# The Synthesis of Logic Programs from Inductive Proofs *

Alan Bundy Alan Smaill Geraint Wiggins

Department of Artificial Intelligence,
University of Edinburgh.

**Abstract**

We describe a technique for synthesising logic (Prolog) programs from non-executable specifications. This technique is adapted from one for synthesising functional programs as total functions. Logic programs, on the other hand, define predicates. They can be run in different input modes, they sometimes produce multiple outputs and sometimes none. They may not terminate. The key idea of the adaptation is that a predicate is a total function in the all-ground mode, *i.e.* when all its arguments are inputs ($pred(+, \ldots, +)$ in Prolog notation). The program is synthesised as a function in this mode and then run in other modes. To make the technique work it is necessary to synthesise pure logic programs, without the closed world assumption, and then compile these into Prolog programs. The technique has been tested on the OYSTER (functional) program development system.

## 1 Introduction

The ideal aspired to in logic programming is allow computer users to describe their problem in the language of predicate logic. A clever interpreter will then run their logical description as a computer program and this will solve the user's original problem. Hence computer users will be freed from the necessity of thinking of their problems in procedural terms.

Current logic programming languages do not realise this ideal, [Bundy 88a]. The logical specification of a computer program may fail to be executable either efficiently or at all. For instance, it may not be in clausal form. The Lloyd-Topor translation process ([Lloyd 87], p113) will put it in clausal form, but the presence of negations in the clause body may cause it to flounder when executed using negation as failure. It may contain non-constructor functions, which need to be turned into predicates before they can be executed. The algorithm produced by a direct execution of the specification may be hopelessly inefficient.

*The research reported in this paper was supported by Esprit BRA grant 3012, and an SERC Senior Fellowship to the first author. We are grateful for feedback from Frank van Harmelen, David Basin and an anonymous referee on earlier drafts. Seán Matthews helped us defeat TEX. This paper first appeared in "Computational Logic" ed Lloyd, J.W., Springer-Verlag, 1990.

One answer to these difficulties is to transform the original specification into an equivalent logical formula which can be executed efficiently. A variety of such transformation techniques have been proposed, *e.g.* [Hogger 81, Bruynooghe *et al.* 89]. In this paper we discuss how to adapt a technique for synthesising functional programs from logical specifications to the synthesis of logic programs. This technique is based on the 'proofs as programs' paradigm, and is implemented, for instance, in the Nuprl program development system, [Constable *et al.* 86]. It is useful to consider the 'proofs as programs' technique for two main reasons.

- It is a powerful technique, which is useful for synthesis, transformation and verification. Adapting it to logic programming might well reveal extensions to current techniques. For instance, since it is based on higher order, typed logics, it will suggest how to adapt logic program transformation to such logics. It relates proof structure to program structure and, hence, program efficiency, thus providing a logical account of computational complexity, which can be used to guide program transformation.

- It is has a well developed theoretical foundation. Relating the 'proofs as programs' technique to existing logic programming transformation techniques might help us to understand them better. In particular, it relates recursive programs and inductive proofs in an intimate way. This may help us understand 'loop spotting' techniques, such as those in [Bruynooghe *et al.* 89], as a form of inductive proof.

The adaption of the 'proofs as programs' technique is not straightforward. It is only able to synthesise programs that are *total functions*, that is programs that are defined for all inputs of the right type and are guaranteed to terminate and to return precisely one output. On the other hand, the declarative meaning of a logic program procedure is a *predicate*. These predicates may be called as procedures in a variety of different input modes. For some combinations of input they fail and return no output, for others they return more than one output on backtracking. They are not guaranteed to terminate. The challenge is to adapt the 'proofs as programs' technique to synthesise these completely different kinds of mathematical objects.

# 2 The Proofs as Programs Technique

We begin by describing the proofs as programs technique. We have implemented this in the OYSTER system, [Horn 88], which is a re-implementation of Nuprl in Prolog. OYSTER and Nuprl are interactive theorem provers for a logic based on Martin-Löf Intuitionistic Type Theory, [Martin-Löf 79], a higher-order, richly typed logic. Using OYSTER, programs are synthesised from their specifications by proving a *specification 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. 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. OYSTER provides an interactive proof editor, which allows the user to guide the process of proof construction.

For instance, the specification of set union could be written as[1]:

$$\forall A{:}\mathcal{U},\ \exists ASets{:}\mathcal{U},\ \forall S_1{:}ASets,\ \forall S_2{:}ASets,\ \exists S_3{:}ASets,\\ \forall El{:}A(El \in S_3 \leftrightarrow El \in S_1 \vee El \in S_2)$$

where $X{:}\tau$ means $X$ is an object of type $\tau$, $A$ is some type of objects, $ASets$ is the type of finite sets of such objects and $\mathcal{U}$ is the type of all simple types[2].

OYSTER's Type Theory is an especially suitable logic for the task of program synthesis because it not only provides a constructive logic, as required, but it greatly simplifies the task of extracting the program from the proof. Every rule of inference of the logic has an associated rule of program construction, so that the program is constructed as the proof progresses. Thus there is a duality between proof steps and program steps, for instance applications of mathematical induction in the proof create recursion in the program. The synthesised program is also in OYSTER's logic, and can, therefore, be interpreted as a higher-order, typed, functional program. This program is called the *extract term.*

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. OYSTER uses these type declarations to do synthesis time type checking, rather than run time or compile time type checking. The function synthesised is only guaranteed to meet the specification when applied to sets of objects of type $A$. We will indicate this by putting a subscript on the function name, *i.e.* $\cup_A$.

# 3 An Example of Program Synthesis

The process of program synthesis via theorem proving is illustrated by the following example, using the specification of set union given above, (2).

After some elimination of quantifiers the state of the proof might be:

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

where the extract term constructed so far is $\lambda a, \lambda s_1, \lambda s_2.\Theta$, where $\Theta$ is the extract term to be constructed from the remainder of the proof. This suggests a program definition of:

$$S_1 \cup_A S_2 \quad = \quad \Theta$$

Suppose we now decide to apply induction on $s_1$. This will produce subgoals corresponding to the base case and the step case of the induction.

$$a{:}\mathcal{U}$$

[1]In order to make these examples intelligible to an audience unfamiliar with intuitionistic type theory we have used standard logical notation rather than that used in OYSTER. We follow the Prolog convention that identifiers starting with capital letters are variables.

[2]$\mathcal{U}$ is not itself a simple type. If it were we would fall foul of Russell's paradox.

$$
\begin{array}{l}
s_2{:}sets(a) \\
\quad \vdash_\Phi \quad \exists S_3{:}sets(a),\ \forall El{:}a(El \in S_3 \leftrightarrow El \in \emptyset \vee El \in s_2)
\end{array}
$$

$$
\begin{array}{l}
a{:}\mathcal{U} \\
el'{:}a \\
s_1{:}sets(a) \\
s_2{:}sets(a) \\
\exists S_3{:}sets(a),\ \forall El{:}a(El \in S_3 \leftrightarrow El \in s_1 \vee El \in s_2) \\
\quad \vdash_\Psi \quad \exists S_3{:}sets(a),\ \forall El{:}a(El\ inst S_3 \leftrightarrow El \in el' \circ s_1 \vee El \in s_2)
\end{array}
$$

where $el' \circ s_1$ is formed by adding a new member $el'$ to the set $s_1$. $el' \circ s_1$ is a set if $el'$ is not already a member of $s_1$. The new state of the program definition is:

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

where $\Phi$ and $\Psi$ are the extract terms of the base and step cases of the proof, respectively.

This is enough of the proof to give the flavour of the synthesis technique. The proof of the base and step cases will now proceed and will instantiate $\Phi$ and $\Psi$ . The final program might be:

$$
\begin{array}{rrcl}
 & \emptyset \cup_A S_2 & = & S_2 \\
El' \in S_2 \ \rightarrow & (El' \circ S_1) \cup_A S_2 & = & S_1 \cup_A S_2 \\
\neg El' \in S_2 \ \rightarrow & (El' \circ S_1) \cup_A S_2 & = & El' \circ (S_1 \cup_A S_2)
\end{array}
$$

# 4 Synthesising Logic Programs

How can this technique be adapted to the synthesis of logic programs, *e.g.* programs in Prolog?

Firstly, Prolog is neither higher-order nor typed, so we need to prevent the occurrence of these features in the synthesised programs. This is easily achieved by using a first-order logic in the place of OYSTER's current logic, or by restricting OYSTER's logic to its first order part. We still require this logic to be constructive and to associate program construction rules with each rule of inference. It will also be convenient to use a typed logic during synthesis, and then drop any reference to types in the final program. Otherwise, in order to ensure that a Prolog procedure is defined for all its arguments, we will have to provide clauses to deal with arguments that lie outside the intended types, *e.g.* clauses for *append*/3 for non-list arguments.

Secondly, OYSTER will produce total functions, whereas we require partial, multi-valued and, sometimes, non-terminating predicates. The 'proofs as programs' technique has been extended to synthesise partial and non-terminating functions, *e.g.* by restricting the domain of the function to sub-domains where it is both defined and terminating. The technique has not been extended to multi-valued functions. It is this problem we address in this paper.

# 5 Compiling Functions

An obvious solution is to synthesise a first-order, total function, then compile it into Prolog. The drawback of this solution is that it produces a Prolog program that is under-defined for some of its arguments and hence in some modes. Suppose we synthesise a function $foo{:}\tau_1 \times \ldots \times \tau_n \mapsto \tau$. This compiles into a Prolog predicate, $foo/n+1$, of type $\tau_1 \times \ldots \times \tau_n \times \tau$. This definition will not necessarily be exhaustive on its last argument, so it may not be defined for modes other than $foo(?, \ldots, ?, -)$.

Consider, for instance, a function $double{:}nat \mapsto nat$, defined as:

$$\begin{array}{rcl} double(0) & = & 0 \\ double(s(M)) & = & s(s(double(M))) \end{array}$$

This will become the Prolog procedure:

$$\begin{array}{rcl} double(0,0). & & \\ double(s(M), s(s(N))) & :- & double(M,N) \end{array}$$

This will be fine in mode $double(?, -)$, but not in mode $double(?, +)$. Consider, for instance, the call $double(M, 3)$[3]. This will fail. Maybe this is what was intended, but one cannot be sure of this since the full range of inputs for the second argument was not considered during the synthesis of the procedure. If *double* were originally defined as a predicate then all the cases would have to have been considered, maybe coming up with a definition like:

$$\begin{array}{rcl} double(0,0) & \leftrightarrow & true \\ double(s(M),0) & \leftrightarrow & false \\ double(0,s(0)) & \leftrightarrow & true \\ double(s(M),s(0)) & \leftrightarrow & false \\ double(0,s(s(N))) & \leftrightarrow & false \\ double(s(M),s(s(N))) & \leftrightarrow & double(M,N) \end{array} \tag{1}$$

which will find the integer half in mode $double(-, +)$, *e.g.* the call $double(M, 3)$ will instantiate $M$ to 1. Note case (1). By giving this the body *true* we ensure a non-*false* value in mode $double(-, +)$, even when the second argument is an odd number. If we want it to fail in such cases we should give (1) the body *false*. This definition compiles to the Prolog procedure:

$$\begin{array}{rcl} double(0,0). & & \\ double(0,s(0)). & & \\ double(s(M), s(s(N))) & :- & double(M,N) \end{array}$$

# 6 Predicates as Functions

We explore an alternative solution to this problem of synthesising multi-mode programs. The proofs as programs technique is adapted to synthesise predicates instead of functions,

[3]Where 3 is shorthand for $s(s(s(0)))$.

so that the resulting Prolog procedure is defined for all its arguments, and hence all input modes. The key idea of the solution is that in the all-ground mode input mode *i.e.* $pred(+, \ldots, +)$) logic programs *are* total functions. Hence, OYSTER and similar systems can be used directly to synthesise logic programs in this mode. If they are called in another mode then they will not be total functions, but this will not matter.

Suppose $foo/n$ is a Prolog procedure. We will consider $foo$ as an n-ary predicate of first-order typed logic. Let the type of the $i^{th}$ argument of $foo$ be $\tau_i$. $foo$ can also be regarded as a function of type $\tau_1 \times \ldots \times \tau_n \mapsto boole$. Observe that, in the all-ground mode, $foo(+, \ldots, +)$, $foo/n$ is a function. If all its arguments are ground then it must take either the value $true$ or $false$. It cannot take both values and it cannot take neither. If $foo$ is also terminating then it will be a *total* function.

For example, consider the procedure $del/3$, which deletes one occurrence of a particular element from a list. Its standard Prolog definition is:

$$del(X, [X|Tl], Tl). \tag{2}$$

$$del(X, [Hd|Tl], [Hd|L']) \quad :- \quad del(X, Tl, L'). \tag{3}$$

with the intended mode $del(+, +, -)$. This is an archetypal example of a partial and multi-valued procedure. The call $del(a, [b, c], L)$ fails, so no value is found for $L$. The call $del(a, [a, b, a], L)$ succeeds twice, first with the value $[b, a]$ for $L$ and second with the value $[a, b]$.

However, in mode $del(+, +, +)$, $del/3$ *is* a function. For instance, $del(a, [b, c], [b, c])$ has value $false$; $del(a, [a, b, a], [b, a])$ has value $true$ and $del(a, [a, b, a], [a, b])$ has value $true$.

# 7 Pure Logic Programs

This observation is obscured in Prolog because of the heavy use of the closed world assumption. There are no explicit truth values. Falsity is equated with failure; truth with success. In order to use our program synthesis techniques we must use a slightly different notation for logic program procedures in which truth and falsity are explicit. We will adapt and extend the notation first introduced in [Bundy 88b]. Procedures identified with predicates and are defined by a set of formulae, which we will call *tracts*[4]. There is precisely one tract for each combination of arguments to the procedure. A *pure logic program* is a set of predicates each of which is defined by a set of tracts.

A *tract* has the form:

$$Condition \quad \rightarrow \quad (Head \quad \leftrightarrow \quad Body)$$

where $Condition$ may be $true$, in which case it is omitted. $Head$ is of the form $Pred(Arg_1, \ldots, Arg_n)$, where $Pred$ is an n-ary predicate. Each argument of $Pred$ is either a recursive argument or a parameter argument. If $i$ is a parameter argument then $Arg_i$ must be a variable. If $i$ is a recursive argument then $Arg_i$ must be a constructor term. A constructor term is

[4]These are what we called *cases* in [Bundy 88b]. Unfortunately, this word is already in use to describe the parts of proofs, *e.g.* step case. We cannot use *clauses* because this word is reserved for describing Prolog programs, which we will want to distinguish from tracts. The Student's English Dictionary describes a 'tract' as a "a short dissertation in which some particular subject is treated", and this seems fairly close to the meaning we intend.

either a variable or a constructor function applied to constructor terms. All the variables in *Head* are distinct. *Condition* and *Body* can be any first-order formulae that do not contain non-constructor functions.

The tracts defining a predicate are mutually exclusive and exhaustive, *i.e.* they give precisely one value for each combination of arguments. This is best illustrated by an example. Consider the predicate $del{:}\tau \times list(\tau) \times list(\tau) \mapsto boole$. Its tracts might be:

$$\begin{aligned} del(X,[],L) &\leftrightarrow false \qquad (4) \\ del(X,[Hd|Tl],L) &\leftrightarrow (X = Hd \wedge Tl = L) \vee \qquad (5) \\ &\quad\;\; (\exists L'{:}list(\tau)\ L = [Hd|L'] \wedge del(X,Tl,L')) \end{aligned}$$

These tracts are exhaustive since the second argument of *del* is of type $list(\tau)$. $del/3$ will take the value *true* whenever the right hand side of tract (5) evaluates to *true*, *e.g.* for $del(a,[a,b,c],[b,c])$.

Compare these tracts with the Prolog clauses (2) and (3), above. Case (4) defines *del* when the recursive argument is an empty list. There is no Prolog clause corresponding to this. Case (5) corresponds to clauses (2) and (3): each disjunct to the body of one clause. Note that $Tl$ and $L$ are *not* identified in the head of the tract (5), as they are in clause (2), rather they are set equal in the corresponding disjunct in the body.

These tracts define a total function when called in mode $del(+,+,+)$. What happens for other modes, *e.g.* $del(+,+,-)$? There is bound to be at least one tract matching each calling pattern, but there may be more than one. This means that *del* will always return a result, but it may return more than one. Sometimes this result will be $false$ and sometimes *true*. In addition, it will instantiate some of the variables in the arguments either partially or totally. In those cases where *del* returns *true*, we can regard these instantiations as the answers sought. This will give us the partiality and multi-valuedness that we seek. In general, we will call the truth values returned by a predicate call the *results*. When the result is *true* we will call the instantiations of any unbound variables the *outputs*.

Predicates may return the same result by several different computation routes. If there are an infinite number of computation routes then the predicate is non-terminating. If only a finite number of these routes return the result $false$ and an infinite number return the result *true*, then we will call the non-termination *benign*. This terminology reflects the observation that this kind of non-termination is often desired as a way of obtaining an infinite set of outputs. If all the infinite number of routes return the result $false$ then we will call the non-termination *malignant*, since this kind of non-termination is rarely desired. Otherwise, there will be some *true* routes and an infinite number of $false$ ones. We will call this kind of non-termination *pre-cancerous*, since it is possible[5] to obtain some outputs before the call turns malignant. If there are an infinite number of *true* routes then it is possible to put off the malignancy indefinitely.

For instance, consider the predicate $is\_nat/1$, which in mode $is\_nat(+)$ tests whether its argument is a natural number.

$$\begin{aligned} is\_nat(0) &\leftrightarrow true \\ is\_nat(s(N)) &\leftrightarrow is\_nat(N) \end{aligned}$$

[5]With a clever interpreter.

In mode $is\_nat(+)$ this predicate is total, *i.e.* single valued and terminating. In mode $is\_nat(-)$ it is benignly non-terminating, producing the infinite set of outputs: $0, s(0), s(s(0)), \ldots$.

Now consider the predicates $is\_list/1$ and $both/1$ defined by:

$$\begin{array}{rcl} is\_list([]) & \leftrightarrow & true \\ is\_list([H|T]) & \leftrightarrow & is\_list(T) \end{array}$$

$$both(X) \leftrightarrow is\_nat(X) \wedge is\_list(X)$$

$both/1$ terminates in mode $both(+)$ but is malignantly non-terminating in mode $both(-)$

# 8 Compiling Pure Logic Programs

§11 defines a compiler for transforming pure logic programs into Prolog programs. Cases whose body is $false$ are omitted. $\leftrightarrow$s are turned into $:-$s. Existential quantifiers in the body are dropped. Conditions are put at the front of bodies and tracts are put into clausal form. Body literals of the form $Var = Term$ are omitted and all occurrences of $Var$ replaced by $Term$. Applying this compiler to the pure logic definition of $del$ above produces the Prolog program given in clauses (2) and (3) above, as required.

This procedure is guaranteed to be total in the mode $del(+,+,+)$. However, it would be little use if it could only be called in that mode. How will it behave in other modes? In general, if an argument marked + is used in mode - then the procedure may be over-defined, *i.e.* return multiple results. This may cause it to be multi-valued and/or non-terminating.

The efficiency of a Prolog procedure is mode dependent. In particular, the efficiency may be dependent on the order of the literals in each clause. This problem may be best dealt with during the compilation into Prolog, rather than during synthesis. This is because order is irrelevant for pure logic procedures. It only becomes significant for the literals and clauses of Prolog procedures. The intended mode of use should be an input to the compilation phase (see §11) and influence its outcome.

# 9 An Example of Predicate Synthesis

We can adapt the proofs as programs technique to the synthesis of pure logic procedures as follows. We prove theorems of the form:

$$\forall X_1{:}\tau_1, \ldots, X_n{:}\tau_n, \exists B{:}boole.\ spec(X_1, \ldots, X_n) \leftrightarrow B \tag{6}$$

where $spec(X_1, \ldots, X_n)$ specifies an n-ary predicate, $foo$, in a constructive, first order logic. From a proof of this theorem we can extract a definition of $foo/n$ as a pure logic program.

For instance, to specify the $del/3$ program we might prove the theorem:

$$\begin{array}{l} \forall \tau{:}\mathcal{U}, \forall X{:}\tau, \forall K, L{:}list(\tau), \exists B{:}boole. \\ (\exists L_1, L_2{:}list(\tau).\qquad L = L_1 <> L_2 \quad \wedge \quad K = L_1 <> [X|L_2]) \quad \leftrightarrow \quad B \end{array}$$

where $<>$ is the infix list append function.

After eliminating the initial quantifiers and applying list induction to $K$, this theorem reduces to the base and step cases:

$$\begin{array}{l} \tau{:}\mathcal{U}, x{:}\tau, l{:}list(\tau) \\ \quad \vdash_{\Phi} \quad \exists B{:}boole.\ (\exists L_1, L_2{:}list(\tau).l = L_1 <> L_2 \wedge [] = L_1 <> [x|L_2]) \leftrightarrow B \end{array}$$

$$\begin{array}{l} \tau{:}\mathcal{U}, x{:}\tau, l{:}list(\tau) \\ hd{:}\tau, tl{:}list(\tau) \\ \forall L{:}list(\tau), \exists B{:}boole. \\ \qquad (\exists L_1, L_2{:}list(\tau).\ L = L_1 <> L_2 \ \wedge\ tl = L_1 <> [x|L_2]) \ \leftrightarrow\ B \\ \quad \vdash_{\Psi} \quad \exists B{:}boole. \\ \qquad\qquad (\exists L_1, L_2{:}list(\tau).\ l = L_1 <> L_2 \ \wedge\ [hd|tl] = L_1 <> [x|L_2]) \ \leftrightarrow\ B \end{array}$$

This proof suggests the following partial definition of $del/3$.

$$\begin{aligned} del(X, [], L) &\ \leftrightarrow\ \Phi \\ del(X, [Hd|Tl], L) &\ \leftrightarrow\ \Psi \end{aligned}$$

and $\Phi$ and $\Psi$ are the extract terms of the base and step cases, respectively.

Since for no $L_1$ and $L_2$ is it the case that $[] = L_1 <> [x|L_2]$ then the base case reduces to $\exists B{:}boole.false \leftrightarrow B$. This is proved by instantiating $B$ to $false$, which suggests an instantiation of the extract term, $\Phi$, to $false$. This completes the base case.

The step case requires a proof by cases using:

$$L_1 = [] \quad \vee \quad \exists Hd_1{:}\tau, Tl_1{:}list(\tau).L_1 = [Hd_1|Tl_1]$$

In the first case the induction conclusion reduces to:

$$\exists B{:}boole. \quad (\exists L_2{:}list(\tau). \quad l = L_2 \quad \wedge \quad [hd|tl] = [x|L_2]) \quad \leftrightarrow \quad B$$

using the rewrite rule $[] <> L \Rightarrow L$. Note that this and the other rewrite rules used in the proof are based on equalities or equivalences and hence valid in both directions. This is necessary for soundness since they are applied under $\leftrightarrow$.

Hence, by instantiating $L_2$ to $l$, and applying the substitution axiom to the induction conclusion reduces it to:

$$\exists B{:}boole. \quad (hd = x \quad \wedge \quad tl = l) \quad \leftrightarrow \quad B$$

This is proved by splitting it into four sub-cases using: $hd = x \vee hd \neq x$ and $tl = l \vee tl \neq l$. In three of these sub-cases $hd = x \wedge tl = l$ reduces to $false$ and the other sub-case it reduces to $true$. $B$ is instantiated accordingly, to complete this first case. This suggests $hd = x \wedge tl = l$ as the extract term of this first case.

In the second case the induction conclusion reduces to

$$\begin{array}{l} \exists B{:}boole. \\ (\exists Hd_1{:}\tau, Tl_1, L_2{:}list(\tau). \\ \qquad l = [Hd_1|Tl_1 <> L_2] \quad \wedge \quad [hd|tl] = [Hd_1|Tl_1 <> [x|L_2]]) \\ \qquad\qquad\qquad\qquad \leftrightarrow \quad B \end{array}$$

We split this into two sub-cases using:

$$l = [] \quad \vee \quad \exists H'{:}\tau, L'{:}list(\tau).l = [H'|L']$$

Since $[] = [Hd_1|Tl_1 <> L_2]$ is $false$, the first sub-case is readily proved with $B$ instantiated to $false$. The second sub-case reduces to:

$$\exists B{:}boole.\ (\exists Tl_1, L_2{:}list(\tau).\ L' = Tl_1 <> L_2 \ \wedge\ hd = Hd_1 \ \wedge\ tl = Tl_1 <> [x|L_2]) \ \leftrightarrow\ B$$

by applying the substitution axiom. We then apply a further case split using: $hd = Hd_1 \vee hd \neq Hd_1$. The second sub-sub-case is readily proved with $B$ instantiated to $false$. The first sub-sub-case reduces to:

$$\exists B{:}boole.\ (\exists Tl_1, L_2{:}list(\tau).\ L' = Tl_1 <> L_2 \ \wedge\ tl = Tl_1 <> [x|L_2]) \ \leftrightarrow\ B$$

This matches the induction hypothesis by instantiating $L$ to $L'$ and renaming the bound variable $L_1$ to $Tl_1$. We can then use the induction hypothesis to prove the induction conclusion and complete this sub-sub-case. The use of the induction hypothesis suggests the recursive call $del(x, tl, L')$ as the extract term of this sub-sub-case and $\exists L'{:}list(\tau).\ l = [hd|L'] \wedge del(x, tl, L')$ as the extract term of the whole second case.

This completes the whole step case, and suggests for it the extract term:

$$(hd = x \wedge l = tl) \quad \vee \quad (\exists L'{:}list(\tau).\ l = [hd|L'] \wedge del(x, tl, L')$$

Hence the final pure logic program suggested is:

$$\begin{array}{rcl} del(X, [], L) & \leftrightarrow & false \\ del(X, [Hd|Tl], L) & \leftrightarrow & (X = Hd \wedge Tl = L) \vee \\ & & (\exists L'{:}list(\tau).\ L = [Hd|L'] \wedge del(X, Tl, L')) \end{array}$$

as required.

## 10 Implementation in OYSTER

We have tested this logic program synthesis process in the OYSTER system. This is not an ideal vehicle for two reasons.

- The programs extracted are functions in the type theory and require transformation into pure logic programs. In general, it is not possible to make this transformation, because ...
- ... OYSTER's logic is higher order. Thus it is possible to extract non-first-order programs that cannot be transformed into pure logic programs — even from first-order specifications.

However, the test was easily performed, since it required no modifications to OYSTER, and provided supportive evidence for the proposals advanced in this paper. Proofs were obtained by interactive use of OYSTER, during which higher order features in either the specification or extract terms were avoided. The extract term was translated into a pure logic program by hand.

In OYSTER's type theory, propositions are identified with the types of their proofs, so a proposition is true if and only if it is inhabited when considered as a type. This suggests a definition of *boole* as $\mathcal{U}$, the type of simple types. However, this permits specification theorems of the form:

$$\forall X_1{:}\tau_1, \ldots, X_n{:}\tau_n, \exists B{:}boole.\ spec(X_1, \ldots, X_n) \leftrightarrow B$$

to be proved in a trivial way by instantiating $B$ to $spec(X_1, \ldots, X_n)$. This synthesies a predicate $foo/n$ whose definition is equal to the original specification, *i.e.*

$$foo(X_1, \ldots, X_n) \quad \leftrightarrow \quad spec(X_1, \ldots, X_n)$$

which is not what we want.

Instead, we defined *boole* as a type containing precisely two members: $true$ and $false$, where $false$ is the empty type *void* and *true* is some simple, inhabited type. Note that equality on this type is decidable, so that the law of excluded middle holds for this type, *i.e.*

$$\forall B{:}boole. \quad B = true \quad \vee \quad B = false$$

Since OYSTER's logic is constructive, this law does not hold in general, in particular, it does not hold for the type $\mathcal{U}$.

This prevents the instantiation of $B$ by $spec(X_1, \ldots, X_n)$, unless $spec(X_1, \ldots, X_n)$ has been reduced to a member of *boole*. In fact, we can only prove the specification theorem by exhibiting a mapping from a proof of $B$ to a proof of $spec(X_1, \ldots, X_n)$, and vice versa. The necessary and sufficient condition for the existence of these mappings is that $spec(X_1, \ldots, X_n)$ be decidable. The role of this is to prove $spec(X_1, \ldots, X_n)$ by exhibiting a decision procedure, namely the synthesised program.

Note that if a formulae is decidable then the law of excluded middle holds for it and we can use this law to make a case split. We used this facility extensively in the proof in §9, *e.g.* $hd = x \vee hd \neq x$ was used in case 1 of the step case. It was not possible to split on $spec(X_1, \ldots, X_n) \vee \neg spec(X_1, \ldots, X_n)$ until it had been shown decidable. Our synthesis proof proceeded by reducing $spec(X_1, \ldots, X_n)$ to a collection of decidable subgoals, using case splits to instantaite $B$ and complete the proof of each subgoals, and then reconstituting their decision predicates into the desired logic program.

OYSTER has been used in this way to prove the example theorem in the last section. The resulting extract term is a function in the type theory. In this case the extract term had a clear correspondence to the desired pure logic program for $del/3$. Unfortunately, it is not possible, in general, to translate such extract terms into pure logic programs. To avoid this problem we are working on a version of OYSTER based on a constructive, first order logic in which the extract terms are pure logic programs.

# 11 A Prolog Compiler

Once we have completely synthesised a pure logic procedure then we need to translate it into Prolog clauses. This can be done by the following compiler, defined by a series of rewritings.

This compiler may translate two logically equivalent pure logic procedures into two logically equivalent[6], but procedurely non-equivalent, Prolog procedures. This is because the compilation process introduces impure features like cut and negation as failure. These are interpreted procedurally and their interpretation can depend on the order of literals and clauses. The compiler described below preserves the original order of expression nesting and tracts as much as possible, and this will determine the behaviour of the target Prolog procedure. Alternatively, one could use information about the intended mode to influence the order of clauses and literals. We see no way to avoid this problem of procedural non-equivalence as long as the target language contains impure features.

Here is the compiler.

1. First we write each tract into the form of a *program statement*, as defined by [Lloyd 87], p107, *i.e.* a formula of the form $A \leftarrow W$, where $A$ is an atom and $W$ is a first order formula.

   Unconditional and Conditional tracts are rewritten from the form:

   $$\begin{array}{rcrcl} & & Head & \leftrightarrow & Body \\ & & & or & \\ Condition & \rightarrow & Head & \leftrightarrow & Body \end{array}$$

   to the form:

   $$\begin{array}{rcl} Head & \leftarrow & Body \\ & or & \\ Head & \leftarrow & Condition \wedge Body \end{array}$$

   Note that these forms are procedurally, but not logically equivalent. Their procedural equivalence relies on the closed world assumption in Prolog. If it was known that *Head* was only ever to be called in fully ground mode then a cut could be safely inserted between the *Condition* and the *Body* of conditional statements, but this will not be safe in other modes.

   For instance, under this transformation our $del/3$ procedure (tracts 4 and 5 above) becomes:

   $$\begin{array}{rcl} del(X,[],L) & \leftarrow & false \\ del(X,[Hd|Tl],L) & \leftarrow & (X = Hd \wedge Tl = L) \vee \\ & & (\exists L' \; L = [Hd|L'] \wedge del(X,Tl,L')) \end{array}$$

2. Next we apply the Lloyd-Topor translation process to turn each program statement into a set of clauses. Essentially, this puts the program statement bodies into clausal form, but with some modifications to minimise the number of negated literals. Quantifiers are eliminated, negations are moved inwards and disjunctions cause a statement to split into two.

[6] In as much as a Prolog procedure may be said to have a logical meaning.

For instance, our $del/3$ procedure becomes:

$$\begin{array}{rcl} del(X,[],L) & \leftarrow & false \\ del(X,[Hd|Tl],L) & \leftarrow & X = Hd \wedge Tl = L \\ del(X,[Hd|Tl],L) & \leftarrow & L = [Hd|L'] \wedge del(X,Tl,L') \end{array}$$

3. Clauses can now be tidied up by removing equalities between variables and terms. This step is optional since it affects neither the meaning nor the behaviour of the clause.

   Any literal in a clause body of the form $Variable = Term$ or $Term = Variable$ is omitted and each occurrence of $Variable$ in the clause is replaced by $Term$.

   For instance, our $del/3$ procedure becomes:

$$\begin{array}{rcl} del(X,[],L) & \leftarrow & false \\ del(Hd,[Hd|Tl],Tl) & \leftarrow & true \\ del(X,[Hd|Tl],[Hd|L']) & \leftarrow & del(X,Tl,L') \end{array}$$

4. Each clause whose body contains the literal $false$ is deleted. Each literal $true$ is omitted from its clause body.

   For instance, our $del/3$ procedure becomes:

$$\begin{array}{rcl} del(Hd,[Hd|Tl],Tl) & & \\ del(X,[Hd|Tl],[Hd|L']) & \leftarrow & del(X,Tl,L') \end{array}$$

5. Finally, we rewrite the clauses into standard Prolog notation. Each clause of the form:

$$Head \ \leftarrow \ Body_1 \ \wedge \ \ldots \ \wedge \ Body_n$$

   is rewritten to the form:

$$Head \ :- \ Body_1, \ \ldots, \ Body_n.$$

   and each body literal of the form $\neg Atom$ is replaced by $not\ Atom$.

   For instance, our $del/3$ procedure becomes:

$$\begin{array}{rcl} del(Hd,[Hd|Tl],Tl). & & \\ del(X,[Hd|Tl],[Hd|L']) & :- & del(X,Tl,L'). \end{array}$$

# 12 Conclusion

We have shown apply the 'proofs as programs' synthesis technique to logic programs. The key idea is to regard predicates as functions onto truth values in the 'all-ground' input mode, *i.e.* when all arguments are instantiated to ground terms. The synthesised procedure can then be run in other modes. In these other modes it will be defined for all inputs, but may give multiple outputs, and/or it may not terminate. That is, there will be at least one tract that matches any procedure call, but there may be many in non-all-ground modes and there may be an infinite number of computation routes.

Instead of synthesising Prolog programs directly, we synthesise procedures in a 'pure logic' form and then compile them into Prolog as a post-processing step. This is necessary in order to regard logic programs as functions onto truth values — these truth values are only implicit as 'success' or 'failure' in Prolog programs, and must be made explicit for the technique to work. This prevents us from reasoning with Prolog programs directly. It also prevents us from dealing with the non-declarative features of Prolog.

We have successfully tested this 'proofs as programs' technique on the synthesis of $del/3$ using the OYSTER system. In order to synthesise a first order program, it was necessary to avoid the use of any higher-order features of the OYSTER logic. To obviate this need to avoid higher-order features, we plan to build a synthesis system especially geared to the synthesis of pure logic programs. The first-order, Deductive Tableau System of [Manna & Waldinger 87] might be a better role model for this than Nuprl or OYSTER. However, it would also be interesting to use OYSTER, or a similar higher-order system, to explore the synthesis of higher-order logic programs.

In principle, it ought to be possible to use the 'proofs as programs' technique for the synthesis, transformation and verification of logic programs. We have not conducted large scale testing of our technique, so we do not yet have the empirical evidence to assess this potential. We plan to do this.

We also plan to compare our technique with other approaches to the transformation of logic programs. In particular, we want to compare it to systems that synthesise (or transform into) recursive programs, *e.g.* [Bruynooghe *et al.* 89]. In our technique this necessarily requires proof by mathematical induction. We suspect that something like induction is also present in these other techniques, although it is sometimes disguised as 'loop spotting' in a symbolic execution tree.

# References

[Bruynooghe *et al.* 89] M. Bruynooghe, D. De Schreye, and B. Krekels. Compiling control. *Journal of Logic Programming*, pages 135–162, 1989.

[Bundy 88a] Alan Bundy. A broader interpretation of logic in logic programming. In R. Bowen, K. & Kowalski, editor, *Proceedings of the Fifth International Logic Programming Conference/ Fifth Symposium on Logic Programming*, pages 1624–1648. MIT Press, 1988. Also available from Edinburgh as DAI Research Paper No. 388.

[Bundy 88b] Alan Bundy. Proposal for a recursive techniques editor for prolog. Research Paper 394, Dept. of Artificial Intelligence, University of Edinburgh, 1988. In the special issue of Instructional Science on Learning Prolog: Tools and Related Issues.

[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.

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

[Horn 88] C. Horn. The Nurprl proof development system. Working paper 214, Dept. of Artificial Intelligence, University of Edinburgh, 1988. The Edinburgh version of Nurprl has been renamed Oyster.

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

[Manna & Waldinger 87] Z. Manna and R.J. Waldinger. The origin of a binary-search paradigm. *Science of Computer Programming*, 9:37–83, 1987.

[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, Hanover, August 1979. Published by North Holland, Amsterdam. 1982.