Synthesis and Transformation of Logic Programs by Constructive, Inductive Proof

Alan Bundy, Jane Hesketh,
Ina Kraan and Geraint Wiggins

April 30, 1998

Published in
Proceedings of LoPSTr-91,
Manchester,
July 1991

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

Synthesis and Transformation of Logic Programs from Constructive, Inductive Proof

Geraint Wiggins
geraint@ed.ac.uk

Alan Bundy
bundy@ed.ac.uk

Ina Kraan
inak@ai.ed.ac.uk

Jane Hesketh
jane@ai.ed.ac.uk

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

Abstract

We discuss a technique which allows synthesis of logic programs in the "proofs-as-programs" paradigm [Constable 82]. Constructive, inductive proof is used to show that the specification of a program is realisable; elaboration of a proof gives rise to the synthesis of a program which realises it. We present an update on earlier ideas, and give examples of and justification for them. The work is presented as foundation for further work in *proof planning*, where we aim to synthesise not only programs, but *good* programs.

1 Introduction

In this paper, we present further developments in work on a method for the synthesis of logic programs originally presented in [Bundy *et al.* 90a]. The method uses the constructive, inductive proof of conjectures which specify the desired programs' input/output behaviour, coupled with simultaneous automatic extraction of the computational content of the proofs. The method has been implemented as a user-directed proof development system which provides a rich environment for the elaboration of synthesis proofs. The implementation, the *Whelk* system, has been designed to be amenable to application of existing ideas and further developments in the technique of *proof planning*, in which inductive proofs are developed automatically [Bundy 88, Bundy *et al.* 91, Wiggins 90].

In Section 2 we present a brief background summary of the ideas of [Bundy *et al.* 90a]. In Section 3, we discuss issues arising from the attempt to synthesise logic programs in our chosen paradigm and style. Section 4 gives an example of program synthesis and transformation within our technique, and Section 5 summarises the ideas presented and suggests some directions for future work.

2 The Basic Technique

Whelk is based on the "proofs-as-programs" paradigm of [Constable 82]. In order

to adapt this approach, intended for the synthesis of *functional* programs, to that of *logic* programs, we view logic programs, in the "all ground" mode, as functions onto a valued Boolean-valued type. We can then in principle perform synthesis of programs by the constructive proof of conjectures of the following general form (though we explain below why this form is not precisely ideal for our purposes):

$$\vdash \forall i{:}\vec{\tau}.\exists b{:}boole.spec(\vec{i}) \leftrightarrow b$$

We will call a conjecture which specifies the behaviour of a program to be synthesised a *specification conjecture*. The conjecture states that, for all vectors of input values ($\vec{i}$) of the correct type(s) ($\vec{\tau}$) [1], there is some Boolean value (b), such that $spec(\vec{i})$ is logically equivalent to that value. We define the Boolean type to contain only the constants $true$ and $false$, so it is not possible merely to instantiate b with $spec(\vec{i})$ to prove the conjecture. Therefore, one might think of the proof process as proving the decidability of the *spec*. See [Bundy *et al.* 90a] for more detail.

The approach differs from the conventional functional proofs-as-programs approach, and from some other attempts to adapt it to logic programming (*eg* [Fribourg 90]) in that it allows us to synthesise programs which are partial, many-valued relations (as logic programs often are) rather than (strict) functions.

Now, in order to prove a conjecture *constructively*, we must show not just that there *is* a value of b in *boole* for each possible combination of inputs, $\vec{i}$ — rather, we must show that we can *construct* that Boolean value. Showing that we *can* construct the value involves showing *how* to construct it. Therefore, the proof must contain (in some maybe abstruse form) instructions for such a construction.

In the event that $i{:}\vec{\tau}$ is empty, we can supply either $true$ or $false$ as the witness for our Boolean variable b, simply by introduction on the existential — this then becomes the body of our synthesised program. If $i{:}\vec{\tau}$ is not empty (so there are universally quantified, typed variables in our specification conjecture) then the Boolean witness may depend on the values of those variables. Our proof will therefore consist of nested case-splits[2], dividing the types of the variables into sub-types for which a single Boolean value may be computed. These case splits, conjoined with the values of the various b's associated with them, will constitute the main body of the synthesised program.

3 *Whelk*: The Current Implementation

3.1 *Whelk*

Whelk has been implemented within a proof development system originally designed for the Oyster system [Horn 88, Bundy *et al.* 90b]. The Martin-Löf Constructive Type Theory of Oyster has been replaced in *Whelk* by a much simpler typed first-order logic. This design decision has raised some questions about how the synthesis-proof approach actually works, which we discuss in this section.

Whelk provides a rich environment for the user guided development of proofs (synthesis proofs or otherwise). It includes a tactic language which allows the construction of more complicated compound rules from the atomic ones which define the proof system; this mechanism will allow the application of automatically planned proof steps in future work.

A fundamentally important feature in the program synthesis application is that each rule of inference in the proof system corresponds one-to-one with a rule of construction for a structure called the *extraction* from the proof. This correspondence is so arranged that the extraction from a proof of a specification conjecture (as

[1] *ie* a vector of input value/type pairs

[2] except in the trivial case where the synthesised program is always true or always false

above) constitutes a logic program which fulfils the *spec* for all input where *spec* is true, and which fails for all input where *spec* is false. This will become clearer in the examples below.

3.2 The Form of The Specification Conjecture

Before we can use *Whelk* for program synthesis, we must motivate our chosen form of specification conjecture. It is important to understand in advance that there are several options open, and that we do not suggest that any one is in any absolute sense better than any other. We feel, however, that the option we have chosen offers the best insight into the operation of the synthesis system from the point of view of the user.

3.2.1 Existential Quantifier and Logical Equivalence

To recapitulate: in [Bundy *et al.* 90a], we loosely outlined an approach where we proved a conjecture of the form

$$\vdash \forall i{:}\vec{\tau}.\exists b{:}boole.spec(\vec{i}) \leftrightarrow b$$

to show that, for all well-typed input there is some truth value (*true* or $false$, here) to which the specification $spec(\vec{i})$ is logically equivalent. Because our proof system is constructive, we have to show not only that some such value exists, but that we can generate it. This is equivalent to saying that $spec(\vec{i})$ is decidable. Also, the extraction may be thought of as a witness for b.

Closer inspection, however, shows that this approach is flawed. The logic of our proof system is a typed first-order logic. However, the existentially quantified variable b in the specification conjecture above is (assuming the usual interpretation of $\leftrightarrow$) a variable over formulae, and not over terms. Thus, the existential quantifier is second-order, which we do not want.

In fact, the original intention was that the right hand side of the equivalence should be notated in a different language from the left (or at least in a disjoint subset of the same language). By arranging the categories of the various expressions carefully, we could overcome the apparent syntactic ill-formation. This, indeed, will be the approach we follow later in this discussion, but with this form of synthesis conjecture, we suggest that the mixture of languages is inelegant and potentially confusing.

An alternative solution to the problem would be simply to allow the second-order quantifier *in this circumstance only* (first-order existentials frequently arise in the body of *spec*); however, this approach is unsatisfying because of its rather *ad hoc* flavour — the necessary restriction on quantifier order is hard to justify in the larger context of the proof system.

3.2.2 Meta-Variable and Logical Equivalence

A better version of essentially the same idea is to allow the proof system to support uninstantiated meta-variables, and to prove a conjecture of the form

$$\vdash \forall i{:}\vec{\tau}.spec(\vec{i}) \leftrightarrow \mathcal{M}$$

where $\mathcal{M}$ is a meta-variable over formulæ. In this approach, the construction of the extraction is much more explicit than in the others presented here, as it actually constitutes the (partially instantiated) value of the variable $\mathcal{M}$ as the proof proceeds. This technique, however, would involve significant changes to the operation of the Oyster/*Whelk* environment, and thus is not ideal for our purposes here.

Further, recent work by Ina Kraan on using the meta-variable approach to give a semantics for synthesis involving the form of conjecture we do use (see 3.2.5) suggests that though there are some difficult logical problems associated with its use as a semantics, it is in itself a potentially useful technique for program synthesis. One advantage over the *Whelk* system is that all the rules required are derivable directly from a standard Sequent Calculus, and thus are known *a priori* to be sound. We intend to pursue this approach in parallel with the *Whelk* system, in order to be able to compare the two.

This technique might be viewed as deductive synthesis, as, for example, in [Bibel & Hörnig 84].

3.2.3 Lifting with Functional Specification

Another meta-flavoured approach would be to change the form of our synthesis conjecture to

$$\vdash \forall i{:}\vec{\tau}.\exists b{:}boole.spec(\vec{i}) =_{boole} b$$

In this formula, we have lifted the original *spec* to the meta-level, and are actually reasoning about the term which names it under a bijective naming function. Thus, all of the connectives and predicates permissible in the logic must have a functional equivalent. For example, the "and" connective, $\wedge : formula \times formula \mapsto formula$, must correspond with a function $\wedge' : boole \times boole \mapsto boole$. This is to an extent inconvenient, because the system must maintain the distinction (in terms of the constructed program, $\wedge$ and $\wedge'$ are *significantly* different) — and it is in the nature of logic systems that such a distinction must be explicit in the inference rules. Thus, the user has to worry about a distinction which is largely irrelevant to the production of the proof (NB as opposed to the program) s/he must elaborate, which complicates the system unnecessarily. Oddly, it turns out to be undesirable to remove the source of this problem in the formal sense (see 3.2.4) — but we can nevertheless protect the user from its most irritating effects, as we explain in 3.2.5.

3.2.4 Decidability Proof

Perhaps the most obvious route to the construction of the program we need is a specification conjecture of the form

$$\vdash \forall i{:}\vec{\tau}.spec(\vec{i}) \vee \neg spec(\vec{i})$$

In this approach, we are showing that there is some truth value for the *spec*, in much the same way as in 3.2.1, and relying on the constructive nature of our proof system to give us an executable extraction. However, there is a significant difference here, because there is no existential quantifier, and no explicit truth value associated with the *spec* for any given value(s) of $\vec{i}$. In removing the truth value, we remove the basis of the problem arising in the sections above, which is clearly one related to self-reference; all the way along, we have been attempting to reason about the truth values of a logical expression in a(n object-level, first-order) system *within* that system, which is not straightforwardly possible.

Unfortunately, this apparent step forward brings with it a disadvantage. In the approach of 3.2.1, above, we were able in some degree to motivate our use of that form of synthesis conjecture by saying that the logic program we would eventually synthesise would be a witness for the existentially quantified Boolean variable, b. Now, though, we have thrown our variable away, and there is no equivalent intuitive foothold on which to base a semantics for our proof system. We also lose the useful notion that we are trying to show the existence of some (logical) output for all possible input, which is the fundamental tenet of the proofs-as-programs approach.

3.2.5 Existential Quantifier and Realisation Operator

On this basis, therefore, we must take a step back, and attempt to compromise on a system which is logically correct and transparent (like 3.2.4), but also easily motivated as an instance of our chosen paradigm (like 3.2.1, 3.2.2 and 3.2.3). Ideally, the system should not introduce needless complication for the user (as does 3.2.3), and it should definitely not mix meta- and object-levels in an unmotivated, and, indeed, syntactically ill-formed, way (as does 3.2.1).

The solution, then, is a compromise between user-friendliness, logical correctness, and intuitive transparency. It involves the addition of a new operator in our logic which will allow us correctly to mix formulæ and terms in the way suggested by 3.2.1. We will also restrict our type *boole* to contain only the values *true* and $false$, which will give us the effect of enforcing the proof of decidability, as in 3.2.4. The scope of our new operator also gives us a motivated way to separate the two different kinds of operator (the distinction between the connectives and the functions in 3.2.3) without bothering the user.

The operator we use to do all this is read as "realises", written

$$\hookrightarrow : formula \times term \mapsto formula$$

and its meaning is defined by:

$$\vdash formula \hookrightarrow b \qquad iff \qquad \begin{cases} \vdash formula \leftrightarrow (b =_{boole} true) \\ \vdash \neg formula \leftrightarrow (b =_{boole} false) \end{cases}$$

where $formula$ is an object level formula in our logic and b is in *boole*. $=_{boole}$ denotes equality in *boole*.

Since our specification formula is now within the scope of the realisation operator, we have a straightforward syntactic distinction between (*eg*) the $\wedge$ and $\wedge'$ of section 3.2.3, which makes little difference from the point of view of the user applying inference rules (indeed, it allows him/her to *ignore* the difference, which is desirable), but allows the proof system to detect unambiguously which way any given connective should be treated.

The semantics above gives us a connection between the formula of the *spec* itself and some Boolean value(s); we must now give a semantics for the extraction system which will supply the synthesised logic program required to produce these values for any given well-typed input.

3.3 Semantics of *Whelk* Extractions

3.3.1 The Languages $\mathcal{L}_\mathcal{E}$ and $\mathcal{L}_\mathcal{I}$

The semantics of our extraction system is fairly complicated, and will be reported in detail elsewhere. There follows enough of a sketch to allow the reader to understand the example in Section 4.

First, we have in the logic of our proof system the usual first-order connectives, $\neg$, $\rightarrow$, $\vee$, $\wedge$, $\forall$ and $\exists$, with the addition of $\leftarrow$ and $\leftrightarrow$ for convenience. We have the realisation operator $\hookrightarrow$, as above, and $\oplus$ (exclusive or) which will allow us to make certain optimisations as part of the proof process. We also have the operator : for sort/type membership, sorts, $boole = \{\ true\ false\ \}$, $nat = \{\ 0\ s(0)\ s(s(0))\ \ldots\ \}$, and the parametric type of lists: $\alpha\, list = \{\ []\ [h_0]\ [h_0, h_1]\ \ldots\ \}$ where $h_i{:}\alpha$. Finally, formulæ may consist of literals (*ie* predicates, often applied to argument terms), statements of equality within a type (written $=_\tau$), combinations of these made with the connectives, or contradiction (written $\{\}$). Terms may consist of literals (*ie* functions, often applied to argument terms). We will call this language the *external* logic, $\mathcal{L}_\mathcal{E}$, on the grounds that it is what the user sees as s/he elaborates proofs.

In order to motivate our extraction system, we also need an *internal* logic, $\mathcal{L}_\mathcal{I}$, which is mostly invisible to the user of *Whelk*. It is almost identical with $\mathcal{L}_\mathcal{E}$, and maps to it, through a function we call the *interpretation*. The elements of the $\mathcal{L}_\mathcal{I}$ correspond one-to-one with the syntactically identical elements of $\mathcal{L}_\mathcal{E}$, *except* in the following cases.

The Boolean terms *true* and $false$ of $\mathcal{L}_\mathcal{E}$ correspond with *predicates true* and $false$ respectively in $\mathcal{L}_\mathcal{I}$.

Only type/sort constructor functions are allowed in $\mathcal{L}_\mathcal{I}$; non-constructor functions have no correspondent in $\mathcal{L}_\mathcal{I}$.

The $\hookrightarrow$ operator has no correspondent in $\mathcal{L}_\mathcal{I}$.

We write $formula^*$ to denote the expression in $\mathcal{L}_\mathcal{I}$ which corresponds with $formula$ in $\mathcal{L}_\mathcal{E}$ under the interpretation.

3.3.2 Realisation Semantics

Recall first that the semantics of $\hookrightarrow$ is defined thus:

$$\vdash formula \hookrightarrow b \qquad \textit{iff} \qquad \begin{cases} \vdash formula \leftrightarrow (b =_{boole} true) \\ \vdash \neg formula \leftrightarrow (b =_{boole} false) \end{cases}$$

where $formula$ is a closed object level formula in $\mathcal{L}_\mathcal{E}$ and b is in *boole*.

Note that, while $formula$ may contain quantifiers, the expression as a whole is unquantified, which makes the semantics simpler — b is simply either *true* or $false$. If we add in universal quantification of the entire formula, to represent arguments to our synthesised predicate, we have a more complicated semantics for the realisation, which can be written thus, in terms of a composition of sub-proofs (the universally quantified input vector $\vec{x{:}\tau}$ has been introduced as a hypothesis):

$$\begin{array}{c} \vec{x{:}\tau}, condition_1 \vdash formula \hookrightarrow b_1 \\ \vdots \\ \vec{x{:}\tau}, condition_n. \vdash formula \hookrightarrow b_n \end{array}$$

where $condition_i$ is a (possibly empty) conjunction of conditions (either sets of equations – *ie* unifiers – or references to axiomatic predicates) free in (some of the) $\vec{x}$. It is important to note that the $condition_i$ must together select all the elements of the product type $\prod_t t \in \vec{\tau}$, so that the predicate is defined for all well-typed input. *Whelk* ensures this during construction of the proof.

The corresponding *pure logic procedure* (see definition in [Bundy *et al.* 90a]), constructed automatically as a side-effect of the elaboration in *Whelk* of the proof, is then defined thus:

$$extract(\vec{x{:}\tau}) \quad \longleftrightarrow \quad \bigvee_{1 \le i \le n} (condition_i{}^* \wedge b_i{}^*)$$

In fact, it turns out that some sub-terms of the $condition_i$ are irrelevant and/or tautologous, and will not appear in the extracted construction.

3.3.3 Example: `zero/1`

For an initial very simple example, let us consider the predicate `zero/1`, which is true if its argument is the natural number 0 and false otherwise. Our synthesis conjecture is:

$$\vdash \forall x{:}nat.\exists b{:}boole.x =_{nat} 0 \hookrightarrow b$$

The proof runs as follows. (Note that though this is a Gentzen Sequent Calculus derivation, it is presented backwards, in refinement style. Thus "introduction" steps actually make operators disappear.) First, we introduce the universal quantifier, to give:

$$x{:}nat$$
$$\vdash \exists b{:}boole.x =_{nat} 0 \hookrightarrow b$$

Then, we import a lemma about the decidability of equality in the naturals, giving

$$x{:}nat$$
$$\forall x{:}nat.\forall y{:}nat.x =_{nat} y \oplus \neg x =_{nat} y$$
$$\vdash \exists b{:}boole.x =_{nat} 0 \hookrightarrow b$$

We eliminate the universals in the hypothesis with the values x and 0, giving:

$$x{:}nat$$
$$\forall x{:}nat.\forall y{:}nat.x =_{nat} y \oplus \neg x =_{nat} y$$
$$x =_{nat} 0 \oplus \neg x =_{nat} 0$$
$$\vdash \exists b{:}boole.x =_{nat} 0 \hookrightarrow b$$

We can now eliminate disjunction, to give two subconjecture (in the context of the above hypotheses):

$x =_{nat} 0$	$\neg x =_{nat} 0$
$\vdash \exists b{:}boole.x =_{nat} 0 \hookrightarrow b$	$\vdash \exists b{:}boole.x =_{nat} 0 \hookrightarrow b$

Next, we proceed by supplying witnesses for the Boolean, b — *true* and $false$ respectively:

$x =_{nat} 0$	$\neg x =_{nat} 0$
$\vdash x =_{nat} 0 \hookrightarrow true$	$\vdash x =_{nat} 0 \hookrightarrow false$

The introduction rules for the $\hookrightarrow$ operator are defined so that we can make the following step:

$x =_{nat} 0$	$\neg x =_{nat} 0$
$\vdash x =_{nat} 0$	$\vdash \neg x =_{nat} 0$

And finally, both branches can now be terminated by tautology, since each subconjecture appears in its own hypotheses.

The *pure logic procedure* synthesised from this proof, and written in $\mathcal{L}_{\mathcal{I}}$, is as follows:

$$zero(x{:}nat) \longleftrightarrow (x =_{nat} 0 \wedge true) \vee (\neg x =_{nat} 0 \wedge false)$$

which corresponds with the semantic scheme given above in the following way. Labelling the left branch of the proof as 0, and the right branch as 1, we are looking for a b_0, a b_1, a $condition_0$ and a $condition_1$. The two b's were supplied by the existential introduction: *true* and $false$ respectively, in $\mathcal{L}_{\mathcal{E}}$, mapped to their predicative correspondents in $\mathcal{L}_{\mathcal{I}}$ by *. $condition_0$ is then $x =_{nat} 0$ and $condition_1$ is $\neg x =_{nat} 0$, which map to their syntactic identities under *.

The program may be trivially partially evaluated to give

$$zero(x{:}nat) \longleftrightarrow x =_{nat} 0$$

and converted (automatically, by *Whelk*) to the Gödel module (Natural `Zero` is the Gödel constant we have chosen, to be distinct from the integer 0):

```
MODULE Zero.

IMPORT Naturals.

PREDICATE Zero:  Natural.

Zero(x) <- x = Zero.
```

Various other construction steps are hidden within the proof; the only points of any major significance are that the initial universal introduction gives rise to the argument of the synthesised predicate, and that the lemma does not appear in the extraction. This latter is achieved because hypotheses (*ie* the $condition_i$ of the semantic scheme) are only built into the construction when they are actually used to show that a conjecture is an axiom.

3.3.4 Inductive Realisation Semantics

For any program synthesis system to have a reasonable coverage, it must be able to synthesise recursive programs — this applies even more to logic and functional program synthesis than to synthesis of other kinds of programs. In order to introduce recursion into programs synthesised by *Whelk*, we take advantages of the close relationship between recursion and induction. Indeed, recursion is necessarily intimately linked with induction throughout the "proofs-as-programs" literature (*eg* [Constable 82]); without this duality synthesis of recursive programs. In the Constructive Type Theory of the Oyster system, for example, each recursive data-type has its own explicit *induction term*.

In a constructive logic, inductive proofs are always of a form in which it is shown that, given an existing *construction*, one can *construct* a further value (*cf* classical logic, where pure existence proof is acceptable). Choice of an induction scheme in the proof corresponds with choice of (class of) algorithm in program construction — for example, given the usual specification of list sorting,

$$perm(X,Y), ord(Y)$$

one can derive either bubble sort (*via* structural induction on the input type) or quicksort (*via*, for example, a divide-and-conquer form of course-of-values induction). The choice of induction schemes to generate "good" programs is an interesting and difficult question, which will be a central topic of our future research, to be implemented in extensions of the existing CLaM proof planner [Bundy *et al.* 91]. This work will be linked with existing work at Edinburgh on transformation of functional synthesis proofs to give more efficient extracted programs [Madden 91].

The use of inductive proof to construct recursive programs requires a small extension to the semantics presented above. Until now, we have excluded the instantiation of Boolean variables by anything other than ground terms in *boole*. Now, in order to allow recursion, we must admit instantiation by a restricted class of Boolean-valued expressions — namely, atoms defined themselves during the proof as pure logic procedures. The general form of the pure logic procedure is now

$$extract_m(x_m{:}\vec{\tau}_m) \quad \longleftrightarrow \quad \bigvee_{1 \leq i \leq n_m} (condition_{\langle m,i \rangle}{}^* \wedge b_{\langle m,i \rangle}{}^*)$$

where $b_{\langle m,i \rangle} \in \{\ true \quad false \quad extract_1(y_1{:}\vec{\sigma}_1) \ldots extract_r(y_r{:}\vec{\sigma}_r)\ \}$, and $m{:}nat$, $r{:}nat$.

It is important to understand that this less restrictive régime is only admitted in the *internal* logic, $\mathcal{L}_{\mathcal{I}}$ — thus, the original idea of forcing a proof, in $\mathcal{L}_{\mathcal{E}}$, of decidability is preserved.

The introduction of recursive calls in the *Whelk* system is handled by the association of specific program components with hypotheses. In the case of the substitutions in the example above, the computational content associated with the hypothesis was simply the hypothesis itself; in the case of the induction hypothesis, that content is a recursive call to the predicate being defined, with the appropriate substitution of arguments. This will become clear in the example of Section 4.

3.4 "Real" Logic Programs

Given a pure logic procedure it is a near-trivial task to convert to languages such as Prolog and Gödel. One of the advantages of this technique is that information contained in the proof can help in detecting significant factors in the execution of the finished program — for example, in general, it is advisable to include a Gödel `DELAY` declaration for inductively constructed predicates, so they they are only unfolded when the induction variable is ground, thus helping prevent unbounded recursion. The elicitation of the information necessary to do this from a program can be difficult; from an inductive proof, it is often trivial.

4 An Example of Program Construction

Regrettably there is only space for one example here. However, synthesis of the `notmember`/2 predicate is a good example of how the *Whelk* system may be used for synthesis, or to produce partial evaluations of existing programs. `notmember`/2, predictably, succeeds if and only if its first argument, a natural number, is not a member of the list of naturals which constitutes its second argument. We use a pure logic procedure exactly equivalent to the conventional `member`/2 definition as a lemma, which motivates one of our case splits. Negation, as defined in the proof system, gives us the rest of the mechanism we need.

We start with the following synthesis conjecture. (In the proof below, for lack of space, we will omit repeated hypotheses and assume an incremental context unless otherwise stated.)

$$\vdash \forall x{:}nat.\forall y{:}nat\ list.\exists b{:}boole.\neg member(x,y) \hookrightarrow b$$

First, we introduce both the universal quantifiers:

$$\begin{array}{l} x{:}nat \\ y{:}nat\ list \\ \vdash \exists b{:}boole.\neg member(x,y) \hookrightarrow b \end{array}$$

and then apply primitive induction on y. The base case of the induction runs fairly simply as follows:

$$\vdash \exists b{:}boole.\neg member(x,[]) \hookrightarrow b$$

We introduce *true* on the Boolean existential:

$$\vdash \neg member(x,[]) \hookrightarrow true$$

We will need a lemma about membership of empty lists to prove this:

$$\begin{array}{l} \forall z{:}nat.\neg member(z,[]) \\ \vdash \neg member(x,[]) \hookrightarrow true \end{array}$$

and we must make the appropriate substitution in the lemma:

$$\begin{array}{l} \neg member(x,[]) \\ \vdash \neg member(x,[]) \hookrightarrow true \end{array}$$

We can now introduce $\hookrightarrow$, as in the earlier example, to give:

$$\begin{array}{l}\neg member(x, []) \\ \vdash \neg member(x, [])\end{array}$$

which is a tautology. We are now left with half a pure logic procedure, the ellipsis "..." being the part as yet unconstructed. The sub-procedure, $notmember_l$ is constructed as a result of the application of induction.

$$\begin{array}{l}notmember(x{:}nat, y{:}nat\ list) \longleftrightarrow \\ \qquad notmember_l(x{:}nat, y{:}nat\ list)\end{array}$$

$$\begin{array}{l}notmember_l(x{:}nat, y{:}nat\ list) \longleftrightarrow \\ \qquad y =_{nat\ list} [] \wedge true \wedge true \vee \\ \qquad \exists v1{:}nat\ list.\exists v0{:}nat.y =_{nat\ list} [v0 \mid v1] \wedge \ldots\end{array}$$

In terms of the semantic scheme, in subproof l of the proof, branch 0 gives us (in $\mathcal{L}_{\mathcal{I}}$)

$$\begin{array}{lll}condition_{\langle l,0\rangle} & \text{is} & y =_{nat\ list} [] \\ b_{\langle l,0\rangle} & \text{is} & true\end{array}$$

The step case of the proof proceeds as follows. Initially, we have two new variables, to define the non-empty list for the induction, and the induction hypothesis. Note that the extraction component associated with this hypothesis in *Whelk* is

$$notmember_l(x{:}nat, v1{:}nat\ list)$$

the witness for this assumption being, of course, the proof of the base case.

$$\begin{array}{l}v0{:}nat \\ v1{:}nat\ list \\ \exists b{:}boole.\neg member(x, v1) \hookrightarrow b \\ \vdash \exists b{:}boole.\neg member(x, [v0 \mid v1]) \hookrightarrow b\end{array}$$

The first thing we do is rewrite the *member* reference according to the definition lemma (which is the usual definition of `member/2`).

$$\begin{array}{l}\vdash \forall z{:}nat. \\ \qquad \forall v0{:}nat. \\ \qquad\qquad \forall v1{:}nat\ list. \\ \qquad\qquad\qquad member(z, [v0 \mid v1]) \leftrightarrow z =_{nat} v0 \vee member(z, v1)\end{array}$$

After the appropriate substitutions in the conjecture above, we have:

$$\vdash \exists b{:}boole.\neg(x =_{nat} v0 \vee member(x, v1)) \hookrightarrow b$$

As in the `zero/1` example, earlier, we now need to decide on the equality sub-term, so we introduce an appropriate lemma, and substitute inside it, giving:

$$\begin{array}{l}x =_{nat} v0 \oplus \neg x =_{nat} v0 \\ \vdash \exists b{:}boole.\neg(x =_{nat} v0 \vee member(x, v1)) \hookrightarrow b\end{array}$$

As before, we eliminate the $\oplus$, giving two branches of the proof, which we will consider separately here. First, the positive equality case:

$$\begin{array}{l}x =_{nat} v0 \\ \vdash \exists b{:}boole.\neg(x =_{nat} v0 \vee member(x, v1)) \hookrightarrow b\end{array}$$

The *formula* here is clearly false, because the left disjunct inside the negation is true by tautology, so we can introduce $false$ on the Boolean, to give:

$$x =_{nat} v0$$
$$\vdash \neg(x =_{nat} v0 \vee member(x, v1)) \hookrightarrow false$$

Introducing $\hookrightarrow$ then yields:

$$x =_{nat} v0$$
$$\vdash \neg\neg(x =_{nat} v0 \vee member(x, v1))$$

which is proven in the following steps, ending with tautology:

$$x =_{nat} v0$$
$$\neg(x =_{nat} v0 \vee member(x, v1))$$
$$\vdash \{\}$$

$$x =_{nat} v0$$
$$\vdash x =_{nat} v0 \vee member(x, v1)$$

$$x =_{nat} v0$$
$$\vdash x =_{nat} v0$$

Our pure logic procedure now looks like this:

$$notmember(x{:}nat, y{:}nat\ list) \longleftrightarrow$$
$$\quad notmember_l(x{:}nat, y{:}nat\ list)$$

$$notmember_l(x{:}nat, y{:}nat\ list) \longleftrightarrow$$
$$\quad y =_{nat\ list} [] \wedge true \wedge true \vee$$
$$\quad \exists v1{:}nat\ list.\exists v0{:}nat.$$
$$\quad\quad y =_{nat\ list} [v0 \mid v1] \wedge ((x =_{nat} v0 \wedge false) \vee \ldots)$$

so in our semantic scheme $condition_{\langle l,1\rangle}$ is $x =_{nat} v0$ and $b_{\langle l,1\rangle}$ is $false$.

Finally, we have the case of inequality between x and $v0$:

$$\neg x =_{nat} v0$$
$$\vdash \exists b{:}boole.\neg(x =_{nat} v0 \vee member(x, v1)) \hookrightarrow b$$

Rewriting this according to the familiar de Morgan Law (one of those which are constructively valid), and introducing on the disjunction inside the $\hookrightarrow$ gives us two branches, the first being trivially provable by introduction of *true* and tautology:

$$\neg x =_{nat} v0$$
$$\vdash \exists b{:}boole.\neg x =_{nat} v0 \wedge \neg member(x, v1) \hookrightarrow b$$

Left branch (branch 2):

$$\neg x =_{nat} v0$$
$$\vdash \exists b{:}boole.\neg x =_{nat} v0 \hookrightarrow b$$

$$\neg x =_{nat} v0$$
$$\vdash \neg x =_{nat} v0 \hookrightarrow true$$

$$\neg x =_{nat} v0$$
$$\vdash \neg x =_{nat} v0$$

The right branch (branch 3) is a copy of our induction hypothesis so we immediately have a tautology. Remember that the construction associated with this step is the recursive call.

$$\exists b{:}boole.\neg member(x, v1) \hookrightarrow b$$
$$\vdash \exists b{:}boole.\neg member(x, v1) \hookrightarrow b$$

Finally, then, we have our pure logic program:

$$notmember(x{:}nat, y{:}nat\ list) \longleftrightarrow$$
$$\quad notmember_l(x{:}nat, y{:}nat\ list)$$

$$notmember_l(x{:}nat, y{:}nat\ list) \longleftrightarrow$$
$$\quad y =_{nat\ list} [] \wedge true \wedge true \vee$$
$$\quad \exists v1{:}nat\ list.\exists v0{:}nat.$$
$$\quad\quad y =_{nat\ list} [v0 \mid v1] \wedge$$
$$\quad\quad\quad ((x =_{nat} v0 \wedge false) \vee$$
$$\quad\quad\quad\ (\neg x =_{nat} v0 \wedge notmember_l(x{:}nat, v1{:}nat\ list)))$$

as we would wish. Our semantic scheme has now been instantiated thus:

$condition_{\langle l,2\rangle}$	is	$y =_{nat\ list} [v0 \mid v1] \wedge x =_{nat} v0$
$b_{\langle l,2\rangle}$	is	$false$
$condition_{\langle l,3\rangle}$	is	$y =_{nat\ list} [v0 \mid v1] \wedge \neg x =_{nat} v0$
$b_{\langle l,3\rangle}$	is	$notmember_l(x{:}nat, v1{:}nat\ list)$

In Gödel, the program comes out (automatically) as:

```
MODULE Notmember.

IMPORT Lists.
IMPORT Numbers.

PREDICATE Notmember :  Number * List( Number ).

Notmember(x,y) <-
        Notmember_1(x,y).

PREDICATE Notmember_1 :  Number * List( Number ).

Notmember_1(x,y) <-
       y = [] \/
       Some [v1]
           Some [v0]
               ( y = [v0|v1] &
                  (~ x = v0 &
                      Notmember_1(x,v1))).
```

5 Conclusion and Future Work

In this paper, we have given an overview of the operation of the *Whelk* program synthesis system. We have outlined the semantics which will allow us to demonstrate that the programs synthesised by the system always fulfil the specification in the conjecture proven during the synthesis process. We suggest that the examples here show that the scope of our synthesis system is quite wide. In particular, it is not restricted to certain special classes of program (*eg* stratified, or locally stratified) as are many comparable synthesis/transformation systems.

Much work yet remains to be done. In particular, we have yet to use our semantics to verify all our inference rules; this verification will be carried out in parallel with development of a system using the meta-level approach mentioned in Section 3.2.2. We also wish to increase the choice of induction schemes available to the system, so that (*eg*) divide-and-conquer and course-of-values algorithms are available to the *Whelk* user.

All this, however, is just the starting point for research in automation of the construction of the synthesis proofs themselves. This work will begin forthwith, based on the existing success of the "rippling" paradigm for proof planning [Bundy *et al.* 90c]. Rippling allows us drastically to reduce the search for correct proofs of synthesis (and other) conjectures by characterising the symbolic behaviour of inductive proofs in a very precise way. The technique can reduce the search space by as many as 33 orders of magnitude (see [Bundy *et al.* 88] for more detail). All of the manipulations carried out in the examples in this paper may be motivated in terms of rippling, and thus the proofs may be planned and applied automatically. Also, we have not yet considered how to use the technique to produce *good* programs — this will be a central topic of work in the short term future.

References

[Bibel & Hörnig 84] W. Bibel and K. M. Hörnig. LOPS — a system based on a strategical approach to program synthesis. In A. Biermann, G. Guiho, and Y. Kodratoff, editors, *Automatic Program Construction Techniques*, pages 69–90. MacMillan, 1984.

[Bundy 88] Alan Bundy. The use of explicit plans to guide inductive proofs. In R. Lusk and R. Overbeek, editors, *9th Conference on Automated Deduction*, pages 111–120. Springer-Verlag, 1988. Longer version available from Edinburgh as DAI Research Paper No. 349.

[Bundy *et al.* 88] A. Bundy, F. van Harmelen, J. Hesketh, and A. Smaill. Experiments with proof plans for induction. Research Paper 413, Dept. of Artificial Intelligence, University of Edinburgh, 1988. Appeared in Journal of Automated Reasoning, 7, 1991.

[Bundy *et al.* 90a] A. Bundy, A. Smaill, and G. A. Wiggins. The synthesis of logic programs from inductive proofs. In J. Lloyd, editor, *Computational Logic*, pages 135–149. Springer-Verlag, 1990. Esprit Basic Research Series. Also available from Edinburgh as DAI Research Paper 501.

[Bundy *et al.* 90b] A. Bundy, F. van Harmelen, C. Horn, and A. Smaill. The Oyster-Clam system. In M. E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 647–648. Springer-Verlag, 1990. Lecture Notes in Artificial Intelligence No. 449. Also available from Edinburgh as DAI Research Paper 507.

[Bundy *et al.* 90c] A. Bundy, F. van Harmelen, A. Smaill, and A. Ireland. Extensions to the rippling-out tactic for guiding inductive proofs. In M. E. Stickel, editor, *10th International Conference on Automated Deduction*, pages 132–146. Springer-Verlag, 1990. Lecture Notes in Artificial Intelligence No. 449. Also available from Edinburgh as DAI Research Paper 459.

[Bundy *et al.* 91] Alan Bundy, Frank van Harmelen, Jane Hesketh, and Alan Smaill. Experiments with proof plans for induction. *Journal of Automated Reasoning*, 7:303–324, 1991. Earlier version available from Edinburgh as DAI Research Paper No 413.

[Constable 82] R. L. Constable. Programs as proofs. Technical Report TR 82-532, Dept. of Computer Science, Cornell University, November 1982.

[Fribourg 90] L. Fribourg. Extracting logic programs from proofs that use extended Prolog execution and induction. In *Proceedings of Eighth International Conference on Logic Programming*, pages 685 – 699. MIT Press, June 1990. Extended version in [Jacquet 93].

[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.

[Jacquet 93] J.-M. Jacquet, editor. *Constructing Logic Programs*. Wiley, 1993.

[Madden 91] P. Madden. *Automated Program Transformation Through Proof Transformation*. Unpublished PhD thesis, University of Edinburgh, 1991.

[Wiggins 90] G. A. Wiggins. The improvement of Prolog program efficiency by compiling control: A proof-theoretic view. In *Proceedings of the Second International Workshop on Meta-programming in Logic*, Leuven, Belgium, April 1990. Also available from Edinburgh as DAI Research Paper No. 455.