A PROLOG Environment for Developing and Reasoning about Data Types[1]

Jieh Hsiang **Mandayam K. Srivas**
Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794
U.S.A.

Abstract

PROLOG is a programming language based on first order logic. The feature that distinguishes PROLOG from most other programming languages is that the execution of PROLOG programs is based on subgoal reduction and unification. Unfortunately, the reliance on unification for execution has also inhibited PROLOG from utilizing some recently developed concepts in programming languages such as abstract data types. In this paper we introduce a discipline for incorporating abstract data types into PROLOG, and study the use of PROLOG as a uniform programming environment for the specification, implementation, and verification of PROLOG programs. We illustrate the application of the environment to the development of abstract data types in PROLOG.

In addition to producing executable specifications, the proposed discipline also provides automatic means of refining a specification into an implementation. We also present a PROLOG-based inductive theorem proving method for proving properties of data types and correctness of implementations.

1. Introduction

In this paper, we study the use of PROLOG as a uniform environment for the systematic development of data types in PROLOG. That is, its use for the specification, implementation and verification of data types represented and used as PROLOG programs.

The idea that PROLOG can naturally be used as a specification cum implementation language is well-known. It has been used in [Dav82] and [HaT82] for systematically developing PROLOG programs. However, none of these works address the formal development of abstract data types, nor do they provide proof facilities for verification. In order to incorporate all three aspects of data type development, our environment consists of a PROLOG interpreter augmented with three primitives - *Refine*, *Ver_Cond_Gen*, and *Ind_Prove*. *Refine* is a program transformation processor; *Ver_Cond_Gen* generates verification conditions that have to be proved to establish the correctness of a data type implementation; *Ind_Prove* is the theorem proving primitive that proves the correctness of the verification conditions and properties of PROLOG programs in general. We have implemented a system (in a version of CPROLOG for the VAX11-780 machine) that supports our methodology. The environment assists in performing the following activities in the development of data types.

[1]Research supported in part by the National Science Foundation under grants DCR-8401624 and DCR-8319966

(1) *The design and specification of abstract data types*: Our discipline requires the separation of the operations of data types ([GuH78], [GTW78]) into two categories: *constructors* - operations which uniquely define the terms in the domain of the data types, and *defined operations* - operations that are defined in terms of the constructors. Only the constructors are represented as functors while the defined operations are represented as predicates. Therefore a *specification* of a data type consists of PROLOG programs that define the defined operations as relations on the constructor terms. Such an approach not only eliminates the need for any special unification algorithm (as required by other approaches such as [Kor83] and [SuY84]), but also produces executable specifications.

(2) *Implementation of data types*: Since the specifications written using the aforementioned discipline are executable PROLOG programs, there is no need for a different implementation. However, sometimes for reasons of efficiency and convenience of representation, one would want to implement the data types (and their operations) differently. We propose methods (that are implemented by *Refine*) for mechanically deriving an implementation for a defined operation of a data type from its specification once an implementation for the constructors are given. The proposed discipline for building specification enables the transformation methods to be easily automated. The methods can also be used to transform a program that uses the specification of a data type into one that uses an implementation of the data type.

(3) *Proving Properties of Programs*: Data type verification includes mainly two aspects: (i) proving that the data types defined (as specifications) have the desired properties (such as associativity for *append*), and (ii) an implementation of the data type is correct with respect to the specification. If the implementation is derived automatically as described above, the correctness of the implementation is guaranteed as long as the implementation of the constructors is correct. If the user wishes to derive the implementation himself (manually), then to prove the correctness, the user has to specify the *representation invariant* and the *representation equivalence* that characterize the representation scheme used by the implementation. *Ver_Cond_Gen* , the verification condition generator, will then generate the verification conditions, which will, in turn, be proved with the help of *Ind_Prove* . The task of generating the verification conditions is considerably easier in PROLOG than in other program languages with verification capability since the verification conditions are representable in the same language.

In order to carry out the entire verification step in PROLOG, we introduce an new inductive theorem proving method. This method has the ability to prove universally quantified properties expressed in Horn clauses, in particular those which require structural induction. It is also tailored to utilize the existing PROLOG facilities such as backtracking, and can be easily built into the PROLOG machinery. While a detailed design of this PROLOG based inductive theorem prover is given in [HsS84a], in this paper we give an overview of the method, and use it to carry out the verification of programs in our environment.

AFFIRM [Mus80] and OBJ [GoT79] are two examples of (non-PROLOG based) systems built for the verification of abstract data types. Neither, however, has been able to unify the different phases of program development into one language framework. In AFFIRM the programs are written in Pascal while the statement of the verification conditions and their proofs are carried out in a framework based on rewrite rule theory [HuO80]. OBJ is a system for writing and testing algebraic program specifications. Although it can in principle be used as a programming language (since it allows algebraic specifications to be executed), it lacks the expressive power of a general purpose language. The important advantages of our system over the above term rewriting based systems are that (1) ours does not need a Knuth-Bendix type completion procedure, that is potentially non-terminating, for proving inductive theorems, and (2) ours can handle conditionally defined axioms more effectively.

The next section describes how abstract data types can be specified in PROLOG. Section 3 is concerned with implementation of abstract data types. It discusses several transformation techniques for transforming a specification of a data type into an implementation. Section 4 deals with the formulation and generation of the verification conditions for proving correctness of abstract data type implementations. Section 5 gives an overview of the inductive PROLOG theorem prover *Ind_Prove*. The last section has the concluding remarks.

2. Abstract Data Types and Their Specifications

2.1. Designing Data Types in PROLOG

A major issue that has to be addressed in building abstract data types into PROLOG is to determine how the operations of the data types should be specified. Two main approaches have been used for incorporating data types into PROLOG. The *predicate approach*, which is practiced commonly, represents all the operations of a data type as predicates on *List* and *Nat* (natural number) which are readily available in PROLOG. Such an approach treats all operations in a data type on a flat structure. The *functional approach* (egs., [Kor83], [SuY84]) encourages representing operations as functors. The axioms that the terms constructed using the operations satisfy are treated in the unification process by some additional mechanisms. The functional methods provide the convenience of having nested terms, but their additional unification mechanisms may become a source of inefficiency or, worse yet, they may change the semantics of the programs.

We use a hybrid approach. We partition the operation set of a data type into two groups: the *constructors* and the *defined operations* [HuH80]. The constructors are a set of operations that can generate every value of the data type *uniquely*. (Such a restriction excludes data types such as *Set* from our domain of application, since the constructors of *Set* do not construct values uniquely. However, if the constructors of the data type exhibit properties such as associativity and commutativity for which special purpose complete unification algorithms exist ([Sti81], [Fag84]), our method still applies provided such a special purpose unification algorithm is implemented in the PROLOG system.) The defined operations perform other interesting computations, and are defined in terms of the constructors. In our approach *only* the constructors are represented as functors, and the defined operations are represented as predicates. An n-ary defined operation $p(x_1, \cdots, x_n)$ is represented as the $(n+1)$-place predicate $p(x_1, \cdots, x_n, x_{n+1})$. Thus, in our system (functional) terms appearing in a PROLOG program contain only constructors of data types.

Consider the data type *List* that is commonly used in PROLOG. The constructors for *List* are the empty list [ ], and the *cons* operator denoted as $[A \mid L]$. All other operations on *List* can be specified in terms of the above constructors, and are considered as defined operations. As a second example, consider the data type *Tree* intended to model binary trees that can store arbitrary information in their nodes. *Tree* can be designed in PROLOG by choosing two new constructors *emptytree* and *mktree*: *emptytree* constructs an empty binary tree; *mktree* (L, N, R) is a ternary constructor that builds a binary tree with L and R as its left and right subtrees, and N as the label at its root. Predicates such as *ltree* and *rtree* that extract the left and the right subtree, and *isin* that checks the membership of a node in a tree, are some of the defined operations of *Tree*.

The most significant difference between our approach and functional approaches is that our method does not need a unification process different from the one in PROLOG. Most functional approaches require more elaborated unification procedures in order to unify terms which satisfy additional (functional) axioms. In paramodulation-type approaches, such as [Kor83], such a unification algorithm affects the efficiency considerably; in others, especially those that use term rewriting-based unification algorithms (such as [SuY84]), these unification algorithms may result in a change in semantics due to incomplete or infinite unification processes. (See [Hul80] for a discussion of unification algorithms for equational theories based on term rewriting.) Other significant advantages, all of which arise because of the separation of constructors from defined operations, of our approach over the other two approaches are that ours provides (1) the ability to successfully execute specifications of data types, (2) the ability to automatically transform specifications into implementations, and (3) the ability to use structural induction in a natural way while proving properties.

2.2. Specification of Data Types

The *Specification* of a data type consists of defining (as PROLOG programs) each of the defined operations as a relation on the constructor terms of the type. The specification of a data type is always executable because, firstly, it is a PROLOG program, and secondly, the constructor terms uniquely construct the values of the data type. For example, the operation *append* on *List* can be specified as follows.

Specification of append

append: List $\times$ *List* $\times$ *List*

append([ ], L, L).
append([X | $L1$], $L2$, [X | $L3$]) :- *append*($L1$, $L2$, $L3$).

As a second example, consider the data type *Tree* with a defined operation *isin* which checks the membership of a node inside a given tree. We assume that there exist data types *Node* and *Bool* (constructors *true* and *false*) specified elsewhere; *diff* is a predicate on nodes that checks syntactic inequality on nodes (egs., *diff*(X,X) fails, and *diff*(X,Y) succeeds where X and Y are unbound variables), and *or* is a *Bool* predicate. Then *isin* can be specified as a predicate relating the terms constructed out of the constructors of *Tree* as follows.

Specification of Tree

Functors $\quad$ *emptytree* : $\rightarrow$ *Tree*
$\qquad\qquad\quad$ *mktree* : *Tree* $\times$ *Node* $\times$ *Tree* $\rightarrow$ *Tree*

Predicate $\quad$ *isin* : *Tree* $\times$ *Node* $\times$ *Boolean*

```
isin (emptytree , E, false).
isin (mktree (L , N , R ), N , true ).
isin (mktree (L , N , R ), E , B ) :- diff(N , E), isin (L , E , B 1)
                                       isin (R , E , B 2), or (B 1, B 2, B ).
```

It is important to note that the purpose of our methodology is to help structure PROLOG programs hierarchically using the data abstraction discipline. Our environment does not perform type checking or type inference [Mis84] in PROLOG. For instance, the environment does not ensure that the arguments to an invocation of *append* are *List* objects. Another issue that we do not address is parameterized data types ([Ehr81], [Gog82], [Pad82]).

3. Implementation of Data Types

Since the specifications of data types written using our discipline are executable PROLOG programs, there is no need for a different implementation. However, sometimes for reasons of efficiency and convenience of representation, one might want to implement the data types (and their operators) differently. The user can implement a data type either manually, or automatically with the help of *Refine*.

3.1. Manual Implementation

Implementing a data type (T) consists of (i) picking a representation scheme for the objects of T in terms of the objects of a chosen *representation* type R, and (ii) implementing the constructors and defined operations of T in terms of the constructors and defined operations of R. We discuss the two approaches to implementation in the next two sections.

Suppose we want to implement *Tree* using a sequential representation scheme in which a tree T is represented as a pair (list) of lists $[L_1, L_2]$. L_1 is the in-order enumeration of the nodes of T; L_2 is a list that keeps track of the position of the root of every subtree of T in the in-order enumeration of the nodes of that subtree. In generating L_2 the subtrees of T are themselves considered in a preorder sequence. Such a representation scheme is convenient for checking membership in a tree as well as decomposing trees into and building trees from smaller ones. A few example representations are given below.

$T_1 = \textit{emptytree}$	[[ ], [ ]]
$T_2 = \textit{mktree}(T_1, C, T_1)$	[[C], [1]]
$T_3 = \textit{mktree}(T_2, B, T_2)$	[[C B C], [2 1 1]]
$T_4 = \textit{mktree}(T_3, A, \textit{mktree}(T_1, E, T_1))$	[[C B C A E], [4 2 1 1 1]]

The PROLOG program shown below gives an implementation for *isin* based on the above representation scheme. Note that the implementation is more efficient than the specification of *isin*, because in the latter a given node would be searched in the right subtree even if it was already located in the left subtree. In the following the predicate that implements an operation of *Tree* uses the name of the operation in uppercase so that the correspondence between them is apparent. (We use this convention throughout the paper for convenience although PROLOG does not allow predicate names to begin with an upper case letter.) The implementation also assumes the existence of a data type *Nat* (for natural numbers) with constructors *zero* and *plus* 1.

Implementation of Tree

EMPTYTREE([[ ], [ ]]).

MKTREE([[ ], [ ]], N, [R 1, R 2], [[N | R 1], [1 | R 2]]).
MKTREE([[X 1 | L 1], [X 2 | L 2]], N, [R 1, R 2], [[X 1 | T 1], [I, X 2 | T 2]]):-
    MKTREE([L 1, L 2], N, [R 1, R 2], [T 1, [J | T 2]]), I = *plus* 1(J).

ISIN([[ ], R 1], X, *false*).
ISIN([[X 1 | L 1], R 1], X 1, *true*).
ISIN([[X 1 | L 1], R 1], Y, B) :- *diff*(X 1, Y), *ISIN*([L 1, R 1], Y, B).

3.2. Automatic Transformation of Specifications

Another way to obtain an implementation is by automatically transforming the specification. We refer to an implementation obtained in this fashion a *direct implementation* since they are obtained directly from a specification. We present two methods for transforming a specification into an implementation, both of which are incorporated into *Refine* in our environment. Both the methods assume that there exist implementations (which the user is expected to furnish) for a small subset of operations of the data type, and produce implementation for the rest of the operations. The first method assumes the existence of implementations for the constructors of the type, and produces implementation for each of the defined operations. The second method assumes the existence of implementations for the constructors as well as a (predefined) set of basic defined operations, and derives implementations for the rest of the defined operations in terms of the former. A direct implementation derived using the second method is usually more efficient than the one derived by the first one because we have a larger (and a more versatile) set of predicates that can be used as primitives in the direct implementation. The two methods are described below.

Although a direct implementation might not always be as efficient as a user defined implementation, it is still useful because it is guaranteed to be correct provided the implementation of the operations in terms of which the direct implementation is derived is correct. A direct implementation can be used to test a given representation scheme before designing an implementation that is more efficient but also more intricate than the direct implementation.

Note that the methods of transformation given below can very well be used to refine a program that uses the specification of a data type to one that uses an implementation of the data type. This is because in our method the formalism of a program that uses the specification of a data type is no different from that of the specification itself.

3.2.1. Method 1

A defined operation (predicate) p of a data type T is specified as a relation on the constructor terms of T. The clauses in the specification of p contain the constructors and the defined operations of T. An implementation of p has to define a predicate P that expresses a relation on the constructor terms of the representation type R. Thus, one way of transforming a specification of p into an implementation is the following: Replace the constructors and defined predicates of T in every clause in the specification by the predicates implementing them. For a defined operation the replacement is trivial since both the operation and its implementation take the form of predicates. For a constructor the replacement is not so obvious because a constructor, which is a functor in specification, appears as a predicate in the implementation. Also, a constructor may appear as a part of a nested term.

A constructor term has to be "flattened" before the constructors are replaced by their corresponding implementing predicate. For example, a constructor term $f\ (g(X), h(Y))$ is replaced by a new variable Z, and the conjunction of predicates $F(X1,\ Y1,\ Z),\ G(X,\ X1),\ H(Y,\ Y1)$ is prefixed to the body of the clause in which the constructor term appears. (F, G, and H are the predicates implementing the constructors f, g, and h, respectively.) Thus, a clause of the form

$p(f\ (g(X), h(Y)), L) \text{ :- } q(X,\ Y,\ L),$

where p and q are two defined operations of T is transformed into the following clause:

$P(Z,\ L) \text{ :- } F(X1,\ Y1,\ Z),\ G(X,\ X1),\ H(Y,\ Y1),\ Q(X,\ Y,\ L).$

There is, however, one problem with the above transformation. The execution of the transformed clause can be quite inefficient. This is especially so when the representation scheme is such that there are several different values of R representing the same value of T. In such a case there can be several different values for X and Y for a given Z such that the conjunction of the predicates F, G, and H is satisfiable in the above clause. The behavior of Q is identical on each of the possible values of X and Y because (1) these values are equivalent representations of Z, and (2) the predicates implementing the operations of T are congruent with respect to the class of equivalent representations. However, an execution of the above program in PROLOG would unnecessarily consider each of these values while backtracking. This could even cause infinite computation if the class of equivalent representations of an abstract value is infinite. The above problem can be avoided by fixing, with an appropriate use of "cut", the instantiations generated for X and Y at the end of the processing of the conjunction of predicates replacing the constructor term. One way of doing this is to substitute a new goal $newp(X,\ Y,\ Z)$, where $newp$ is a predicate symbol that does not clash with any other symbols in use, for the conjunction of predicates obtained by flattening the constructor term. *Newp* is defined to be the conjunction of predicates followed by a "cut". Hence, we transform the clause shown above into the following two clauses.

$P(Z,\ L) \text{ :- } newp(X,\ Y,\ Z),\ Q(X,\ Y,\ L).$
$newp(X,\ Y,\ Z) \text{ :- } F(X1,\ Y1,\ Z),\ G(X,\ X1),\ H(Y,\ Y1),\ !.$

It should be noted that the introduction of "cut" as described above is completely automatic, and is not left to the discretion of the user. Although the implementation derived is not in pure PROLOG, the semantics of the program will not be changed. This is because (1) the constructors of T create the values of T uniquely, and (2) the instantiations discarded by the use of "cut" are equivalent to (with respect to a particular value of T) the first instantiations generated by *newp*. Note that condition (1) has to be satisfied for the transformation to work. The reason for this is explained at the end of the section.

A precise description of the transformation steps to be performed on every clause in the specification is given below:

(1) *Constructor replacement*: Every constructor term of t that appears in the clause is subjected to the following constructor-replacement steps.
   (a) *Flatten* the term t into a conjunction of predicates $\overline{G}$ as informally illustrated above. Let $X_1, \cdots, X_n$ be the variables in t, and Z be the new variable in G that the whole term stands for.
   (b) Replace every occurrence of t in the clause by Z.
   (c) Add the new goal *newpred*$(X_1, \cdots, X_n, Z)$ to the front of the body of the clause, where *newpred* is a new predicate symbol (chosen so that it does not clash with any existing predicate symbol in the program) defined by the clause given in step (d) to follow.
   (d) Add the following new clause to the program: *newpred*$(X_1, \cdots, X_n, Z)$:- $\overline{G}$, !.

(2) *Defined operation replacement*: Every occurrence of a defined operation of the data type (such as, *isin*) is replaced by its corresponding implementing predicate (*ISIN*, in our example).

For example, an application of the above transformation steps on the specification of *isin* would result in the direct implementation shown below.

A Direct Implementation of Isin

ISIN$(T, E, false)$:- *newpred*$1(T)$.
ISIN$(T, N, true)$:- *newpred*$2(L, N, R, T)$.
ISIN(T, E, B) :- *newpred*$3(L, N, R, T)$, *diff*(N, E),
    ISIN$(L, E, B1)$, *ISIN*$(R, E, B2)$, *or*$(B1, B2, B)$.

newpred$1(T)$:- *EMPTYTREE*(T), !.
newpred$2(L, N, R, T)$:- *MKTREE*(L, N, R, T), !.
newpred$3(L, N, R, T)$:- *MKTREE*(L, N, R, T), !.

This kind of mechanical transformation cannot be applied in a system like FUNLOG ([SuY84]) for transforming FUNLOG programs into PROLOG programs without nested terms. This is because terms in FUNLOG may contain, as functors, constructors as well as defined operations of data types. Hence, the terms that are being transformed have an equivalence relation (that is not identity) defined on them by a set of axioms. This can create problems when the *semantic unification* (used in [SuY84]) of terms results in more than one unifier. In such a case the corresponding transformed PROLOG program can run into an infinite loop while trying to backtrack. For instance, consider the following clause in FUNLOG, and its corresponding translation by *Refine* into PROLOG. f and g are defined functions on *Integer* whose predicate counterparts are F and G.

$p(f(g(X)))$:- $q(X)$. (In FUNLOG)
$p(Z)$:- $F(X1, Z)$, $G(X, X1)$, $q(X)$. (Transformed into PROLOG)

Let us suppose that $f(g(X)) = 1$ has more than one unifier, one of which satisfies q. (This can happen if f and g are many-to-one.) Then, the execution of the query $p(1)$ may run into an infinite loop in the PROLOG program while the FUNLOG program gives an answer. This can happen if the first value for $X1$ generated by PROLOG does not give the right value for X, and g maps infinitely many values to the same value. In this case PROLOG never gets an opportunity to backtrack to F to get an alternate value for $X1$ since it is busy resatisfying G infinitely many times. Note that in this case one cannot introduce "cut" as described earlier because different possible instantiations generated for X and $X1$ in this case are not equivalent.

3.2.2. Method 2

As mentioned earlier, method 2 produces an implementation assuming that there exist implementations for the constructors and a special set of basic defined operations, called the *decomposers* and *constructor-checkers*, of the type. Hence, this method can only be applied to a special class of data types, called *expressively rich* data types [KaS80], that have the decomposers and constructor-checkers as part of their operation set.

To automate this method it is necessary to identify the operations of the type that can serve as the decomposers and the constructor-checkers of the type. In general, we need a theorem prover for this purpose because it is necessary to check if an operation satisfies the properties that characterize a decomposer (or a constructor-checker). Since our environment has the required theorem proving ability (in the form of *Ind_Prove*) the method can be mechanized without much difficulty.

The decomposers and constructor-checkers of a type permit one to decompose and uniquely determine the structure of the constructor terms of the type. For example, the data type *Tree* introduced earlier would become expressively rich when augmented with the operation *components* (extracts the left subtree, the right subtree, and the root of the tree), *isempty* (checks if a tree is empty), and *isnonempty* (checks if a tree is nonempty). The operation *components*, acts as the decomposer; *isempty* and *isnonempty* can be used as constructor-checkers for the constructors *emptytree* and *mktree*, respectively. The specifications and implementations of these operations are given in Appendix I.

This method derives an implementation for a defined operation with respect to the decomposers and constructor-checkers. It is similar to the previous method except that the constructor replacement is done in terms of the decomposers and constructor-checkers. More details of the method can be found in [HsS84b].

4. Expressing Properties to be Proved

Two kinds of properties concerning data types need to be verified: (1) properties that a specification ought to satisfy, and (2) correctness of an implementation of a data type. Since our theorem prover primitive *Ind_Prove* is within the PROLOG framework, these properties should be formulated as PROLOG clauses as well. While properties in (1) have to be given by the user manually, the verification conditions in (2) can be generated with the assistance of another primitive, *Ver_Cond_Gen* of our system.

Let the property to be proved be $\forall \overline{X} \phi(\overline{X})$, where $\phi(\overline{X})$ is the Horn clause $\forall \overline{Z}(P_1(\overline{X}, \overline{Z}) \wedge \cdots \wedge P_n(\overline{X}, \overline{Z}) \supset Q(\overline{X}, \overline{Z}))$. The formula $\phi(\overline{X})$ is converted into the PROLOG clause:

$$prop(\overline{X}) \text{:-} P_1(\overline{X}, \overline{Z}), \cdots, P_n(\overline{X}, \overline{Z}), Q(\overline{X}, \overline{Z}).$$

with the consequent $Q(\overline{X}, \overline{Z})$ as the last subgoal to be satisfied. ***Prop*** **will be used only as an input to the theorem prover** ***Ind_Prove*****, and not for any other purposes.**

Note that the logical meaning of ***prop*** is not tautologically equivalent to ϕ. This discrepancy does not have any deleterious effect because ***prop*** is only used as a means of representing the property to be proved and not as a predicate in any other clause. Moreover, our theorem prover requires the antecedents (P_i's) to be processed before the consequent (Q). The left-to-right evaluation strategy of PROLOG and the order of predicates in ***prop*** accomplish this ordering requirement automatically.

4.1. Correctness of Abstract Data Types

Proving the correctness of an implementation of a data type consists of first specifying the intended representation scheme, and then showing that the algebra defined by the specification is an homomorphic image of the one defined by the implementation under the intended representation scheme.

The representation scheme is specified by means of two predicates: *representation invariant*, and *representation equivalence*. The representation invariant characterizes the (sub)set of values (of the representation type) that are permitted to represent the abstract values. The representation equivalence characterizes an equivalence relation that relates all valid representation values representing the same abstract value. In the *Tree* implementation, for example, the representation invariant is expressed in two parts (for convenience) both of which have to be satisfied. *Inv* 1 expresses the constraint that "if a pair of lists $[L_1, L_2]$ is a valid representation of a tree then L_1 and L_2 have the same length". *Inv* 2 expresses the constraint necessary on the numbers in L_2: the first element in L_2 which denotes the position of the root in L_1 has to be a number between 1 and the length of L_2; the two segments of L_2 that represent the two subtrees should also satisfy this constraint. The representation equivalence in this case is just "equality on pairs of lists" because every tree has a unique representation as a pair of lists.

```
inv1([ ], [ ], true).
inv1([X1 | L1], [X2 | L2], true) :- inv1(L1, L2, true).
inv1([ ],[X2 | L2], false).
inv1([X1 | L1], [ ], false).

inv2([ ], true).
inv2([X | L], true) :- length([X | L], N), ge(X, 1), ge(N, X),
                       prefix(L, X, L1), postfix(L, X, L2),
                       inv2(L1, true), inv2(L2, true).

repequiv([A1, A2], [B1, B2], true) :- A1 = A2, B1 = B2.
```

repequiv ([A 1, A 2], [B 1, B 2], *false*) :- *diff*(A 1, A 2), $B\,1 \neq B\,2$.
repequiv ([A 1, A 2], [B 1, B 2], *false*) :- $A\,1 = A\,2$, $B\,1 \neq B\,2$.
repequiv ([A 1, A 2], [B 1, B 2], *false*) :- *diff*(A 1, A 2), $B\,1 = B\,2$.

Given the specification of a data type, an implementation of it, and a definition of the representation invariant and representation equivalence, *Ver_Cond_Gen* generates the verification conditions as described below.

Verification Conditions for the Constructors

These conditions ensure that the values constructed using the predicates implementing the constructors satisfy the invariant. Thus, there is a verification condition for every predicate implementing a constructor that says "if the arguments to the predicate satisfy the invariant, then so does the value constructed by the predicate". For example, the verification conditions which ensure that the implementation satisfies the constraint expressed by *inv* 1 is given below.

invprop 1(L 1, L 2) :- *EMPTYTREE*([L 1, L 2]), *inv* 1(L 1, L 2, B), B=*true*.
invprop 2(L 1, L 2, R 1, R 2) :-
    inv 1(L 1, L 2, *true*), *inv* 1(R 1, R 2, *true*),
    MKTREE([L 1, L 2], N, [R 1, R 2], [T 1, T 2]), *inv* 1(T 1, T 2, B), B=*true*.

Just as a reminder, *invprop* 1 and *invprop* 2 are two clauses representing properties to be proved by the theorem prover. The logical meaning corresponding to *invprop* 1 is:

EMPTYTREE([L 1, L 2])$\wedge$*inv* 1(L 1, L 2, B)$\supset B$=*true*,

where all the variables are universally quantified.

Verification Conditions for the Defined Operations

These conditions ensure that the implementation satisfies the homomorphism property. There is a verification condition corresponding to every clause in the specification of a defined operation. The verification condition for each clause is derived from the clause as follows:

(1) Subject every constructor term in the clause to the following steps:
    (a) Flatten the constructor term, and attach the resulting conjunction of predicates to the body of the clause. (Same as Step 1.a in Section 3.2.1.)
    (b) Replace every occurrence of the term by a new variable Z. (Same as Step 1.b in Section 3.2.1.)

(2) Replace every defined operation in the clause by its corresponding implementing predicate. (Same as Step 2 in Section 3.2.1.)

(3) Replace every equality on the representation values by the representation equivalence.

(4) Express the logical implication characterized by the PROLOG clause as a conjunction of goals according to the form required by *Ind_Prove*.

The verification conditions generated by *Ver_Cond_Gen* for the operations *isin* of *Tree* are given below, recall that *isin* is defined as:

```
isin(emptytree, E, false).
isin(mktree(L, N, R), N, true).
isin(mktree(L, N, R), E, B) :- diff(N, E), isin(L, E, B1)
                                isin(R, E, B2), or(B1, B2, B).
```

Verification Conditions for ISIN

```
prop1(L1, R1, E) :-
        EMPTYTREE([L1, R1]), ISIN([L1, R1], E, B), B = false.
prop2(L1, R1, N, L2, R2, N) :-
        MKTREE([L1, R1], N, [L2, R2], T), ISIN(T, N, B), B = true
prop3(L1, R1, N, R2, L2, E) :-
        MKTREE([L1, R1], N, [L2, R2], T), ISIN(T,N  Result1),
        diff(N, E), ISIN([L1, R1], E, B1), ISIN([L2, R2], E, B2),
        or(B1, B2, Result2), Result1 = Result2.
```

5. The Theorem Prover Ind_Prove

In this section we present a brief and informal description of the theorem proving method used in the verification phase of the environment. A more detailed description can be found in [HsS84a].

The main task that our theorem prover needs to perform is to prove properties with universally quantified variables. PROLOG, which can be regarded as a prover for proving *existentially* quantified properties, is unable to fulfill this task. The conventional way of dealing with universally quantified variables in a refutational-type theorem prover (such as PROLOG) is to treat them as skolem constants (after negating the target sentence, of course) (e.g. [Sti84]). Such a method does not work satisfactorily if the domain of variables are defined inductively (such as *List*) since skolem constants cannot be unified with any of the constructors.

We solve this problem by introducing a deductive theorem proving method for first order inductive theory representable in Horn clauses. The basic inference mechanism in the theorem prover is backward deduction, the same as in PROLOG. The first major notion we introduce is a way of handling unsatisfiable goals whose unsatisfiability is due to the appearance of skolem constants. For example, given a goal is *append*(*sk*,[ ],*X*), where *sk* is a skolem constant, *append*(*sk*,[ ],*X*) is not satisfiable since *sk* unifies with neither [ ] nor [*A* | *L*]. However, we know that this goal ***should*** be satisfiable since *sk*, being a list, has to be either [ ] or [*A* | *L*] for some *A* and *L*. We handle this problem by delaying the evaluation of this goal until later by temporarily unifying *X* with a list *l* which satisfies *append*(*sk*,[ ],*l*), without worrying about what *l* really is. In our notation, *X* is unified with Ω(*l*:*append*(*sk*,[ ],*l*)). This method of delaying the evaluation of goals and binding variables is called *Ω-binding*, and *append*(*sk*,[ ],*l*) is called the *Ω-constraint* of *X*. A goal which can be satisfied in such a way is called *Ω-satisfiable*. A similar notion has also been used by Kornfeld ([Kor83]) for enriching the unification to include equational axioms.

At the end of processing all the goals in the theorem to be proved (expressed as a Horn clause), the Ω-constraints will be put together in a certain way to produce a *Lemma*. The need of Ω-binding arises because the universally quantified variables have to be skolemized.

However, Ω-binding does not *solve* the problem created by skolemization but merely postpones the time of decision making to when the lemma is generated. Therefore it is important to have some mechanism for ensuring that the Ω-constraints thus generated are indeed "simpler" than the original problem. One way to achieve that is to delay the time of skolemization. For this purpose, we introduce a technique called *skolemize by need*. Under this method, we treat universally quantified variables as free variables, and skolemize them only when necessary (that is, when a value for that variable has to be determined). This technique produces three effects: (1) It prevents the variables from being instantiated indefinitely. (2) The variables are skolemized, automatically, according to the inductive structures of the constructing terms. (3) The Ω-bindings produced by such process usually contain unsatisfiable goals which are simpler than the original goals, and these unsatifiable goals will sometimes lead to appropriate induction hypothesis. Because of the delaying of skolemization, the skolem constants so produced no longer cover the whole domain over which the universally quantified variables are defined. Therefore we also provide, in the prover, a mechanism of generating a complete set of skolemizations (not just one) to ensure the completeness. The method is very similar to the PROLOG backtracking mechanism, only we are more selective of the choice points for backtracking. We also use a *limited forward chaining* mechanism for producing potential induction hypothesis from the Ω-constraints.

The prover can be (and has been) built without too much difficulty in PROLOG since PROLOG already provides the basic mechanisms such as unification and backward deduction. However, we do need to add an additional *occur check* mechanism in PROLOG ([Pla84]) to avoid overlapping of variables which may lead to inconsistency.

Due to the lack of space, we refer the interested readers to [HsS84a] for a detailed description of the theorem proving method. Some examples of the proofs are given in Appendix II.

6. Conclusions

We have investigated the use of PROLOG as a uniform environment to carry out all three phases - specification, implementation, and verification - of systematic program development. For the specification part, we have proposed a methodology of designing data types in PROLOG that represents the constructors as functors and the defined operations as predicates. This approach to data types not only provides executable specifications, but also eliminates the need for an elaborate unification algorithm encountered in the other approaches. We have given several transformation techniques for transforming specifications into implementations. In particular, we have presented an automatic transformation method in which "cut" is produced mechanically without changing the semantics of the program. In order to verify PROLOG programs within PROLOG's own mechanism, we developed a deductive theorem proving method for the first order inductive theory representable in Horn clauses. This theorem prover, which can be easily incorporated into PROLOG, enables us to prove universally quantified inductive properties of PROLOG programs. It also does not employ, explicitly, any inductive inference rule. Methods for generating verification conditions are also given.

There is plenty of scope for further work in the area. In the data type development area we need to study more about the tradeoffs involved in representing the operations in functional style as opposed to in a predicate style. It would be interesting to extend our transformation method to data types (such as *Set*) where the constructors exhibit standard properties such as associativity and commutativity. It appears that when the equivalence class of constructor terms defined by the properties is finite, the transformation method can be extended by generating one clause for every term in the equivalence class. It would also be interesting to see if more sophisticated transformation techniques ([Sri83]) can be used to obtain better forms of direct implementation. In the theorem proving area we need to extend the form of the properties that can be proved beyond the Horn clause form that can currently be handled, such as incorporating Stickel's complete inference rule for PROLOG ([Sti84]) into our system. It is also necessary to study if the method can be used to disprove properties.

Acknowledgements

We are thankful to Eric Johnson, C. Mohan, Jean-Luc Remy, and David S. Warren for their helpful comments on earlier drafts of the paper.

1. References

[Dav82] R. E. Davis, "Runnable Specification as a Design Tool", in *Logic Programming,* K. L. Clark and S. Tarnlund, (eds.), Academic Press, January 1982, 141-152.

[Ehr81] H. Ehrig, "Algebraic Theory of Parameterized Specifications with Requirements", *6th CAAP*, 1981.

[Fag84] F. Fages, "Associative-Commutative Unification", *7th Conf. on Automated Deduction*, Nappa Valley, CA, May, 1984, 194-208.

[GTW78] J. A. Goguen, J. W. Thatcher and E. G. Wagner, "Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types", in *Current Trends in Programming Methodology,* vol. IV Data Structuring, R. T. Yeh, (ed.), Prentice Hall (Automatic Computation Series), Englewood Cliffs, NJ, 1978.

[GoT79] J. A. Goguen and J. J. Tardo, "An Introduction to OBJ: A Language for Writing and Testing Formal Algebraic Program Specifications", *Proceedings of the Conference on Specification of Reliable Software*, Cambridge, MA 02139, 1979.

[Gog82] J. A. Goguen, "Parameterized Programming", *Proceedings of the Workshop on Reusability in Programming,* 1982.

[GuH78] J. V. Guttag and J. J. Horning, "The Algebraic Specification of Abstract Data Types", *Acta Informatica,* **10**, 1 (1978), 27-52.

[HaT82] A. Hansson and S. Tarnlund, "Program Transformation by Data Structure", in *Logic Programming,* K. L. Clark and S. Tarnlund, (eds.), Academic Press, January 1982, 141-152.

[HsS84a] J. Hsiang and M. K. Srivas, "On Proving First Order Inductive Properties in Horn Clauses", Technical Report 84/75, SUNY at Stony Brook, Stony Brook, NY 11794, 1984.

[HsS84b] J. Hsiang and M. K. Srivas, "A PROLOG Environment for Developing and Reasoning about Data Types", Technical Report 84/074, SUNY at Stony Brook, Stony Brook, NY 11794, 1984.

[HuH80] G. Huet and J. M. Hullot, "Proofs by Induction in Equational Theories with Constructors", *21st IEEE Symposium on Foundations of Computer Science*, 1980, 797-821.

[HuO80] G. Huet and D. C. Oppen, "Equations and Rewrite Rules: A Survey", in *Formal Languages: Perspectives and Open Problems*, R. Book, (ed.), Academic Press, 1980.

[Hul80] J. M. Hullot, "Canonical Forms and Unification", *5th Conference on Automated Deduction*, Les Arcs, France, 1980, 318-334.

[KaS80] D. Kapur and M. K. Srivas, "Expressiveness of the Operation Set of a Data Abstraction", *Seventh Annual ACM Symposium on Principles of Programming Languages*, Las Vegas, Nevada, January 28-30, 1980, 139-153.

[Kor83] W. A. Kornfeld, "Equality in Prolog", *Proc. 8th IJCAI*, Karlsruhe, Germany, August 1983, 514-519.

[Mis84] P. Mishra, "Towards a Theory of Types in Prolog", *1984 International Symposium on Logic Programming*, Atlantic City, New Jersey, Feb. 6-9, 1984, 289-298.

[Mus80] D. R. Musser, "Abstract Data Types in the AFFIRM System", *IEEE*, **1**, 6 (Jan. 1980), .

[Pad82] P. Padawitz, "Correctness, Completeness and Consistency of Equational Data Type Specifications", in *Ph.D. Thesis*,, Technische Universitat, Berlin, 1982.

[Pla84] D. A. Plaisted, "The Occur-Check Problem in Prolog", *1984 International Symposium on Logic Programming*, Atlantic City, New Jersey, Feb. 6-9, 1984, 272-280.

[Sri83] M. K. Srivas, "A Rewrite Rule Based Approach to Program Transformation", *The Rewrite Rule Laboratory Workshop*, Schenectady, NY 12345, September 1983.

[Sti81] M. E. Stickel, "A Unification Algorithm for Associative-Commutative Functions", *J. ACM*, **28**, (1981), 233-264.

[Sti84] M. E. Stickel, "A Prolog Technology Theorem Prover", *1984 International Symposium on Logic Programming*, Atlantic City, New Jersey, Feb. 6-9, 1984, 212-219.

[SuY84] P. A. Subrahmanyam and J. You, "Conceptual Basis and Evaluation Strategies for Integrating Functional and Logic Programming", *1984 International Symposium on Logic Programming*, Atlantic City, New Jersey, February 6-9, 1984, 144-153.

Appendix I

Specification of Tree

Constructors

$emptytree: \rightarrow Tree$
$mktree: Tree \times Node \times Tree \rightarrow Tree$

Defined Operations

$components: Tree \times Tree \times Node \times Tree$
$isempty \quad : Tree \times Boolean$
$isnonempty: Tree \times Boolean$
$isin \quad : Tree \times Node \times Boolean$

components (emptytree , error , error , error).
components (mktree (L , N , R), L , N , R).

isempty (emptytree , true).
isempty (mktree (L , N , R), false).

isnonempty (emptytree , false).
isnonempty (mktree (L , N , R), true).

isin (emptytree , E , false).
isin (mktree (L , N , R), N , true).
isin (mktree (L , N , R), E , B) :- diff(N , E), isin (L , E , B_1)
isin (R , E , B_2), or (B_1, B_2, B).

Implementation of Tree

EMPTYTREE ([[ ], [ ]]).

MKTREE and ISIN are given in Section 3.1.

ISEMPTY ([[ ], [ ]], *true*).
ISEMPTY ([X | L], [I | R]], *false*).

ISNONEMPTY ([[ ], [ ]], *false*).
ISNONEMPTY ([X | L], [I | R]], *true*).

COMPONENTS (T , Left , N , Right) :-
    LTREE (T , Left), *NODEOF* (T , N), *RTREE* (T , Right).

LTREE ([[ ], [ ]], *error*).
LTREE ([L_1, [I | L_2]], [T_1, T_2]) :-
    prefixof(L_1,I, T_1), *prefixof*(L_2,I, T_2).
{prefixof(L ,I, T) extracts a list of length I-1 at the head of L}

RTREE ([[ ], [ ]], *error*).
RTREE ([L_1, [I | L_2]], [T_1, T_2]) :-
    postfixof(L_1,I, T_1), *postfixof*(L_2,I, T_2).
{postfixof(L ,I, T) extracts the tail of L starting from (I+1)th element}

$NODEOF([[\,], [\,]], error)$.
$NODEOF([L, [I \mid R]], N)$:- $ithelementof(L, I, N)$.
{*ithelementof* (L, I, N) *checks if the ith element of L is N*.}

Appendix II

In the following we show the proof of some of the verification conditions (derived in section 4) for the implementation of *Tree*. We present the proof of *invprop* 2 and *prop* 3 for the implementation of *isin*. The prover generates a set of instantiations, each with a *premise* (further constraint on the instantiations), and a *lemma*. Assuming that the instantiation is $\overline{X}_0$, the logical interpretation of the triplet is:

For every $\overline{X}_0$ which also satisfies *Premise*, if *Lemma* is true, then *prop* $(\overline{X}_0)$ is true.

1. Proof of Invprop2

In this case the proof generates a well-spanned set of four instantiations for the input variables. For the first and second instantiation the lemma generated is *true*. For the third and the fourth case, the lemma generated is an instance (for smaller argument values) of the property being proved, and hence forms the induction hypothesis.

$Invprop\,2(L_1, L_2, R_1, R_2)$:-
    $inv\,1(L_1, L_2, true)$, $inv\,1(R_1, R_2, true)$,
    $MKTREE([L_1, L_2], N, [R_1, R_2], [T_1, T_2])$, $inv\,1(T_1, T_2, B)$, $B = true$.

Instantiations	Premise	Lemma
$([\,],[\,],[\,],[\,])$	*true*	*true*
$([\,],[\,],[Y_1 \mid R_1],[Y_2 \mid R_2])$	*true*	*true*
$([X_1 \mid L_1],[X_2 \mid L_2],[\,],[\,])$	*true*	$invprop\,2(L_1, L_2, [\,], [\,])$
$([X_1 \mid L_1],[X_2 \mid L_2],[Y_1 \mid R_1],[Y_2 \mid R_2])$	*true*	$invprop\,2(L_1, L_2, R_1, R_2)$

2. Proof of Prop3

For *prop* 3, *Ind_Prove* generates nine sets of instantiations for the variables L and R which span the domain $List \times List$. Every Lemma generated is either trivially true, or is implied by the induction hypothesis (derived from forward chaining), or can be proved to be true by applying *Ind_Prove* on it again. In the following the symbol $\neq$ is used as a synonym for the operation *diff* on nodes.

$prop\,3(L_1, R_1, N, L_2, E)$:- $TREE([L_1, R_1], N, [L_2, R_2], T)$, $ISIN(T, N, Result\,1)$,
    $N \neq E$, $ISIN([L_1, R_1], E, B_1)$, $ISIN([L_2, R_2], E, B_2)$,
    $or(B_1, B_2, Result\,2)$, $Result\,1 = Result\,2$.

Instantiations	Premise	Lemma
$L_1 \leftarrow [\,], N \leftarrow \hat{N}, E \leftarrow \hat{E}$		
$R_1 \leftarrow \hat{R}_1, R_2 \leftarrow \hat{R}_2$		
(a) $L_2 \leftarrow [\,]$	$\hat{N} \neq \hat{E}$	*true*

(b) $L_2 \leftarrow [\hat{E} \mid \hat{L}_2]$	$\hat{N} \neq \hat{E}$	*true*
(c) $L_2 \leftarrow [\hat{Y} \mid \hat{L}_2]$	$\hat{N} \neq \hat{E}$, $\hat{Y} \neq \hat{E}$	*true*

$L_1 \leftarrow [\hat{E} \mid \hat{L}_1], N \leftarrow \hat{N}, E \leftarrow \hat{E}$
$R_1 \leftarrow \hat{R}_1, R_2 \leftarrow \hat{R}_2$

(a) $L_2 \leftarrow [\,]$	$\hat{N} \neq \hat{E}$	*true*
(b) $L_2 \leftarrow [\hat{E} \mid \hat{L}_2]$	$\hat{N} \neq \hat{E}$	*true*
(c) $L_2 \leftarrow [\hat{Y} \mid \hat{L}_2]$	$\hat{N} \neq \hat{E}$	*true*

$L_1 \leftarrow [\hat{X} \mid \hat{L}_1], N \leftarrow \hat{N}, E \leftarrow \hat{E}$
$R_1 \leftarrow \hat{R}_1, R_2 \leftarrow \hat{R}_2$

(a) $L_2 \leftarrow [\,]$	*true*	$prop\,3(\hat{L}_1, \hat{N}, [\;], \hat{E})$
(b) $L_2 \leftarrow [\hat{Y} \mid \hat{L}_2]$	$\hat{X} \neq \hat{E}$	$prop\,3(\hat{L}_1, \hat{N}, [\hat{Y} \mid \hat{L}_2], \hat{E})$
(c) $L_2 \leftarrow [\hat{E} \mid \hat{L}_2]$	$\hat{X} \neq \hat{E}$, $\hat{N} \neq \hat{E}$	$TREE([\hat{L}_1, \hat{R}_1], \hat{N}, [[\hat{E} \mid \hat{L}_2], \hat{R}_2], T)$, $ISIN(T, \hat{E}, B)$, $B = true$.

Proof of the Last Lemma

To complete the proof of *prop* 3 we have to prove the last lemma generated above. We use *Ind_Prove* once more to do this. The lemma is expressed as a new property to be proved as follows. Note that the relevant constraints in the *Premise* should be made as a part of the new property to be proved.

$$newprop(L_1, R_1, N, L_2, R_2, E) \text{ :- } N \neq E, TREE([L_1, R_1], N, [[E \mid L_2], R_2], T),$$
$$ISIN(T, E, B),\ B = true.$$

Instantiations	Premise	Lemma
$([\;], \hat{R}_1, \hat{N}, \hat{L}_2, \hat{R}_2, \hat{E})$	*true*	*true*
$([\hat{X}_1 \mid \hat{L}_1], \hat{N}, \hat{L}_2, \hat{R}_2, \hat{E})$	*true*	$newprop(\hat{L}_1, \hat{R}_1, \hat{N}, \hat{L}_2, \hat{R}_2, \hat{E})$