) hypothesis of confluence implies that if there are more than one axiom applicable to a combination of arguments, then the choice does not affect the result of the computation.} guarantees that only one axiom is applicable to any given combination of arguments. Thus, the determination of the axiom to be applied to an actual combination of arguments is coded as one alternation statement, possibly with several levels of nesting, in which the order of the tests is irrelevant. The second step consists in extracting, from the actual arguments, those occurrences, if any, which are arguments of computations appearing in the right side of the axiom. The third step consists in coding the body of each branch of the function's alternation statement. For each operation's implementation there is one branch for each operation's defining axiom. The body of each branch is simply the concrete notation of the term occurring in the right side of the corresponding axiom.

Figure 7 shows the result of the application of these three steps to the operation \texttt{leaf_count} specified in Figure 4. Both the \textit{LISP} and \textit{C} direct implementations are shown. The first step must determine which of the two axioms of \texttt{leaf_count} must be applied to the actual argument. For this, it is sufficient to test whether or not the argument, \( x \), is the internal representation of the abstract \texttt{bin_tree} value \texttt{empty}. According to the representation methodology previously described, this is obtained in \textit{LISP}

170
by checking the first element of the list representing $x$ and in $C$
by checking the token in the root node of the linked structure
representing $x$. If $x$ is not the representation of $\text{empty}$, then,
un the operation is complete, it must be the internal representa-
tion of an abstract value whose leading symbol is $\text{make}$. In this
we must execute the second step to fetch $L$ and $\tilde{R}$ which
occur in the right side of the second axiom of $\text{leaf.count}$. The
representation of $x$'s occurrence $L$ is fetched by the $\text{LISP}$ form
(third $x$) and by the C expression $\text{csarg(x,2)}$. $\text{csarg}$ is a C
macro whose first argument is a term and second an occurrence
to be accessed. Fetching $x$'s occurrence $R$ is similar. The third
step is obvious.

\begin{verbatim}
(defun leaf-count (x)
  (cond ((eq (first x) 'empty) 1)
        (t (* (leaf-count (third x))
             (leaf-count (fourth x))))))

bin_tree leaf-count(bin_tree x)
  { if (x->token == empty) return(1); 
    else return(leaf-count(csarg(x,2)))
      *leaf-count(csarg(x,3)); 
  }

Figure 7. Direct implementation in LISP, lines 1-4, and in C,
lines 5-10, of the operation $\text{leaf.count}$ specified in Figure 4.
\end{verbatim}

The direct implementation in Prolog is radically different. There
is no correspondence with the first and second step des-
dcribed above for the translation in $\text{LISP}$ and $\text{C}$. If $f$ is an
operation with $n$ arguments, the direct implementation in Pro-
log of $f$ is a predicate $f$ with $n+1$ arguments. The additional
argument of $f$ is used for returning the result of $f$ applied to
the other arguments. For each of the operation's defining axioms one
Horn clause is generated, and no fetching of occurrences is neces-
sary since the pattern matching feature of Prolog is identical to
that of our small specification language. In order to describe the
details of the translation we introduce a few notational conven-
tions. $P$ is the set of Prolog well-formed terms and $P^*$ is the set
of non-null strings over $P$. The comma symbol is overloaded, it
denotes both separation of string elements and concatenation of
strings, i.e., if $x = x_1, \ldots, x_i$ and $y = y_1, \ldots, y_j$ are strings,
with $i, j \geq 0$, then $x, y = x_1, \ldots, x_i, y_1, \ldots, y_j$. If $f$ is a function whose
range is a set of non-null strings, then $f(x)$ is the last element of
$f(x)$, and $f(x)$ is $f(x)$ without its last element. Combining the
previous two notations, we have $f(x) = f(x), f(x)$.

The scheme for translating defined operations of the small
language into Prolog predicates is based on the function $r$ which
maps abstract terms into strings of Prolog terms. Symbols of the
small language are mapped into Prolog symbols with the same
spelling. This abuse of notation, which simplifies our formulas,
is resolved by the context in which symbols occur and, whenever
possible, by the font in which symbols are typed, i.e., italic for the
small language and typewriter for Prolog. $T$ is a “fresh” Prolog
variable, i.e. a variable which does not occur elsewhere.

\begin{align*}
  r(t) &= \begin{cases} 
    X & \text{if } t = X \text{ and } X \text{ is a variable;} \\
    \tau(t_1), \ldots, \tau(t_k), c(\tau(t_1), \ldots, \tau(t_k)) & \text{if } t = c(t_1, \ldots, t_k) \text{ and } c \text{ is a constructor;} \\
    \tau(t_1), \ldots, \tau(t_k), f(\tau(t_1), \ldots, \tau(t_k), T) & \text{if } t = f(t_1, \ldots, t_k) \text{ and } f \text{ is an operation.}
  \end{cases}
\end{align*}

The definition (5.1) is extended by (5.2) to include the defining
axioms of the small language in its domain and, consequently,
the set of Horn clauses in its range. $r$ is extended as follows.

\begin{align*}
  r(f(t_1, \ldots, t_k) \rightarrow t) &= \\
  \begin{cases} 
    t & \text{if } r(t) \text{ is null;} \\
    t(\tau(t_1), \ldots, \tau(t_k)) & \text{otherwise.}
  \end{cases}
\end{align*}

$r$ associates a Prolog predicate $f$ to each operation $f$ of the small
language. Translations from algebraic specifications to Prolog in-
duce those proposed by [HSIA84, PETZ85, EMDE871], but none
of these methods is based on the (5.1). The correctness of our
methods is proved in [ANTO89b]. More precisely, $r$ is the value
(normal form) of $f(t_1, \ldots, t_k)$ if and only if $f(t_1, \ldots, t_k)$ is sat-
fified.

\begin{verbatim}
leaf.count(empty,1).
leaf.count(make(.,L,R),Z) :- leaf.count(L,X), leaf.count(R,Y), Z = X*Y.

Figure 8. Direct implementation in Prolog of the operation $\text{leaf.count}$ specified in Figure 4.
\end{verbatim}

The invocation in Prolog of an operation's concrete counter-
part looks quite different from the abstract notation. A simple
stratagem allows us to use a concrete notation equal to the
abstract one. This extension is supported by a predicate, isab,
which is for abstract data types what the is predicate is for
numbers. isab, which is declared to be an infix operator with
the same characteristics of is, is abstractly defined, through $r$,
as follows.

\begin{align*}
  \text{isab}(r(X), X) & \leftarrow \text{call}(r(X)).
\end{align*}

As an example of use, consider the problem of counting the num-
ber $N$ of repetitions of elements in a list $L$. Suppose that, in ad-
dition to length, specified in Figures 2 and 3, and is, specified
in Figure 1, the operation uniq, for removing duplicates from a
sorted list, has been specified too. The computation of $N$ can be
denoted in Prolog as follows.

\begin{align*}
  N = \text{isab}(\text{length}(L) - \text{length}(
\text{uniq}(	ext{sis}(L))))
\end{align*}

6. Implementation

Four aspects of our discussion can be automated: 1) the
binary choice procedure, 2) the expansion of the occurrences of
the recursive reduction symbol, 3) the translation from the small
specification language to a target programming language, and 4)
run-time utilities specific of a hosting environment.
The binary choice procedure is useful during the design of operations — an activity supporting the direct implementation of specifications, but largely independent of it. The strategy is a part of an Emacs major mode for algebraic specifications. The mode also provides a form of syntax directed editing for statements of the small language.

The notion of recursive reduction simplifies the notation, hence the coding and understanding of operations. For this reason, the expansion of a "+" symbol is not desirable from a human standpoint; code generation from axioms is an exception. Thus, the expansion of "+" symbols is a component of the front-end of the translator for the direct implementation of specifications.

The translator from algebraic specifications to a target programming language is the core of our approach to the direct implementation. The translator has been implemented using the Unix utilities lex, a scanner generator, and yacc, a parser generator for attribute grammars. The translator front-end performs both syntactic and contextual analyses, e.g. it detects violations of arities, missing and multiple symbol declarations, etc. During this analysis, it generates a symbol table and an internal representation of the specification axioms. The translator back-end generates source code in a target programming language. The code is assembled from the internal representation of the axioms and the information stored in the symbol table. There is one such module for each hosting environment.

Finally, for each hosting environment, a few specific utilities are necessary. For example, in Prolog a run-time version of the function \( \tau \) is necessary to support the isConnected predicate. In C, there are functions for the allocation, initialization, collection, and re-allocation of the nodes used in the linked structures which internnally represent abstract values. In C, we have also implemented routines for the input and output of abstract values. These facilities are already provided by the LISP and Prolog run time environments. In LISP, there are macros such as the one outlined in (5.2) for the definition of constructor functions.

7. Concluding remarks

Several authors have investigated the direct implementation of algebraic specifications adopting the strategy described here, i.e. translating a specification into source code of a target programming language. In particular, the target language is Prolog in [HSIA84, PETZ85, EMDE87], Pascal in [ANTO87], C in [JALO87], and Modula 2 in [KLA89]. All these attempts suffer of two common weaknesses: first, they neglect the fact that designing a formal specification often requires more time, skill, and understanding than coding a program, second and most important, the validity of a specification, which is the basis for its direct implementation, is taken for granted, although the problem of its verification is generally unsolvable [KNUT70, HUET80b], very hard to tackle in practical situations [FUTA84, GUTrT85], and substantially more difficult than the direct implementation itself.

Our contribution is twofold. First, we attack the problem of the direct implementation of specifications contextually with the efficient design of defect-free specifications, thus attempting to correct both of the above weaknesses. This is a fundamental point, since without it specification-based software construction and rapid prototyping are likely to be processes slower and less reliable than the traditional methodologies. Second, we consider various models of computation as hosting environments for the specification implementation. This generality provides a deeper insight into both certain aspects of the implementation, such as the "internal representation" and the "concrete notation", and their relationships with both specification characteristics, such as completeness, parsimony, and termination, and specification key concepts, such as the different roles played by constructors and operations.

Future work is possible along three independent lines. First, the specification language can be enriched to include quotient sorts, partial functions, and infix and mixfix operators. When quotient sorts are involved, the parsimony of the operations is not sufficient to guarantee the confluence of the specification. Lack of confluence implies the existence of critical pairs which, however, in this situation, may occur only between one defining axiom and one quotient axiom. Critical pairs can be easily detected by a superposition algorithm. If there exists a critical pair of irreducible terms, the specification is invalid and must be redesigned or discarded. However, the footnote of section 5.2 justifies why, if confluence is provided, our translation scheme continues to hold unchanged. Although there are no challenging problems for the direct implementation, the extension to quotient sorts has some practical value.

The second line of work consists in adding new hosting environments to those considered in this note. Languages with good support for data abstraction, such as C++, and Ada, seem to be the most interesting ones. Similar to the previous problem, this extension is unchallenging as far as code generation is concerned. However, it is valuable for practical applications, and may shed further light on basic and general issues concerning module interfacing and reusability, since it attack the problem at a higher level.

The third and most interesting line of work consists in the use of current and possibly new design strategies for code generation and, more important, optimization. Undecidable properties, such as termination, become easy to prove when certain design decisions are made explicit, e.g. as done by the notation of recursive reduction. Traditionally, compilers and specification analyzers do not have access to this information and spend a considerable effort to infer it. The binary choice method and the notion of recursive reduction capture, in simple forms, basic design decisions. If these design decisions are made explicit through an appropriate notation, translators from specifications to executable code can do a better job, making this approach to software construction more competitive and appealing.

8. Acknowledgments

The authors are indebted with the Support group of the University of Maryland, and particularly with Marvin Zeikowitz, for discussions concerning the direct implementation of algebraic specifications. John Shaw designed and implemented the front-end of the translator and the back-end for the C language.