A broader interpretation of logic in logic programming

A Broader Interpretation of Logic in Logic Programming

Alan Bundy

DAI Research Paper No. 388

May 20, 1988

Invited talk at the
Fifth International Logic Programming Conference/
Fifth Symposium on Logic Programming.
To appear in the proceedings.

Department of Artificial Intelligence
University of Edinburgh
80 South Bridge
Edinburgh EH1 1HN
Scotland

A Broader Interpretation of Logic in Logic Programming *

Alan Bundy

Abstract

We argue that the restriction of logic programs to sets of Horn clauses, even with negation as failure, is an unacceptable inhibition to programmers' expressiveness and forces them to make premature procedural commitments. Programmers should be permitted to use the full power of logic when specifying logic programs. In particular, we give examples of the need for functions, quantification, disjunction and predicate variables. Unfortunately, direct interpretation of programs written in such broader logics presents severe difficulties. One route to solving this problem is to treat the broader logic programs as specifications and to refine them into programs before executing them. Horn clauses might be the target programming language. Some refinement techniques can be borrowed from formal methods in software engineering. This suggests a modification of Kowalski's famous slogan to "Algorithm = Refined(Logic) + Control". We illustrate these ideas by describing the Nuprl system for program synthesis and the work we are doing to guide the process of synthesis by the use of proof plans.

Key words and phrases. Logic programming, logic, non-Horn clauses, constructive logic, program synthesis, Nuprl, proof plans.

1 The Vision of Logic Programming

In his missionary works introducing logic programming to the world, *eg* [Kowalski 79b,Kowalski 79a], Kowalski used the slogan "Algorithm = Logic + Control". The normal interpretation of this slogan is that people can describe their problems in the language of predicate logic, *without regard to how this description might be used to solve their problem*, and then a clever interpreter will run their logical description as an algorithm to solve their problem.

This is a wonderful vision. It frees users from thinking in terms of their solution and permits them merely to describe their problem, thus making the power of computing available to the non-programmer. The computer is left to construe the problem description as a computer program.

Unfortunately, we all know that life is not as simple as this.

1. The major embodiment of logic programming, Prolog[1] , has a rather simple minded interpreter, which has to be given a version of the problem description which is readily interpreted as a procedure. In particular, the program must be written in what Lloyd, [Lloyd 87], calls *normal programs*, *ie* Horn clauses plus negation as failure. Some researchers are working on more intelligent interpreters which can cope with more than just normal programs, but ...

*I am grateful to Paul Brna, Bill Clocksin, John Lloyd, Richard O'Keefe, Bob Harper, Dale Miller, Jan Newmarch, Lincoln Wallen and Alan Smaill, for feedback on and conversations about this paper. Some of the research reported here was supported by SERC grant GR/E/44598, Alvey/SERC grant GR/D/44270 and an SERC Senior Fellowship to the author.

[1]References to Prolog in this paper are to versions using the standard interpreter and built-in predicates, *eg* Quintus Prolog or Micro-Prolog, but not to versions with non-standard features, *eg* IC-Prolog or NuProlog.

2. ...for all of the interpreters proposed so far there are certain predicate logic problem descriptions which cannot be interpreted efficiently. For some interpreters there are problem descriptions which cannot be interpreted at all. These problem descriptions must be transformed into an algorithmic form before interpretation, *eg* from full predicate logic into normal programs.

3. Even full predicate logic may not be sufficiently expressive to allow the problem to be described in a natural way. Some researchers are working on alternative kinds of logic with greater expressive power.

4. Users may not find it easy to express their problems in any logical formalism either because their ideas are not sufficiently worked out to allow precise description or because they find logic an unnatural medium for expressing their ideas.

In this paper we will be concerned mainly with points 2 and 3 above.

2 Formal Methods in Software Engineering

Many of these issues also arise in formal methods in software engineering, but different terminology is used to describe them.

- The logical description of the problem is called a *specification*. There is some recent work on directly interpreting such specifications, called *animation*. But animation is not intended to replace programming; it is merely a technique for checking the correctness of the specification.

- The specification is turned into a program by *refinement*. Refinement can, for instance, be effected by the application of *transformation* rules or by a process of *synthesis*. Transformation rules can also be used to form a more efficient program from a less efficient one.

- Traditionally the program produced was in an imperative language, but recently there has been a lot of interest in *functional programming languages*, which can be readily seen as a variety of logic programming languages.

- There has been a lot of work in software engineering on *designer logics* to provide expressive power for particular applications, *eg* temporal logics to reason about the relative timing of different processes/hardware in parallel processing.

- The problem of forming the specification of a program is called *requirements capture*, but this is a relatively neglected area of software engineering.

We will be concerned to combine the best ideas from logic programming and software engineering in the solution of the problems facing both communities. We will also bring in relevant ideas from artificial intelligence (AI).

3 Comparing Logic Programming with Formal Methods

Here are some of the ways in which I think that these three communities can benefit from each other.

- The use of logic both to specify a program and to encode it can simplify the task of refinement; factoring out the problem of translating between rival formalisms and allowing concentration on the problem of deriving better algorithms. The tasks of synthesis, transformation and verification become intimately related, and tend to merge into each other.

- The work on animation could benefit from the clever interpreters that have been written for logic programs. These illustrate that specifications written with little regard from their algorithmic behaviour can still be run reasonably efficiently.
- The work on transformation of logic programs could benefit from the refinement techniques developed in formal methods, which has taken this problem as central and done far more work on it.
- The work on using more expressive logics has been largely orthogonal between the three communities. The logic programming community has concentrated on representing knowledge within predicate logic, but has also extended predicate logic by the incorporation of meta-knowledge. The formal methods community has experimented with a variety of non-standard logics for describing programming situations and the AI community has experimented with non-standard logics for representing common-sense knowledge. These approaches need to be combined to get the best of all worlds.
- The formal methods techniques generate proof obligations, and these proofs need to be guided both to avoid the combinatorial explosion and to produce efficient programs. AI theorem proving research can inform this work.
- The problem of requirements capture can benefit from AI research on knowledge elicitation and on intelligent front ends.

We will be particularly concerned with the application of a program synthesis technique from formal methods to the derivation of logic programs. This program synthesis technique will require the use of AI search control techniques to guide the proving of the proof obligations that arise. We will outline a search control technique we are developing.

4 The Need to Go Beyond Horn Clauses

What is it about sets of Horn clauses that make them especially suitable for describing procedures?

- In general, the interpreters of logic programs are automatic theorem provers. Most theorem provers have been designed to operate on sets of clauses. They tend to be particularly efficient on Horn clauses.
- Horn clauses have a straightforward procedural interpretation.
- Any sound and complete clausal theorem prover can be used to interpret any set of Horn clauses. This interpretation will terminate and can be viewed as a correct execution of the Horn clauses under the above procedural interpretation.
- Under this procedural interpretation it can be shown that Horn clauses are sufficient to implement any algorithm, ie Horn clauses are Turing complete, [Tarnlund 77].

— no function

However, in practice, programmers have felt the need to go beyond Horn clauses in writing logic programs. Prolog has many non-Horn clause features built into it in order to meet these felt needs, for instance: negation as failure (*not*), set extension (*setof*), meta-level predicates (*var*, *assert*, *call*, = .., *etc*). We will call this language *extended Horn clauses*. Its non-predicate logic features extend even Lloyd's normal programs. However, even extended Horn clauses are not felt to be sufficient by Prolog programmers. Examination of practical Prolog programs reveals

many apparently non-declarative, programming tricks which, in fact, have a declarative but non-Horn clause interpretation[2].

For instance, the *failure driven loop* is generally considered to be a programming hack which violates the declarative reading of Prolog, but which is a necessary evil in practical programming. An example of a failure driven loop is the following definition[3] of *list_pioneers*/0.

$$\begin{array}{lll} list\_pioneers & :- & pioneer(X),\ write(X),\ nl,\ fail. \\ list\_pioneers. & & \end{array}$$

Given a database like:

$$\begin{array}{r} pioneer(kowalski). \\ pioneer(colmerauer). \\ pioneer(hayes). \\ pioneer(cordell\_green). \end{array}$$

a call to *list_pioneers* will instantiate X by a call of *pioneer*/1, write the value of X and a newline and then fail. Since *write*/1 and *nl*/0 are not resatisfiable the failure will cause *pioneer*/1 to be resatisfied. This cycle will be repeated until there are no more ways of satisfying *pioneer*/1, at which point the first clause will fail and the second will be called and succeed trivially.

Prolog programmers have found failure driven loops to be useful for performing an action for each way of satisfying some goal. Until the introduction of *setof*/3 as a system predicate[4], they were a necessary evil. For instance, they were required to implement the early versions of *setof*/3, which were common in users' personal utility files. The introduction of *setof*/3 has made most uses of failure driven loops strictly speaking unnecessary. For instance, *list_pioneers* above could be defined as:

$$\begin{array}{rcl} list\_pioneers & :- & setof(X, pioneer(X), XList), \\ & & apply\_list(write\_nl, XList). \\ write\_nl(X) & :- & write(X),\ nl. \\ apply\_list(P, []). & & \\ apply\_list(P, [Hd|Tl]) & :- & apply(P, Hd),\ apply\_list(P, Tl). \\ apply(P, X) & :- & Goal = ..[P, X],\ call(Goal). \end{array}$$

where *apply_list*(P, L) calls $P(Hd)$ for each member, Hd, of the list, L, and *apply*(P, X) calls $P(X)$. Failure driven loops are still useful for avoiding the overhead of defining *apply_list*/2, *apply*/2, *etc*, and for avoiding formation of a possibly very long list of variable bindings. They are still essential if the list of variable bindings would be infinite.

However, failure driven loops are not as ugly as they seem. They can be seen as an attempt to extend the logic: 'faking' non-Horn clause features by exploiting the behaviour of the Prolog interpreter on Horn-clauses. The alternative is to provide these non-Horn clause features directly, so that users can have the procedural power they want without violating the declarative semantics.

[2] And, of course, many which do not.

[3] In the example programs in this paper we follow the Edinburgh Prolog conventions, *eg* that atoms starting with capital letters are variables and those starting with lower case letters are constants.

[4] *setof*$(X, P, XList)$ can, of course, be interpreted as a perfectly respectable mathematical object for forming the extensions of sets. In mathematics set extension is written as:

$$XList = \{X : P(X)\}$$

setof/3 is equivalent to set extension when $XList$ is non-empty and fails otherwise.

The logical extension required in this case is universal quantification[5]. *list_pioneers*/0 could then be defined as:

$$list\_pioneers \ :- \ \forall X(pioneer(X) \longrightarrow (write(X) \wedge nl)) \tag{1}$$

which has a declarative semantics provided that $write(X)$ and nl can be given one[6].

Lloyd and Topor, [Lloyd & Topor 84] have shown that logic programs like (1) above can be translated into normal programs. In fact, any logic program of the form $A :- W$, where W is an arbitrary predicate logic formula and A is a positive literal, can be put into clausal form, and the negative literals interpreted as negation as failure applied to a positive literal. So the only extension to Horn clauses that is strictly required is negation as failure. In the case of (1) this translation yields:

$$\begin{aligned} list\_pioneers \ &:- \ not\ foo. \\ foo \ &:- \ pioneer(X),\ not\ write(X). \\ foo \ &:- \ pioneer(X),\ not\ nl. \end{aligned}$$

which is declaratively correct, but which will mess up the order of writing pioneers and newlines. *foo* would be better defined as:

$$foo \ :- \ pioneer(X),\ not\ write\_nl(X).$$

This grouping of *write*/1 and *nl*/0 together before the application of the Lloyd-Topor process is in line with a general principle of Lloyd's of separating the non-declarative parts of the program from the declarative parts.

Unfortunately, the Lloyd-Topor translation process does not always produce executable programs. Negation as failure is not always safe on non-ground literals, so goals containing only non-ground negative literals cannot always be safely executed. Such computations are said to *flounder*. Furthermore, the Lloyd-Topor process gives only an extensional interpretation of universal quantification, *ie* $\forall X.p(X)$ is unpacked into an exhaustive search for values of X for which $p(X)$ is provable. Some problems call for an intensional interpretation of universal quantification in which $\forall X.p(X)$ is proved in general, *eg* from $\forall X.q(X)$ and $\forall X(q(X) \longrightarrow p(X))$. Note that this requires universal quantification in the head of a clause, and not just the body. λProlog, [Miller & Nadathur 88], provides such an intensional interpretation of universal quantification.

A further example of this phenomenon of extensions to the Horn clause logic being required to cure violations of the declarative semantics, can be seen in the procedure *apply*/2 defined above and repeated here.

$$apply(P, X) \ :- \ Goal = ..[P, X],\ call(Goal). \tag{2}$$

The definition of *apply*/2 goes not only beyond Horn clauses, but also beyond predicate logic, on two counts: the use of the variable *Goal* to stand for both a term and a formula; and the use of the meta-predicate = ../2. Warren,[Warren 81], shows that *apply*/2 can be alternatively defined by an unbounded set of Horn clauses, *eg*

$$apply(write\_nl, X) \ :- \ write\_nl(X).$$

[5]The logical connectives $\longrightarrow$ and $\wedge$ used in this example do not take us outside Horn clauses, and so do not, strictly speaking, constitute an extension.

[6]Giving these formulae a declarative semantics does present a problem since they are called mainly for their side-effect and any declarative reading is nominal. But this is another well known problem which is independent of the failure driven loop problem.

where the double use of the symbol *write_nl* can be seen as 'coincidental' overloading rather than as a violation of the syntactic rules of predicate logic[7]. However, Warren admits that a more congenial solution is to extend the logic of logic programming to second order and allow predicate variables, *eg* $P(X)$. No definition of $apply/2$ is then required, and $apply\_list/2$ can be directly defined as:

$$\begin{aligned} apply\_list(P, [\,]). \\ apply\_list(P, [Hd|Tl]) \quad &:- \quad P(Hd),\ apply\_list(P, Tl). \end{aligned}$$

which is easier to read and more directly reflects the user's intentions.

However, this definition of $apply\_list/2$ suffers from one of the same problems that afflicted our original definition of $apply/2$,(2) above, namely it uses the variable P to stand for both a term and a predicate. To cure this problem we need to recognise that the recursion over lists in $apply\_list/2$ is really providing a form of bounded universal quantification. If this were provided directly by extending the logic then, not only would there be no overloading of P, but the readability and correspondence to the user's intentions would be even better. No definition of $apply\_list/2$ would then be required, since $list\_pioneers/0$ could be directly defined as:

$$\begin{aligned} list\_pioneers \quad &:- \quad setof(X, pioneer(X), XList), \\ &\quad\ \ \forall X \in XList(write(X) \wedge nl). \end{aligned}$$

Several researchers are working on the extension of logic programming to provide features like those mentioned above. For instance, NU-Prolog, [Naish 86], allows arbitrary formulae in the body of a clause. It uses the Lloyd-Topor translation process, and stops with a warning message if the computation flounders. λProlog provides higher-order functions and λ terms and extends Horn clauses to hereditary Harrop formulae.

The argument of this section is that such extensions to the logic are not just the dream of mathematically inclined theoreticians, but are a genuinely felt need of practical Prolog programmers. This need has not usually been expressed by an explicit call for such logical extensions. It *is* frequently expressed by the use of non-declarative programming hacks, which on closer inspection turn out to express non-Horn clause or even non-first order logical concepts.

5 Writing Non-Procedural Problem Descriptions

This need to extend the logic beyond Horn clauses becomes even more acute if one takes seriously the logic programming vision of freeing users from procedural considerations when describing their problems, *ie* to borrow the terminology of formal methods, if one tries to write logical specifications rather than logic programs.

Currently, logic programmers have to bear in mind how their specifications will be executed as programs. In Prolog, for instance, they are firstly constrained by the fact that the specification must be in extended Horn clauses and secondly constrained by the behaviour of the Prolog interpreter. Bearing these constraints in mind, a procedure, $union/3$, to form the union of two finite sets, might be 'specified' as:

$$\begin{aligned} union([\,], L, L). \\ union([Hd|Tl], L, Res) \quad &:- \quad member(Hd, L),\ union(Tl, L, Res). \\ union([Hd|Tl], L, [Hd|Res]) \quad &:- \quad not\ member(Hd, L),\ union(Tl, L, Res). \end{aligned} \tag{3}$$

In writing this the programmer has been restricted in a number of ways.

[7] $= ../2$ can be defined within predicate logic by a similar trick.

- The procedure has had to be expressed as a ternary predicate instead of a binary function. A binary function corresponds more closely to the way someone familiar with set theory might think about the procedure.
- A representational commitment has had to be made to represent the sets as lists.
- An algorithmic commitment has had to be made, namely that the procedure will work by standard list recursion on the first argument.
- A commitment has had to made to the mode of use, namely $union(+,+,-)$[8].

It is possible to write specifications without these restrictions. In the case of the union procedure this might done by giving the crucial property of a function $\cup/2$ as:

$$\forall S_1,\ \forall S_2,\ \forall El\ (El \in S_1 \cup S_2 \quad \longleftrightarrow \quad El \in S_1 \vee El \in S_2) \tag{4}$$

which gives a non-Horn clause when put in clausal form. It also uses a function to name a procedure.

If one applies the Lloyd-Topor process to translate this specification into extended Horn clauses then one gets the Prolog program:

$$\begin{aligned} union(S_1,S_2,S_3) &:- \quad not\ ion(S_1,S_2,S_3). \\ ion(S_1,S_2,S_3) &:- \quad E \in S_3,\ not\ E \in S_1,\ not\ E \in S_2. \\ ion(S_1,S_2,S_3) &:- \quad E \in S_1,\ not\ E \in S_3. \\ ion(S_1,S_2,S_3) &:- \quad E \in S_2,\ not\ E \in S_3. \end{aligned}$$

Together with a standard definition of $\in/2$ this will execute correctly provided $union/3$ is called with ground arguments. However, it will flounder if any of the arguments are non-ground.

6 Non-Executable and Inefficient Specifications

Of course, the original specification of $\cup/2$, (4) above, is not executable by the Prolog interpreter. The NU-Prolog interpreter can interpret it by applying the Lloyd-Topor translation as shown above, but this only works for mode $union(+,+,+)$. I am not aware of any logic programming interpreter which can interpret it for the most useful mode of $union(+,+,-)$, or better. If an interpreter cannot be provided, which adds control to the logic of (4) to produce an algorithm, then the Kowalski vision of "Algorithm = Logic + Control" cannot be realised.

In general, the interpreter of a logic programming language is a theorem prover. A complete theorem prover for a logical theory applied to the set of formulae that totally and unambiguously specifies some procedure *will* be able to execute that procedure for any legal set of inputs. However, the result of this execution may be less than helpful; the algorithm simulated by the theorem prover may be very inefficient, or the execution may not even return an output.

For an example of inefficient execution consider the following specification of a sorting algorithm taken from [Kowalski 79b].

$$sorted(List, Result) \quad :- \quad permutation(List, Result),\ ordered(Result). \tag{5}$$

If this is executed by the Prolog interpreter with *List* instantiated to a particular list and *Result* to a variable, then *permutation* will generate permutations of *List* and pass these to *ordered* until an ordered permutation is found. This could entail $n!$ calls of *permutation* and *ordered* before

[8] $union(+,+,+)$ does not work unless the list elements are all in the same order.

an ordered permutation is found - a *very* inefficient sorting algorithm. An even worse example can be found by reversing the order of *permutation* and *ordered*. Then all possible ordered lists will be generated until one is found which is a permutation of *List*. This 'algorithm' may never terminate.

The execution may not even return an output if a classical theorem prover is used to interpret non-Horn clauses. The execution may then correspond to a pure existence proof, so that the theorem prover proves that a procedure has an output but does not produce it. Consider, for instance, the non-Horn clause:

$$a \in s \vee b \in s.$$

together with the goal clause:

$$:- X \in s.$$

This will succeed without producing a value for X.

I do not know of any theoretical results on this, but I would speculate that for any given specification there *is* a theorem prover which could simulate any algorithm which meets that specification. However, such a result would not guarantee the existence of a theorem prover which effectively and efficiently executed *any* specification. To illustrate the practical difficulties one faces in designing such theorem provers consider what would be involved in simulating the mergesort algorithm from the specification of *sorted*/2 in (5) above. Kowalski shows [Kowalski 79b] how an intelligent theorem prover could co-routine between the subgoals of *permutation* and *ordered* to simulate a sensible sorting algorithm similar to selection sort. NU-Prolog can implement this co-routining. However, selection sort is not the most efficient sorting algorithm known and it is by no means clear how more efficient algorithms, *eg* mergesort, could be realised merely by changing the interpreter.

A further illustration of these practical difficulties can be found in [McCarthy 82]. McCarthy compares various efficient algorithms for map colouring with the best that has been achieved so far with intelligent meta-interpreters for Prolog. He finds a huge gulf between them.

What is suggested by consideration of these examples is that efficient execution of logical specifications entails that the specifications be first refined into a more executable form before being interpreted as procedures. One might adapt Kowalski's slogan to "Algorithm = Refined(Logic) + Control" to reflect this. A classic example of this approach is the transformation of the specification of the *sorted*/2 procedure, (5) above, into the various standard sorting algorithms, see *eg* [Darlington 78]. This problem of refining logical specifications into more executable form has a longer history in the field of formal methods than it has in logic programming. In the rest of this paper we consider what lessons logic programming might draw from the experience of formal methods.

7 A Technique for Program Synthesis

In my research group we have been experimenting with the Nuprl program development system, [Constable *et al* 86,Bundy *et al* 88]. In Nuprl, programs are synthesised from their specifications by proving a theorem of the form:

$$\forall Inputs,\ \exists Output.\ spec(Inputs, Output)\ .$$

where $spec(Inputs, Outputs)$ is a relationship between the inputs and the output of the desired program. For instance, the specification of $\cup/2$ could be written as[9]:

$$\forall A{:}u,\ \exists ASets{:}u,\ \forall S_1{:}ASets,\ \forall S_2{:}ASets,\ \exists S_3{:}ASets,\ \forall El{:}A$$
$$(El \in S_3 \longleftrightarrow El \in S_1 \vee El \in S_2)$$

[9] In order to make these examples intelligible to an audience unfamiliar with Nuprl, I have taken some liberties with the notation, using standard logical notation while preserving the spirit of Nuprl.

where $X:T$ means X is an object of type T, A is some type of objects, $ASets$ is the type of finite sets of such objects and u is the type of all simple types[10]. This theorem is proved constructively and the resulting proof is analysed to extract the implicit algorithm it defines for calculating the required output given any combination of inputs. A constructive proof is required to avoid the possibility of a pure existence proof in which the existence of an output is proved without any implicit algorithm being defined. Nuprl provides an interactive proof editor, which allows the user to guide the process of proof construction.

7.1 The Nuprl System

Nuprl is based on Martin-Löf Intuitionistic Type Theory, [Martin-Löf 79]. Not only does this provide a constructive logic, as required, but it greatly simplifies the task of extracting the program from the proof. This is because every rule of inference of the logic has an associated rule of program construction, so that the program is constructed as the proof progresses. This program is also in the Martin-Löf logic, and can, therefore, be interpreted as a higher order, typed, logic/functional program. Because the logic is typed, the type of each variable, in the example specification of union above, has to be declared in the variable's quantification. Nuprl uses these type declarations to do synthesis time type checking, rather than run time or compile time type checking. The typed logic also forces a restriction of the application of the union function to sets of objects of type A. We will indicate this by putting a subscript on the function name, *ie* $\cup_A$.

The procedural commitments that a Prolog programmer is forced to make within the specification are made by the Nuprl user during the course of the proof[11]. For instance, the choice of the data-structure to represent sets is made during the proof of the existence of $ASets$ given A. This decision is represented below by the instantiation of $ASets$ to $sets(A)$, where $sets$ is a function which takes any type and returns the type of finite sets of objects of that type. $sets(A)$ might, for instance, be defined as the type of equivalence classes of lists of objects of type A, where two lists are in the same equivalence class if and only if they have the same members. A decision to define $\cup_A/2$ by recursion on the structure of S_1 is made by a decision to prove the theorem by induction on S_1. In general, inductive proofs give rise to recursive programs: the form of the induction determining the form of recursion and, hence, the efficiency of the program. We are, therefore, particularly interested in proofs by induction and in the type of induction used.

7.2 An Example of Program Synthesis

Before the decision to prove the theorem by induction the state of the proof might be:

$$
\begin{array}{l}
a:u \\
s_1:sets(a) \\
s_2:sets(a) \\
\qquad \vdash_\Theta \quad \exists S_3:sets(a),\ \forall El:a(El \in S_3 \longleftrightarrow El \in s_1 \vee El \in s_2)
\end{array}
$$

where the program developed so far is:

$$S_1 \cup_A S_2 \;=\; \Theta$$

and Θ is the program to be generated by the rest of the proof.

After the application of induction the state of the proof might be:

$$a:u$$

[10] u is not itself a simple type. If it were we would fall foul of Russell's paradox.

[11] Except for the commitment to the mode which is made during the specification here too.

9

$sets(a) = \underbrace{a\ list}_{lists(a)} // \ l_1, l_2.\ \forall x:a$ quotient

$member(x, l_1) \leftrightarrow member(x, l_2)$

$$s_2 : sets(a)$$
$$\vdash_{\Phi} \quad \exists S_3 : sets(a),\ \forall El : a(El \in S_3 \longleftrightarrow El \in \emptyset \vee El \in s_2)$$

$$a : u$$
$$el' : a$$
$$s_1 : sets(a)$$
$$s_2 : sets(a)$$
$$\exists S_3 : sets(a),\ \forall El : a(El \in S_3 \longleftrightarrow El \in s_1 \vee El \in s_2)$$
$$\vdash_{\Psi} \quad \exists S_3 : sets(a),\ \forall El : a(El : S_3 \longleftrightarrow El \in el' \circ s_1 \vee El \in s_2)$$

where $el' \circ s_1$ is the set formed by adding a new member el' to the set s_1. The new state of the program is:

$$\begin{aligned} \emptyset \cup_A S_2 &= \Phi \\ (El' \circ S_1) \cup_A S_2 &= \Psi \end{aligned}$$

where Φ and Ψ are the programs to be generated by the base and step cases of the proof, respectively.

The final program might be:

$$\begin{aligned} & \emptyset \cup_A S_2 = S_2 \\ El' \in S_2 \quad &\longrightarrow \quad (El' \circ S_1) \cup_A S_2 = S_1 \cup_A S_2 \\ \neg El' \in S_2 \quad &\longrightarrow \quad (El' \circ S_1) \cup_A S_2 = El' \circ (S_1 \cup_A S_2) \end{aligned}$$

The program produced by Nuprl is certainly a program in logic. Whether it is a logic program depends on whether the restriction to first order clauses is regarded as a defining feature of logic programs. In many cases the Nuprl program is readily translated into Prolog. For instance, it is fairly easy to see how to translate sets of first order, conditional equations, like those above, into Prolog, *cf* the recursive definition of *union*/3 in (3) above.

7.3 The Properties of Program Synthesis

What are the properties of the Nuprl technique of program synthesis and how do these compare with the other techniques of program refinement that have been used within logic programming, *eg* [Clark & Sickel 77,Hogger 81]?

- The input to Nuprl is a logical specification. This specification does not need to be executable and, hence, is not restricted to a subset of logic, *eg* Horn clauses.
- Consequently, a Nuprl specification can specify a procedure either as a function or as a relation, as appropriate.
- Another consequence is that a Nuprl specification need not include a commitment to how data is to be represented.
- Nuprl synthesis ceases when the synthesis proof is finished, whereas some forms of logic program transformation are open ended and require user guidance to determine when the transformation should stop.

- Any output of Nuprl synthesis is guaranteed to be an executable program which meets the input specification, but there are no assurances as to how efficient that program is.

These properties suggest that synthesis is most useful for extracting an executable procedure from a non-executable specification. An efficient program might be produced either by further refinement of the program output by synthesis or by careful guidance of the synthesis process so that it produces an efficient program in the first place. To refine an inefficient program into a more efficient one, the techniques of program transformation developed by Darlington, Clarke and Hogger might be used.

Alternatively, Goad suggests the transformation of synthesis proofs rather than procedures, [Goad 80]. Synthesis proofs provide a richer environment for transformation than procedures since the procedural decisions are represented explicitly in the synthesis proof and do not need to be unpicked from the procedure before they can be remade. In our group Madden, [Madden 88], has been implementing this idea within the Nuprl system. In his proposed system the user would first use Nuprl to develop a simple synthesis proof and hence produce a simple and probably inefficient program. S/he would then use Madden's system to transform the synthesis proof into one producing a more efficient program. For instance, induction steps resulting in inefficient recursions would be replaced with induction steps producing more efficient recursions. Simultaneously we are thinking about how Nuprl can be guided to use more efficient induction steps in the first place, and hence to output an efficient program not requiring further refinement.

8 Guiding the Search of Inductive Proofs

Our research with Nuprl has concentrated on improving the computer's contribution to the guidance of the proof, so that the user receives the maximum assistance with the construction of the program. Not only do we want the user to avoid the combinatorial explosion inherent in any non-trivial theorem proving, but we also want to help the user make choices which will result in efficient programs. Since recursion plays such a dominant role in logic programs we are particularly interested in which inductive rule of inference is chosen and applied to the problem. The best work to date on the guidance of inductive proofs is that by Boyer and Moore, [Boyer & Moore 79]. Hence, we have been adapting the techniques embedded in the Boyer-Moore theorem prover to the Nuprl environment, [Stevens 88].

We are representing this search control knowledge using a logical formalism, which we call a *meta-logic*. The universe of the meta-logic consists of logic programs and specifications, and of heuristic strategies and tactics for generating synthesis proofs. The meta-logic is used to specify these strategies and tactics so that program synthesis and plan formation can be used to construct *proof plans* for guiding the theorem proving.

These proof plans should have the following properties:

- **Usefulness**: The plan should control the search for a proof.
- **Expectancy**: The use of the plan should carry some expectation of success.
- **Uncertainty**: On the other hand, success cannot be guaranteed. (A proof plan which was always successful would amount to a decision procedure, and the problem of program synthesis is, in general, undecidable. We do not want to limit proof plans to a collection of decision procedures.)
- **Patchability**: If the plan should fail it should be possible to patch it by providing alternative steps as replacements for the failing ones.

If our representation of proof plans can capture the correct balance between 'expectancy' and 'uncertainty', then we can provide a tool for predicting whether the synthesis proof will succeed in a reasonable time. Necessarily, such a tool would not be perfect, but it would give a reasonable prediction, and it could be used to localise the possible causes of failure.

Following LCF, [Gordon *et al* 79], a *tactic* is a meta-level procedure for generating a small part of the proof, e.g. unfolding a recursively defined function, [Burstall & Darlington 77] or applying mathematical induction. Tactics work by applying some object-level rules of inference to the current state of the proof. Tactics can be put together to form *strategies*, which might generate a large part of a proof – or the whole of it.

Note that some tactics may fail, causing failure of the whole strategy. For instance, a tactic for unfolding recursive definitions might fail if there were no recursive definition available which matched the current expression. Consider trying to unfold the expression:

$$even(s(a))$$

given only the recursive definition:

$$\begin{array}{rcl} even(0) & & \\ \neg even(s(0)) & & \\ even(s(s(X))) & \longleftrightarrow & even(X) \end{array}$$

A proof method is a meta-level specification of a proof tactic or strategy[12] consisting of a set of slots with values. It contains a *preconditions* and an *effects* slot. Both contain descriptions in the meta-logic: the preconditions' value describes the expression that the tactic applies to, and the effects' value describes the expression that the tactic produces if it succeeds. Other slots contain: the name of the method, the declaration of the variable types, descriptions of the formula input to and the formula output from the tactic, and a program for the tactic itself. An example of the method for unfolding is given in table 1.

Name	$unfold$
Declarations	$\forall Exp{:}exprs,\ \forall F{:}funcs,\ \forall X{:}vars,$ $\forall B{:}terms,\ \forall Z{:}vars, \forall S{:}constr.$
Input	$Exp[F(S(X))]$
Output	$Exp[B(X,F(X))]$
Preconditions	$F(S(Z)) = B(Z,F(Z))$
Effects	nil
Tactic	<a program that applies recursive rewrite rules>

Table 1: Method for the Unfold Tactic

A proof plan is a meta-level specification of a proof strategy, *ie* it is a kind of super-method. It is so constructed that the precondition[13] of each of its sub-methods are either implied by its own precondition or by the effects of earlier sub-methods. Similarly, its effect is implied by the effects of its sub-methods. The original conjecture should satisfy the precondition of the plan; the effect

[12] For simplicity we will use the word 'tactic' to include both 'tactic' and 'strategy' below.

[13] The precondition of a method is satisfied if the input to the tactic matches the Input slot and the Preconditions slot is true under this matching. Similarly, the effect of a method is realised if the output of the tactic matches the Output slot and the Effects slot is true under this matching.

of the plan should imply that the conjecture has been proved. Executing a proof plan consists of running each of its tactics according to the program it specifies. A proof plan can either be hand coded by the system builder, or the techniques of automatic program synthesis and plan formation can be used to construct it.

This representation meets the requirements given above, point by point, as follows:

- **Usefulness**: As the tactics run they will each perform a part of the object-level proof.
- **Expectancy**: If the conjecture meets the preconditions of the plan and each tactic succeeds then the effects of the plan will be true and the conjecture will be proved.
- **Uncertainty**: However, a tactic may fail, causing failure of the plan.
- **Patchability**: Since the preconditions and effects of a failing tactic are known, program synthesis and plan formation techniques may be (re)used to patch the gap in the plan with a subplan.

9 Conclusion

In this paper we have argued that the wonderful vision of logic programming, summed up in the slogan: "Algorithm = Logic + Control", can only be realised if the "Logic" in the slogan is interpreted in a much broader way than has been customary. In specifying a logic program the programmer must not be restricted to first order Horn clauses, but must be free to use quantification, disjunction, negation, variable predicates and functions, set extension, *etc*, as required. Otherwise, it will not be possible to free the programmer from the need to think in procedural terms.

Unfortunately, broadening the logic in this way brings a cost. It becomes possible to write program specifications that cannot be executed with standard logic programming interpreters; or that cannot be executed efficiently; or that do not return an output when executed. We need to extend the power of the interpreters to refine the logic before controlling its execution. The slogan is modified to: "Algorithm = Refined(Logic) + Control". Such logic refining interpreters can be borrowed from work in formal methods in software engineering. In particular, the technique of program synthesis, as embodied in systems like Nuprl, deserves close attention. It enables the programmer to go from a non-executable[14] specification to an executable one. It separates the choosing of an algorithm and of the data-structures from the specifying of the program.

These procedural choices are made during the course of the synthesis proof. In Nuprl most of the guidance of this proof must come from the user. To make the process a practical one for novice users it is necessary to provide more automatic guidance. We are currently exploring the potential of proof plans as a technique for providing this guidance. Each proof plan generalises the structure of previous successful proofs, and enables the flexible application of this structure to new proofs. A set of such proof plans can summarise the collected programming experience of many users over many years and put them at the disposal of the inexperienced logic programmer.

References

[Boyer & Moore 79] R.S. Boyer and J.S. Moore. *A Computational Logic.* Academic Press, 1979. ACM monograph series.

[14] By the built-in Nuprl interpreter.

[Bundy *et al* 88] A. Bundy *et al.* Proving properties of logic programs: a progress report. In L. Clarke, editor, *Proceedings of the 1988 Alvey Technical Conference*, The Alvey Directorate, 1988.

[Burstall & Darlington 77] R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, 1977.

[Clark & Sickel 77] K.L. Clark and S. Sickel. Predicate Logic: a calculus for deriving programs. In R. Reddy, editor, *Proceedings of IJCAI-77*, pages 419–420, IJCAI, 1977.

[Constable *et al* 86] R.L. Constable, S.F. Allen, H.M. Bromley, *et al. Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall, 1986.

[Darlington 78] J. Darlington. A synthesis of several sorting algorithms. *Acta Informatica*, 11:1–30, 1978.

[Goad 80] C.A Goad. Proofs as descriptions of computation. In W. Bibel and R. Kowalski, editors, *Proc. of the Fifth International Conference on Automated Deduction*, pages 39–52, Springer Verlag, Les Arcs, France, July 1980. Lecture Notes in Computer Science No. 87.

[Gordon *et al* 79] M.J. Gordon, A.J. Milner, and C.P. Wadsworth. *Edinburgh LCF - A mechanised logic of computation.* Volume 78 of *Lecture Notes in Computer Science*, Springer Verlag, 1979.

[Hogger 81] C.J. Hogger. Derivation of logic programs. *JACM*, 28(2):372–392, April 1981.

[Kowalski 79a] R. Kowalski. Algorithm = Logic +Control. *Communications of ACM*, 22:424–436, 1979.

[Kowalski 79b] R. Kowalski. *Logic for Problem Solving. Artificial Intelligence Series*, North Holland, 1979.

[Lloyd & Topor 84] J.W. Lloyd and R.W. Topor. Making Prolog more expressive. *J. Logic Programming*, 1(3):225–240, 1984.

[Lloyd 87] J.W. Lloyd. *Foundations of Logic Programs. Symbolic Computation*, Springer-Verlag, 1987. Second, extended edition.

[Madden 88] P. Madden. Automatic program optimization via the transformation of Nuprl synthesis proofs. In L. Clarke, editor, *Proceedings of the 1988 Alvey Technical Conference*, The Alvey Directorate, 1988.

[Martin-Löf 79] Per Martin-Löf. Constructive mathematics and computer programming. In *6th International Congress for Logic, Methodology and Philosophy of Science*, pages 153–175, Hannover, August 1979. Published by North Holland, Amsterdam. 1982.

[McCarthy 82] J. McCarthy. *Coloring maps and the Kowalski doctrine.* Report STAN-CS-82-903, Stanford University, 1982.

[Miller & Nadathur 88] D. Miller and G. Nadathur. An overview of λProlog. In *Proceedings of the Fifth International Logic Programming Conference/ Fifth Symposium on Logic Programming*, MIT Press, 1988.

[Naish 86] L. Naish. *Negation and control in Prolog.* Volume 238 of *Lecture notes in Computer Science*, Springer Verlag, 1986.

[Stevens 88] A. Stevens. *A Rational Reconstruction of Boyer-Moore Recursion Analysis.* Research Paper 360, Dept. of Artificial Intelligence, Edinburgh, 1988. Also in Proceedings of ECAI-88.

[Tarnlund 77] S. A. Tarnlund. Horn clause computability. *BIT*, 17(2):215–226, 1977.

[Warren 81] D.H.D. Warren. Higher-order extensions to Prolog - are they needed? In *Tenth International Machine Intelligence Workshop, Cleveland, Ohio*, 1981.