Algebraic Methods in Prolog Programming

Sergio Antoy

TR 89-5

Rev. 1 September 1989

Algebraic Methods in Prolog Programming

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: We discuss some difficulties of coding and invoking functions implemented by predicates and we propose a number of conceptual tools for overcoming these difficulties. We use an algebraic model which supports strategies for designing complete, parsimonious, and terminating functions. We describe a translation scheme for converting these functions into predicates and we prove a number of properties of this transformation. Our investigation provides an algebraic interpretation of the cut and raises two issues of lack of orthogonality in the Prolog programming languages. We outline practical tools, based on our ideas, which simplify substantially some significant steps of the design and use of certain Prolog predicates. We extend current work on the translation from algebraic specifications and term rewriting systems to logic programs along two lines: from functions we generate relations which are satisfiable exactly once for each plausible invocation and our translation scheme accommodates both exceptionally terminating computations and quotient sorts. Several examples are presented.

Keywords: Prolog predicate design, Functional predicate, Algebraic Specifications, Completeness, Parsimony, Termination, Linearity, Cut, Algebraic to logic translation, Prolog engineering environment, Binary choice procedure, Recursive reduction.

1. Introduction

An innovative and powerful feature of the Prolog programming language is the adoption, as main underlying computational model, of the notion of relation instead of the more traditional one

This material is based upon work partially 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.

of function. Relations are more general than functions and the language exploits this increased generality through the backtracking mechanism. The combination relations/backtracking makes the language well-suited to a vast and interesting class of problems and contributes to the simplicity and elegance of their corresponding solutions. The example, concerning "parenthood", suggested in [CLOC84, p. 16] is emblematic in this respect.

Functions, which are ubiquitous in programming, must then be coded in Prolog by means of predicates which we call *functional*†. The notion of function is simpler than that of relation. The relational approach sacrifices some of this simplicity, in particular, the clear separation of the functional approach between input arguments and output value and, consequently, the fact that in a (total) function for each combination of input arguments there exists *exactly one* output value. When a predicate models a function, this "existence-and-uniqueness" condition is achieved by balancing various opposing mechanisms, such as an exhaustive, but not redundant, enumeration of pattern cases for the arguments in the consequents of Horn clauses and the use of the *cut*. To make the discussion more concrete, consider, for example, the problem of counting how many lists, in a list of lists, have a given length. A solution is proposed in Figure 1 by the predicate `count`.

```
count([],_,0).                                          (1)
count([X|Y],L,N) :- count(Y,L,I), c1(X,L,J), N is I+J.  (2)
c1([],0,1) :- !.                                        (3)
c1([],_,0).                                             (4)
c1([_|_],0,0) :- !.                                     (5)
c1([_|Y],L,N) :- M is L-1, c1(Y,M,N).                   (6)
```

Figure 1. `count(`x`,`l`,`n`)` is satisfied if and only if x is a list containing exactly n lists of length l.

We address the problem of insuring that each *plausible*† invocation of `count`, and hence of `c1`, is satisfied *exactly once*. Plausible invocations have the form `count(`x`,`l`,`n`)` where x is a *fully instantiated* list of lists, l is a non-negative integer, and n is an uninstantiated variable (or an integer whose value is the number of lists in x of length l, since we are concerned with the invocations which should be satisfied exactly once). Several things can go wrong with the code of Figure 1. A predicate may fail for some plausible combinations of its arguments — this may occur if the enumeration of pattern cases for the *input* arguments in the consequents of the clauses is not complete or if the recursion is used incorrectly — and effective backtracking may take place — this may occur if the enumeration of pattern cases for the *input* arguments in the consequents of the

† A formal definition of this concept will be provided later.

clauses is not parsimonious or if the cut is used incorrectly.

A satisfactory degree of confidence that `count` and `c1` are free from these defects is by no mean trivial to obtain. Verification, within the Prolog boundaries, of the "one-time-satisfiability" of a logic program can be attempted with extreme difficulty. The code is utterly opaque in several respects: e.g. there is no explicit information that the arguments of `count` must be typed and that the first two arguments must be fully instantiatied. Thus, unsatisfactorily, the solution has so far relied on the skill and intuition of the programmer. In this note we propose a discipline for problems of this kind. This discipline is to be employed in the design phase of a predicate and is based on an algebraic computational model.

This paper is organized as follows. First, we present a small algebraic language which is necessary to formalize a number of concepts, such as *functional predicate* and *plausible argument*, and properties, such as *completeness*, *parsimony*, and *termination*, which could not be directly defined as easily for predicates or Horn clauses. Then, we introduce two strategies, based on the *binary choice* procedure and the notion of *recursive reduction*, and we prove that, when they are employed for designing an algorithm expressed in our small language, they guarantee the above properties. Finally, we propose a translation scheme for deriving Prolog code from statements in our small language, we prove its correctness, and we relate the properties achieved by our design strategies to desirable characteristics of the Prolog code. Our approach can be mechanized and we will outline a prototypical implementation of its key steps.

2. A small algebraic language

The underlying model of our language closely follows the initial semantics of algebraic specifications [GOGU78] with which we assume some familiarity. At the core of any algebraic specification there is a typical set of axioms. In addition to the axioms, we also choose to declare both the constructors of each sort and the arity of each element of the signature. The language has only two statements: one for defining *sorts* and the other for defining *operations*. Statements are terminated by a period. There are no predefined sorts, thus familiar types, such as "number" and "list", will be the basis of most examples. Sorts are defined through their *constructors*. A constructor is an element of the *signature*. 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. Thus, referring to Figure 2, *0* is a constant constructor of *nat* and *succ* is a constructor of *nat* which takes one argument of sort *nat* — see line 4.

```
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 2. Definition of the sort *nat*, modeling the natural numbers, and the addition.

Operation definitions consist in a declaration of the operation arity and a set of *defining axioms*. Natural and conventional notations for the elements of the signature are preserved by allowing the use of infix and "mixfix" [GUTT85] operators. The declaration of the domain of a constructor or operation supplies a template for these characteristics. We ignore problems of precedence and associativity since they are irrelevant to our discussion. The defining axioms are "directed" equations [HUET80a] or rewrite rules — the intuitive meaning being that an instance of the left side of an axiom is replaceable by its corresponding right side, but not vice versa. Following Prolog's conventions, variables in the axioms are denoted by capitalized identifiers or the underscore symbol. Thus, referring to Figure 2, "+" is a binary, infix operation on *nat* defined by two axioms.

The few remaining features available in the small language are presented in the next figure. Sorts modeling some common types, such as "set" or "integer", need axioms involving only variables and constructors. These are called *quotient axioms* and the corresponding sorts *quotients*. Generic sorts are defined through the use of some parameter sorts. For example, the sort *list* (Figure 3) is parameterized by the sort *element* — the sort of the elements of a list. Finally, our model admits partial functions. Computations which are undefined for some arguments, such as the *head* of a null list, are called *exceptional* and are denoted by the distinguished symbol "?" on the right side of an axiom.

Our language, unlike similar ones, e.g. Larch [GUTT85], Obj2 [FUTA85], or LIL [GOGU86], lacks type declarations for variables. For reasons which will become clear shortly, the left sides of 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, explicit type declarations for variables are potential sources of inconsistencies, and seem to imply broader scopes. Furthermore, a concept we will introduce in a following section enhances the use of anonymous variables for which a type declaration is neither appropriate nor convenient. The use of anonymous variables makes definitions

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

Figure 3. 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.

more understandable by hiding irrelevant information. The arguments or arguments' components which do or do not affect the result of a computation are respectively given or denied a name.

3. Completeness and Parsimony — Binary Choices

Our goal is devising discipline for designing functional predicates. This entails a "one-time-satisfiability" condition which is related to two properties of sets of tuples, which are discussed in this section. We denote with $\mathcal{C}$ the signature's constructors, with $\mathcal{X}$ a set of variables, with "=" the *syntactic* equality, and with $T(\bullet)$ the set of terms built over the (sub)signature denoted by the symbol $\bullet$. We say that y is an *instance* of x if there exists a substitution σ such that $y = \sigma(x)$.

Examples: Referring to Figure 2, *0* is an instance of both *0* and X with substitutions {} and $\{0/X\}$ respectively. $succ(succ(X))$ is an instance of $succ(Y)$ with substitution $\{succ(X)/Y\}$. *0* is not an instance of $succ(X)$.

Definitions. A set S of k-tuples of terms in $T(\mathcal{C} \cup \mathcal{X})$ is *complete* if every k-tuple t of ground terms is an instance of at least one element of S. S is *parsimonious* if t is an instance of at most one element of S.

The concepts of completeness and parsimony are naturally extended to the operations of our small language. We say that an operation is complete (resp. parsimonious) if the set of tuples of terms

consisting of the arguments in the left sides of the operation's defining axioms is complete (resp. parsimonious). In many cases, we need the slightly stronger properties.

Definitions. An operation f is *strongly complete* (*parsimoniuos*) if f is complete (parsimonious) and each operation occurring in the right sides of some axiom defining f is strongly complete (strongly parsimonious).

Examples: 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 an instance of neither one. All the operations defined in this note are strongly complete and strongly parsimonious, with the exception of those presented in Figure 8.

We now describe a procedure for generating complete and parsimonious sets of tuples. The procedure is non-deterministic, since some selections and choices are left indefinite. Also, the termination of the procedure for arbitrary choices is not guaranteed. In practical applications, these selections and choices correspond to design decisions. 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 $\mathcal{C}$ 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 practical applications of the procedure, a place always occurs as an argument in a term t. If f is the leading symbol of t, then the sort of an occurrence is established by arity of f.

Binary Choice Procedure.

Input: 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 $s_1 \times \ldots \times s_n \rightarrow s$, insert in R a tuple

obtained by replacing in t the place at u with the term $c(\square,\ldots,\square)$ in which there are exactly n places and the sort of the i-the place, for $1 \leq i \leq n$, is s_i.

6. [Iterate] Go to step 2).

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 4. 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.

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* [STAL87] editor, as a template-driven interactive process. A simplified session of the coding of the operation computing the length of a list is presented in Figure 4. 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 its declaration, see Figure 3. Our interest in the binary choice procedure is motivated by the following result. First notice that the existence of an upper bound on the number of "inductive" choices is a sufficient (and necessary) condition for the termination of the procedure.

Theorem 1. If the binary choice procedure terminates, then its output is a complete and parsimonious set.

Proof. Let R denote the output of the binary choice procedure.

– Completeness: Huet proposes in [HUET82] an inductive, *constructive* definition of completeness, for linear tuples. R satisfies Huet's definition, hence his Lemma 3 guarantees our completeness.

– Parsimony: Let t_1 and t_2 be tuples of R such that t is an instance of both, i.e. there exist two

substitutions σ_1 and σ_2 such that $\sigma_1(t_1) = t$ and $\sigma_2(t_2) = t$. If $t_1 \neq t_2$, then for some occurrence u, selected in step 3 of the binary choice procedure, the leading symbols of t_1 and t_2 at u are different constructors. This implies $\sigma_1(t_1) \neq \sigma_2(t_2)$, which is a contradiction. Thus, $t_1 = t_2$ and R is parsimonious.

4. Termination — Recursive Reductions

A main computational device of the Prolog model is recursion. Incorrect use of recursion leads to non-terminating computations, thus it is a potential cause of the non-satisfiability of a Prolog predicate for some arguments. We address recursion in Prolog through a simple yet powerful notion of recursion defined for the operations of our small language.

A sort s is *recursive* if at least one of its constructors has at least one argument of sort s. The sorts *nat* and *list* previously presented are recursive — *succ* and the mixfix symbol "[|]" denote their respective recursive constructors. Two conditions are necessary for applying recursion in an axiom defining an operation f: 1) the sort of at least one argument of 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 s has only one argument of sort s. Figure 6 shows informally how to remove this limitation.

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

Figure 5. Axioms for the operation *length* whose left sides were defined in Figure 4. 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, and could be omitted entirely.

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

(4.1)
$$\rho(t) = \begin{cases} \rho(t'), & \text{if } t = c(\ldots, t', \ldots), \text{ where} \\ & \quad c \text{ is a recursive constructor and} \\ & \quad t' \text{ is its recursive argument;} \\ t, & \text{otherwise.} \end{cases}$$

The definition (4.1) is extended by (4.2) to include in its domain the left sides of defining axioms, i.e. terms of the form $f(x_1, \ldots, x_k)$ where f is an operation and, for all i, x_i is in $T(\mathcal{C} \cup \mathcal{X})$.

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

We also extend our notation with this concept. The symbol "!" on the right side of an axiom denotes the recursive reduction of the left side, see Figure 5. The symbol "!" in the small algebraic language bears no relation with the Prolog cut. Our interest in the notion of recursive reduction is motivated by the following result.

Theorem 2. For any operation f, if the terms on the right sides of its defining axioms are built using terminating operations and/or recursive reductions, then f terminates for every argument.

Proof. The proof is by structural induction [BURS69] on the "inductive" arguments of f.

sort *node* **parameter;** (1)
sort *bin_tree* (2)
  constructors (3)
    empty; (4)
    make(*node*, *bin_tree*, *bin_tree*). (5)
operation $leaf\_count(bin\_tree) \rightarrow nat$ (6)
  axioms (7)
    $leaf\_count(empty) \rightarrow succ(0)$; (8)
    $leaf\_count(make(\_, L, R)) \rightarrow !L+!R.$ (9)

Figure 6. 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 $leaf\_count(R)$.

The notion of termination in the sense of Theorem 2 (see also [DERS85]) should not be confused with the more familiar notion of termination associated to Turing machines or while loops [KFOU82]. However, these concepts are equivalent for operations which are not *under-specified* [ANTO89].

5. Translation

In the previous sections, we have defined certain properties of computations and introduced two strategies which allows us to design operations of our language satisfying those properties. In this section, we take advantage of these results by translating these operations into Prolog predicates and proving how the above properties affect the predicate behavior.

We adopt the following notations: Σ is a signature partitioned into constructors and defined operations, i.e. $\Sigma = \mathcal{C} \uplus \mathcal{D}$. $\mathcal{P}$ is the set of Prolog well-formed terms and $\mathcal{P}^+$ is the set of non-null strings over $\mathcal{P}$. 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 consists of non-null strings, then $\dot{f}(x)$ denotes the last element of $f(x)$, and $\bar{f}(x)$ denotes $f(x)$ without its last element. Combining the previous two notations, we have $f(x) = \bar{f}(x), \dot{f}(x)$. `abort` is an implementation dependent Prolog predicate which aborts the computation or handles exceptions.

The scheme for translating defined operations of the small language into Prolog predicates is based on the function $\tau: T(\Sigma \cup \mathcal{X}) \rightarrow \mathcal{P}^+$ defined below. τ maps terms into strings of terms. Symbols of the small language are usually 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. Anonymous variables and the symbol "!" denoting recursive reductions are notational conveniences which do not affect the definition of τ. `T` is a "fresh" variable, i.e. a variable which does not occur elsewhere. Initially, to simplify the exposition, we assume that there are no quotients axioms. This limitation will be removed in a following section.

$$\tau(t) = \begin{cases} \texttt{abort}, \_ & \text{if } t = ?; \\ \texttt{X} & \text{if } t = X \text{ and } X \in \mathcal{X}; \\ \bar{\tau}(t_1), \ldots, \bar{\tau}(t_k),\ \texttt{c}(\dot{\tau}(t_1), \ldots, \dot{\tau}(t_k)) & \text{if } t = c(t_1, \ldots, t_k) \text{ and } c \in \mathcal{C}; \\ \bar{\tau}(t_1), \ldots, \bar{\tau}(t_k),\ \texttt{f}(\dot{\tau}(t_1), \ldots, \dot{\tau}(t_k), \texttt{T}),\ \texttt{T} & \text{if } t = f(t_1, \ldots, t_k) \text{ and } f \in \mathcal{D}. \end{cases} \tag{5.1}$$

τ's inner working is, loosely speaking, the following: τ applied to a term t performs a post-order traversal of t during which the following transformation takes place recursively: if a subterm t' of t is in $T(\mathcal{C} \cup \mathcal{X})$, then t' is copied unchanged, else t' is replaced by an appropriate term t'', and the invocations of appropriate predicates to instantiate t'' to the evaluation of t' are generated and prepended (recursively) to the term resulting from the traversal of t.

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. τ is extended as follows.

$$\tau(f(t_1, \ldots, t_k) \rightarrow t) = \begin{cases} \texttt{f}(t_1, \ldots, t_k, \dot{\tau}(t)). & \text{if } \bar{\tau}(t) \text{ is null;} \\ \texttt{f}(t_1, \ldots, t_k, \dot{\tau}(t)) :- \bar{\tau}(t). & \text{otherwise.} \end{cases} \tag{5.2}$$

τ associates a Prolog predicate `f` to each operation f of the small language. We call τ-association this correspondence. If f is a k-ary operation, then its τ-associated predicate `f` is $k+1$-ary. The extra argument of `f`, conventionally the last one, is for returning the result of applying f to the other arguments. The relationship between an operation f and its τ-associated predicate `f`, except for exceptional computations, is captured by the following theorem.

Lemma 1. If $\mathcal{L}$ is a logic program whose predicates are τ-associated to non-exceptional operations in $\mathcal{D}$, then, for all $t_1,\ldots,t_k$ in $T(\mathcal{C})$ and f in $\mathcal{D}$, the goal `f(`t_1`,...,`t_k`,T)` is satisfied for `T` $= t$, for some t in $T(\mathcal{C})$, if t is derivable from $f(t_1,\ldots,t_k)$.

Proof. The proof is by strong arithmetic induction on n, the length of the derivation $f(t_1,\ldots,t_k) \stackrel{n}{\rightarrow} t$.

– Base case: $f(t_1,\ldots,t_k) \rightarrow t$ if and only if there exists an axiom $f(x_1,\ldots,x_k) \rightarrow x$ of which $f(t_1,\ldots,t_k) \rightarrow t$ is an instance. Since t is in $T(\mathcal{C})$, by case analysis of (5.1), $\bar{\tau}(x)$ is null. Thus, by (5.2), $\tau(f(x_1,\ldots,x_k) \rightarrow x) =$ `f(`x_1`,...,`x_k`,`x`)` and consequently `f(`t_1`,...,`t_k`,T)` is satisfied for $T = \dot{\tau}(x) = t$.

– Inductive case: $f(t_1,\ldots,t_k) \stackrel{n+1}{\rightarrow} t$ if and only if there exists a t' such that $f(t_1,\ldots,t_k) \rightarrow t' \stackrel{n}{\rightarrow} t$. $f(t_1,\ldots,t_k) \rightarrow t'$ if and only if there exists an axiom $f(x_1,\ldots,x_k) \rightarrow x$ of which $f(t_1,\ldots,t_k) \rightarrow t'$ is an instance. Since t' is not in $T(\mathcal{C})$, by case analysis of (5.1), $\bar{\tau}(x)$ is not null. Hence, there exists in $\mathcal{L}$ a Horn clause of the form `f(`x_1`,...,`x_k`,`$\dot{\tau}(x)$`) :-` $\bar{\tau}(x)$. By the inductive hypothesis, the claim holds for all goals in $\bar{\tau}(x)$. The Prolog evaluation of $\bar{\tau}(x)$ when x is instantiated to t' has the effect of instantiating $\dot{\tau}(x)$ to t. Thus, by the second case of (5.2), the claim is proved.

operation *count*$(list, nat) \rightarrow nat$ (1)
axioms (2)
count$([],\_) \rightarrow 0$; (3)
count$([X|Y], L) \rightarrow$! $+$ *c1*(X, L). (4)

```
count([],_,0).                                              (5)
count([X|Y],L,T3) :- count(Y,L,T1), c1(X,L,T2), +(T1,T2,T3). (6)
```

Figure 7. The operation *count* and its τ-associated Prolog predicate `count` discussed in Figure 1. The simbol "!" in line 4 stands for *count(Y,L)* which is translated into the string of Prolog terms "`count(Y,L,T1), T1`". The variable `T1` is generated by translator. Other subterms of line 4 are processed similarly. Some cosmetic improvements of line 6 are discussed in the next section.

We have now the terminology for formally defining the concepts of functional predicate and plausible combination of arguments.

Definitions. We call *functional predicate* a predicate `f` τ-associated to a strongly complete, strongly parsimonious, and terminating operation f. If the arity of f is $s_1 \times \ldots \times s_k \rightarrow s$, then $t_1,\ldots,t_k$,`T` is a *plausible combination of arguments* of `f` if and only `T` is an uninstantiated variable and, for $1 \leq i \leq k$, t_i is (τ-associated to) a term in $T(\mathcal{C})$ of sort s_i.

Our next result is the basis for a limitation of the satisfiability of functional predicates.

Lemma 2. If $\mathcal{L}$ is a logic program whose predicates are τ-associated to terminating operations in

$\mathcal{D}$ and $\mathtt{f}(t_1,\ldots,t_k,\mathtt{T})$ is a goal satisfiable twice for the same plausible combination of arguments, then there exists a non-parsimonious operation in $\mathcal{D}$.

Proof. If the goal $\mathtt{f}(t_1,\ldots,t_k,\mathtt{T})$ is satisfiable twice and all operations in $\mathcal{D}$ are terminating, then we may assume, without loss of generality, that the goal itself matches the consequents of two distinct Horn clauses of f. Since each clause is the translation according to τ of a distinct axiom of f, f is not parsimonious.

We now state and prove our main result.

Theorem 3. If $\mathcal{L}$ is a logic program whose predicates are all functional, then any goal $\mathtt{f}(t_1,\ldots,t_k,\mathtt{T})$ in which the arguments are plausible is satisfied exactly once.

Proof. By definition, f is τ-associated to an operation f which is terminating and strongly complete. Under these hypotheses, f applied to any combination of ground arguments $t_1,\ldots,t_k$ in $T(\mathcal{C})$ reduces to some term t in $T(\mathcal{C})$. Thus, by Lemma 1, f is satisfiable for any combination of plausible arguments. By hypothesis, the operation f and any other operation called by f are parsimonious, thus by Lemma 2, f is satisfiable at most once for any combination of plausible arguments.

Examples. The operation *max* of Figure 8 is not parsimonious. In fact, the term $max(0,0)$ is reduced to 0 by two distinct axioms. As a consequence, if max is the predicate τ-associated to *max*, then the goal max(0,0,T) is satisfiable twice. The operation *min* of Figure 8 is not complete. In fact, the term $min(0,succ(0))$ is a normal form. As a consequence, if min is the predicate τ-associated to *min*, then the goal min(0,succ(0),T) fails.

operation $max(nat,nat) \rightarrow nat$ (1)
axioms (2)
$max(0,Y) \rightarrow Y;$ (3)
$max(X,0) \rightarrow X;$ (4)
$max(succ(X),succ(Y)) \rightarrow succ(!).$ (5)
operation $min(nat,nat) \rightarrow nat$ (6)
axioms (7)
$min(0,0) \rightarrow 0;$ (8)
$min(succ(X),0) \rightarrow 0;$ (9)
$min(succ(X),succ(Y)) \rightarrow succ(!).$ (10)

Figure 8. The operation *max*, which has not been designed with the binary choice procedure, computes the maximum of two natural numbers. *max* is not parsimonious. The operation *min*, which has not been designed with the binary choice procedure, computes the minimum of two natural numbers for some combinations of arguments. *min* is not complete.

Finally, we briefly address the case of exceptionally terminating computations. This is known

to be a very difficult problem which has been approached using partial algebras or conditional rewriting. Our discussion is rather informal and pragmatic. The attempt to satisfy a goal $\mathtt{f}(t_1,\ldots,t_k,\mathtt{T})$, in which $\mathtt{f}$ is τ-associated to an operation f, mimics a particular derivation of the term $f(t_1,\ldots,t_k)$, i.e. that in which the leftmost innermost redex occurrence of t is reduced first. This *left-to-right innermost-first* derivation loosely speaking corresponds to the parameter passing mechanism called "call-by-value" in many programming languages. This derivation terminates exceptionally if and only if it employes an exceptional axiom. In this case, the Prolog interpreter attempts to satisfy the predicate `abort` which does not return or handles an exception. Thus, the predicate `f` behaves as the operation f in this case too.

6. Observations

The language which we are using to present our ideas is totally orthogonal in its treatment of types and its notation. To the contrary, Prolog presents some irregularities in the way it handles numbers and some operations on them. This is motivated by the desire of using conventional and natural notations, by the fact that numerical computations are functional *par excellence*, and by the choice of sacrificing generality for simplicity and efficiency. A comparison between the two approaches, particularly from the point of view of translating the first into the second, provides a few interesting results presented in the next sections. The analysis of some differences between the two models sheds a better light on two situations in which the cut is routinely used, supports an interesting and convenient alternative way of invoking functional predicates, which extends a Prolog feature, and suggests a few corrections to our translation scheme for a totally smooth integration of functional predicates and code traditionally produced.

We begin by observing that the Prolog interpreter is unable to pattern-match numbers against some representations expressed through the constructors. For example, the Prolog term $\mathtt{succ}(x)$ does not match any number for any value of x. Thus, while the algebraic value 0 can be denoted in Prolog by the term `0`, the algebraic value $succ(X)$ must be denoted by a Prolog variable, say `T`. A consequence is the loss of parsimony, since `T` matches `0` as well. Traditional Prolog code overcomes this problem by ordering clauses and employing the cut, for example see the predicate `sum_to` [CLOC84, p. 81]. We discuss a correction of τ for the the translation of some operations with arguments of sort *nat* in the next section.

The second observation concerns linearity. The consequents of functional predicates are always linear terms. However, Prolog predicates modeling some functions, for example those manipulating containers [GUTT85], such as the predicate `member` [CLOC84, p. 55], are occasionally coded using

non-linear consequents. In Section 8 we discuss the relationship between completeness, parsimony, and linearity and relate it to the cut.

The third observation concerns the fact that simple numeric computations are performed in Prolog "by interpretation", rather than by predicates as a stricter logic approach would impose. Thus, for example, the addition of numbers is not implemented by a ternary predicate, say "`add`", such that `add(`x`,`y`,`z`)` is satisfied if z is $x + y$, but is implemented by the predicate "`is`" and the *function* `+` through the construct "z `is` x`+`y". This choice appears to be simpler and more practical and natural than its pure logic alternative. It is quite simple to change τ to accomodate this characteristic. More interesting is a generalization of this idea, that we discuss in Section 9. This generalization entails an extension of the "`is`" predicate so that its second argument can include values of any type rather than just numbers.

7. Numeric functional predicates

Since Prolog does not recognize *succ* as a constructor of the natural numbers, the translation of an occurrence of `succ` is an exception to the third case of (5.1). Different translations are necessary depending on whether *succ* occurs in the left or right side of an axioms, i.e. it goes in consequent or antecedent of a clause. For an occurrence in the right side, a suitable translation looks like that of an operation. Obviously, no axioms are or could be provided for *succ* and the evaluation is performed by Prolog primitives.

(7.1) $$\tau(succ(x)) = \bar{\tau}(x),\ \texttt{T is } \dot{\tau}(x)\texttt{+1, T}$$

Some obvious optimizations are possible for nested applications of *succ* and/or when x is a ground term, e.g. $succ(succ(0))$ is simply translated into 2.

An occurrence of *succ* in the left side of an axiom is translated into a fresh variable `T` and, in addition, if the argument of the occurrence is referenced in the right side, then its value must be derived from that of `T`. This translation, denoted by η, is used similarly to τ.

(7.2) $$\eta(succ(x)) = \bar{\eta}(x),\ \dot{\eta}(x)\ \texttt{is T-1, T}$$

$\bar{\eta}(succ(x))$ is prepended to the antecedent. $\dot{\eta}(succ(x))$, i.e. `T`, replaces $succ(x)$.

This translation creates an obvious problem. Let us call `f` a predicate resulting from this translation, f the operation from which `f` is derived, and $\texttt{f}_i$ one clause of `f` in which `T` occurs in the consequent. This occurrence of `T` matches any instance of a number, *including* `0`. However, if f is

complete, then any instance of arguments matching $\mathtt{f}_i$ and instantiating T to 0 would also match the arguments of some other clause $\mathtt{f}_j$ of f for $j \neq i$. The problem with this translation is that (7.2) fails to make f parsimonius. The problem is resolved by using the cut in each clause resulting from an axiom containing occurrences of *0* in its input arguments and insuring that the clauses are asserted in the right order, i.e. first those containing cuts. An example of an application of this strategy is in Figure 9 which shows the operation *c*1 and its τ-associated Prolog predicate c1 which performs part of the computation discussed in Figure 1. Any optimization has been suppressed to make the steps of the translation more easily understandable.

operation $c1(list, nat) \rightarrow nat$ (1)
axioms (2)
$c1([], 0) \rightarrow succ(0)$; (3)
$c1([], succ(\_) \rightarrow 0$; (4)
$c1([\_|\_], 0) \rightarrow 0$; (5)
$c1([\_|\_], succ(\_)) \rightarrow !$. (6)

```
c1([],0,T1) :- !, T1 is 0+1.                          (7)
c1([],_,0).                                           (8)
c1([_|_],0,0) :- !.                                   (9)
c1([_|T1],T2,T3) :- T4 is T2-1, c1(T1,T4,T3).         (10)
```

Figure 9. The operation *c*1 and its τ-associated Prolog predicate c1 discussed in Figure 1. The clause in line 7 has not been optimized to facilitate the understanding of the discussion in the text. It is interesting to notice the absence of named variables in *c*1's axioms. The names of the variables of c1 are generated by the translator.

8. The Cut

The algebraic formalism characterizes precisely two situations in which the cut is a device to avoid coding predicates which are satisfiable more than once by some combinations of arguments. One situation, discussed in the previous section, stems from the fact that Prolog is capable of matching a number against the term 0, but not against a term of the form, e.g., succ(x). The necessity of using a variable in place of the second term fails to provide parsimony. The cut must then be used in Prolog programs for situations of this kind to correct the problem.

Another situation in which the cut must be used with the same purpose is related to the notion of linearity [HUET80b, HUET82] which we recall below.

Definition. A k-tuple t of terms in $T(\Sigma \cup \mathcal{X})$ is *linear* if any variable in t occurs in only one occurrence.

Linearity, similar to completeness and parsimony, is naturally extended to operations and predicates

by considering the tuples consisting of the arguments in the left sides of defining axioms, or of the input arguments in the consequents of clauses. Non-linear variables are those occurring in more than one occurrence in a tuple. Non-linear tuples are those with occurrences of non-linear variables. The following theorem relates linearity to completeness and parsimony.

Theorem 4. 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 $\mathcal{C} = \{nat\}$ (see Figure 2) and address the problem of its generalization later. For each 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 — the set of pairs $\{\langle X, X\rangle, \langle black, white\rangle, \langle black, white\rangle\}$ is complete and parsimonious over the signature of nullary constructors $\{black, white\}$.

The above theorem leads to two interesting corollaries which we discuss informally. If `f` is a Prolog predicate and $\mathtt{f}_i$ is a clause of `f` non-linear over an infinite type, such as number, list, and tree, then $\mathtt{f}_i$ must be followed by at least one similar linear clause $\mathtt{f}_j$ if `f` is a complete predicate. Furthermore, if `f` is a parsimonious predicate, then either a cut must be used in $\mathtt{f}_i$ or some form of inequality must be included in $\mathtt{f}_j$.

9. Interpretation

With few exceptions for programs performing numeric computations or symbolic manipulation, nested terms in Prolog programs are limited to those in $T(\mathcal{C})$. Functional predicates may be invoked, following this approach, as any other Prolog predicate. However, τ provides an alternative form of invocation of functional predicates which makes their use as convenient as the use of arithmetic operators. Numerical computations could be performed in Prolog by a set of "arithmetic predicates"

to be used as shown by the ternary "+" in Figure 7, line 6. This approach, although feasible and Prolog-like, is inconvenient. Prolog provides a built-in infix predicate, `is`, for the interpretation and evaluation of arithmetic expressions. This approach is more natural when writing code for numerical computations. The algebraic model, through the translation τ, extends this feature to user-defined abstract data types. This extention is based on a predicate, `isab`, which is for abstract data types what the `is` predicate is for numbers. `isab` is declared to be an infix operator with the same characteristics of `is` and is abstractly defined as follows.

(9.1) `isab(`$\dot{\tau}$`(X),X) :- call(`$\bar{\tau}$`(X)).`

As an example of use, let us assume that simple primitives for list manipulation, such as *length* (see Figures 4 and 5), *sort* for sorting a list, and *uniq* for removing duplicates from a sorted list, have been defined. The number `N` of repetitions of elements in a list `L` may then be coded in Prolog as follows.

(9.2)
```
N isab length(L)-length(uniq(sort(L)))
```

Statement (9.2) is more practical to code, more reliable to use, and much simpler to understand than its following traditional Prolog equivalent.

(9.3)
```
length(L,T1), sort(L,T2), uniq(T2,T3), length(T3,T4), N is T1-T4
```

A drawback of using `isab` is the cost of the run-time translation (interpretation) required for its second argument. This is overcome by "compiling" invocations of `isab` into their corresponding expansions. The natural place and time for this compilation occur during the design of a program. The solution, again, is obtained through the *GNU Emacs* editor. With a keystroke, the invocation of `isab` preceding the cursor in the text being edited is replaced by an equivalent sequence of predicate invocations. The original term is retained in the form of a comment for documentation purposes.

10. Quotients

In this section we outline, how to extend τ to the situations in which there are quotient axioms. We make the assumption that the term rewriting system established by the axioms is *canonical* [HUET80b], i.e. *Noetherian* and *confluent* [KNUT70]. We believe that in most cases, without this assumption an algebraic specification is not an appropriate model to define types and computations. When there are no quotient axioms the Noetherianity of the system is guaranteed

by the recursive reduction strategy. The confluence is a trivial consequence of the binary choice strategy. In fact, if all the operations are parsimonious, then there are no critical pairs, and hence the system is confluent. Both our startegies cannot be used for quotient axioms, however when quotient axioms are added to a canonical term rewriting system, the canonicity of the new system may be verified with the techniques described in [DERS85] as far as the termination is concerned, and with a superposition algorithm [HUET80b] as far as the confluence is concerned.

The existence of quotient axioms affects our discussion in a limited way. All our results remain valid with the exception of Lemma 1. Lemma 1 too continues to holds if we require that t is derivable from $f(t_1,\ldots,t_k)$ without employing any quotient axiom. The implication is that the "one-time-satisfiability" of the functional predicates for plausible arguments is preserved, but the result of a computation may not be in normal form. For example, referring to the definition of the sort *integer* of Figure 3, *0* and *succ(pred(0))* are two different representations of the same object. Only the former is a normal form.

A simple solution of this representational problem consists in normalizing the result of a computation when appropriate. Normalization is performed by the predicate `normalize`. `normalize(`x`,`y`)` is satisfied if and only if y is the normal form of x. The predicate `normalize` is a driver for the predicate `quotient` which is defined as follows. Each quotient axiom $l \rightarrow r$ is translated into the clause

(10.1) `quotient(`l`,T) :- quotient(`r`,T).`

`normalize` reduces its first argument using quotient axioms until no further reduction is possible. A convenient place to normalize results of computations is when `isab` is called. Obviously, `normalize` must be coded so that it is satisfied exactly once for each invocation.

The traditional unification mechanisms of algebraic languages and Prolog differ as far as the *occur check* feature [STER86] is concerned. This difference is irrelevant for defining axioms since the binary choice method generates sets of linear tuples. However, quotient axioms may be non-linear, thus the same unification mechanism must be adopted by both models to guarantee semantic equivalence. Incidentally, we notice that the existence of a normal form occasionally may be achieved through special forms of unification. This problem is marginal to our discussion, but the solution we propose is compatible with this approach.

As with natural numbers, the integration of abstract integers with Prolog integers needs some adjustments. The following two definitions extend τ and η for non-variable occurrences of integers.

Both work as well for naturals and η offers an alternative to the use of the cut previously proposed for instances of *0*. This alternative is slightly less efficient, but it overrides the problem of ordering the clauses resulting from the translation of defining axioms.

(10.2)
$$\tau(t) = \begin{cases} \texttt{0} & \text{if } t = 0; \\ \bar{\tau}(x),\ \texttt{T is } \dot{\tau}(x)\texttt{+1, T} & \text{if } t = succ(x); \\ \bar{\tau}(x),\ \texttt{T is } \dot{\tau}(x)\texttt{-1, T} & \text{if } t = pred(x). \end{cases}$$

(10.3)
$$\eta(t) = \begin{cases} \texttt{0} & \text{if } t = 0; \\ \texttt{T>0, } \bar{\eta}(x),\ \dot{\eta}(x) \texttt{ is T-1, T} & \text{if } t = succ(x); \\ \texttt{T<0, } \bar{\eta}(x),\ \dot{\eta}(x) \texttt{ is T+1, T} & \text{if } t = pred(x). \end{cases}$$

11. Implementation

Four processes of our discussion can be automated: 1) the binary choice procedure, 2) the expansion of the occurrences of the recursive reduction symbol, 3) the functions τ and η, and 4) the compilation of `isab` occurrences.

The binary choice procedure is useful during the design of operations — an activity largely independent of Prolog programming. 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 backtracking capabilities of Emacs make extremely simple the recovery from wrong selections and choices. We are currently experimenting with the option of limiting the capabilities of Emacs so that operations are guaranteed to be complete and parsimonious, although this choice seems too restrictive at times.

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 a "!" symbol is a modular component of the first version of τ which we will describe shortly.

The functions τ and η are the core of our translation scheme. τ is useful in two versions. One, jointly with η, for both the translation of algebraic axioms into Prolog predicates and the compilation of `isab`. These activities too are naturally executed during the design-editing phase. The second version of τ is useful for the execution of `isab` occurrences. In this case, τ is part of the Prolog run-time environment and is implemented by a predicate. This version is unconcerned with "!" symbols, since they have already been expanded during the translation of operations into predicates. In addition to this predicate, a bit of extra information is necessary for discriminating symbols between constructor and operations. This information takes the form of facts which are

derived from the statements of the small language and are automatically generated at the time the axioms are translated.

Our system is not yet fully integrated and its user interface may be improved. All processes running within *Emacs* are activated by a single keystroke, a fact that makes their use very practical. We are currently experimenting with two different forms of implementation: by means of *Emacs-LISP* functions and by means of *inferior processes* as described in [HALM88].

12. Related work

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 required for a principle of definition assumed by an "inductionless" method of proof by induction. [THIE84] and [KOUN85] extend this definition to the non-linear case and propose procedures for completing incomplete sets. However, both papers miss the relationship between completeness, parsimony, and linearity, and hence the limitation of their extensions. In these papers, non-parsimonious sets, called respectively ambiguous and redundant, are addressed by a process of tree growth. [ANTO89] proposes the two strategies recalled at the beginning of this paper and combines them for designing abstractions which are neither under- nor over-specified. These concepts are at the basis of predicates which are satisfiable exactly once for plausible invocations.

The relationships between algebraic and logic models of computation have been widely investigated. Early work concerning various efforts for translating algebraic axioms into Prolog predicates is summarized in [PETZ85]. These efforts have been directed toward prototyping algebraic specifications through their direct implementation and/or through the use of Prolog supported automatic theorem provers. Our viewpoint is opposite — the algebraic model is used to simplify, speed up, and make more reliable the coding and invocation of Prolog predicates.

[HSIA84] discusses at length the problems of translating algebraic axioms into predicates and proposes two methods. Our approach offers several advantages with respect to these methods: no initial user defined implementation is required since the meaning of constructor is fully exploited; partial functions and quotients are supported; full integration with the Prolog environment is available; if the operations are designed using the binary choice and the recursive reduction strategies, then the problems of endless loops of Method 1 are overcome and Method 2 becomes unnecessary; and last but not least, τ is formally defined, thus the effects of this translation can be simply stated and rigorously proved.

More recently, efforts have been directed to the integration of functional and/or equational programs into logic programs. [EMDE87] presents two approaches to this problem. The translation scheme of the approach called *compilational* is very similar to τ, although we believe that τ is more elegant and terse. The main result of the compilational approach is Theorem 7.4. Our results are complementary to it in two respects: 1) the theorem is proved under the hypotheses of Noetherianity and *strong-E-canonicity* [EMDE87, Definition 7.2]; we provide relatively weak sufficient conditions for these assumptions, and 2) the theorem addresses satisfiability, but not re-satisfiability as we do in our Theorem 3.

[TOGA87] extends the "basic" set of axioms to be translated into a logic program with equations, such as the distributive law of multiplication [TOGA87, Equations (3) and (4)], which are not a part of the definitions, but can be easily proved as theorems [BOYE79]. This work defines, *non-constructively*, the notion of *successfully terminating* computation and requires this hypothesis for the proof of the main result, Theorem 4. Operations designed with the strategies recalled at the beginning of this note lead to successfully terminating computations (in this sense) for all inputs.

13. Concluding remarks

Prolog's underlying relational model and pattern matching and backtracking features greatly contribute to the suitability of the language to a vast and interesting class of problems. However, the possibilities offered by this combination are not easy to master and the resulting power is sometimes difficult to control. This occurs more frequently for those computations which are traditionally thought of and naturally expressed as functions. The relational model makes more difficult the design of these "functional algorithms" and more cumbersome their use within Prolog programs.

Our contribution consists in a methodology which speeds up the design of "functional predicates", guarantees their behavior, and simplifies their use. Our results are achieved through two key steps. First, understanding some relevant properties of sets of tuples and discovering strategies to generate sets satisfying such properties. Second, devising a scheme for migrating algorithms from the environment most convenient for their design to the Prolog environment with no loss of efficiency and in a form natural to use. Interestingly enough, both steps are supported by formal proofs and lead to very practical implementations.

Our investigation addresses two areas of application. One concerns the engineering of Prolog programs, also in conjunction with environments directed toward increasing code productivity and reliability. The other concerns the design of the Prolog language, particularly the consequences of

missing or limited features and the requisites for their addition or extension. This research may be of value for the designers of future programming languages who aim at merging in a single environment a functional (equational) and a relational (logic) model of computation.

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.

[BOYE79] Boyer R.S. and J.S. Moore, A Computational Logic, Academic Press, New York, NY 1979.

[CLOC84] Clocksin W. F. and C. S. Mellish, Programming in Prolog, second ed., Springer-Verlag, New York, 1984.

[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.

[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.

[GUTT85] Guttag J.V., J.J. Horning, and J.M. Wing, "The Larch Family of Specification Languages", *IEEE Software*, Sept 1985, 24-36.

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

[HSIA85] Hsiang J. and M.K. Srivas, "A Prolog Environment for Developing and Reasoning about Data Types", LNCS **186**, 227-405, Springer-Verlag, NY, 1985.

[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.

[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", LNCS **204**, 1985, 348-362.

[PETZ85] Petzsch H., "Automatic Prototyping of Algebraic Specifications using Prolog", Recent Trends in Data Type Specification, selected papers of the 3rd Workshop on Theory and Applications of Abstract Data Types, Springer-Verlag, 1985, 207-223.

[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.

[STER86] Sterling L. and E. Shapiro, The Art of Prolog, The MIT Press, Cambridge, MA, 1986.

[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.