Negation and Control in Automatically Generated Logic Programs

Geraint A Wiggins

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

Abstract

I discuss issues of control and floundering during execution of automatically synthesised logic programs. The process of program synthesis can be restricted, without loss of generality, so that the only negated calls appearing in a program are unifications, as in [10]. If called non-ground, these predicates are *delayed* in a logic programming language with a flexible execution rule. [10] presents proposals for automatically delaying calls to predicates until they are instantiated appropriately. This can be implemented easily and safely in my synthesis approach, using meta-level knowledge about inductive proof. This technique is more general, more reliable and less laborious than the original.

1 Introduction

In this paper, I discuss an aspect of my work on the *Whelk* logic program synthesis system, which is further explained in [3], [13] and [12].

An important assumption made in the synthesis process is that one is synthesising predicates in the *all-ground* mode (*ie* with all arguments fully instantiated), and that a predicate synthesised in this way will be usable in other less fully instantiated modes. In general, this is not a safe assumption, because the synthesised program may contain calls which lead to unbounded recursion as a result of the presence of unbound variables in the top level conjecture.

Floundering under negation is a related problem. If programs are called in the all-ground mode, floundering is minimised. Nevertheless, the problem can still arise, and is exacerbated by use of the synthesised predicates in other modes.

This paper explains how these two problems can be solved through the use of meta-knowledge about inductive proof, which is the basis of the *Whelk* system. Section 2 outlines the *Whelk* system. Section 3 covers floundering under negation and the realisation of a solution originally proposed by [10], as a user-independent side effect of the synthesis technique. Section 4 shows that automatic generation of delay declarations can be more general and more reliable in the proof-based synthesis paradigm than when working with *a priori* existing programs. Section 5 summarises, draws conclusions and outlines future research.

Note that, while the examples here demonstrate the technique at work in *Whelk* synthesising Gödel programs, its operation is general, and is dependent on neither system.

2 Program Synthesis and Transformation in *Whelk*

2.1 Introduction

Whelk is a Gentzen Sequent Calculus proof development system for a first order typed logic with equality. An adaptation of the *proofs as programs* paradigm [9, 6] to synthesise functional programs allows us in certain circumstances automatically to derive programs from the proofs elaborated in *Whelk*; the necessary changes to the technique are detailed in [3, 13]. We call the adapted version *proofs as relational programs.*

The synthesised relational programs stand in a close structural relationship to the corresponding proofs. In the longer term, this will allow us to plan the construction of proofs which will lead to particular kinds of program (*eg* ones which use efficient algorithms), using adaptations of *proof planning* techniques described in [5, 2, 4, 8].

The key idea is to view the execution of our desired logic program *with no uninstantiated arguments* as evaluation of a boolean valued function. Then, we prove a *specification conjecture* which postulates that for all possible arguments of the correct type, there exists some truth value logically equivalent to the truth of the specification of that program. This is equivalent to proving that the specification is decidable, which we write thus, reading ∂ as "It is decidable whether..." (I have omitted the types here to avoid clutter):

$$\vdash \forall \overline{a}.\partial S(\overline{a})$$

where $S(\overline{a})$ is the specification of the program we wish to synthesise, and

$$\vdash \forall \overline{a}.\partial S(\overline{a}) \quad \textit{iff} \quad \vdash \exists P.\forall \overline{a}.S(\overline{a}) \leftrightarrow P(\overline{a}) \text{ and } P, \text{ a relation, is decidable}$$

Various other ways of expressing the conjecture are discussed in [13]. A fuller description of the system is given in [12].

2.2 Example: `zero/1`

For example, a specification conjecture which, when proven in *Whelk*, would lead to the construction of the `zero/1` predicate, which is true iff its argument is zero, would be:

$$\vdash \forall n.\partial(n = 0)$$

While this example is sufficiently simple to make the synthesis procedure unnecessary, it still serves as a useful example of the general framework in which synthesis is performed. See [13] for the full elaboration of the synthesis proof.

The resulting *pure logic program* [3] is

```
zero(n) ↔ n=0 ∧ true ∨ ~n=0 ∧ false.
```

and (automatic) conversion to Gödel [7] yields:

```
MODULE Zero.
IMPORT Naturals.
PREDICATE Zero :  Natural.
Zero(n) <- n=0.
```

2.3 Induction and Recursion

The proofs as programs technique relies on an intimate relationship between proofs by induction and recursive programs. The *Whelk* logic is arranged so that the identification of a subconjecture with an induction hypothesis leads to the inclusion of a recursive call in the synthesised program.

In the current *Whelk* prototype, we are limited to primitive and two-step induction on natural numbers and parametric lists. In principle, however, there is no reason for such a restriction — later, we will extend the system to include more powerful induction schemes. This will allow us to generate various algorithms for a given specification; for example, while bubble sort corresponds with primitive inductive proof of the sorting specification, quicksort corresponds with course-of-values induction. Thus, correct selection of induction schemes is crucial to the successful operation of the technique, and work is proceeding on its automation.

2.4 Non-ground use of synthesised programs

The introduction of recursive calls raises a problem. In synthesising programs this way, we suppose that we are using the all-ground mode, and *assume* that calls to the synthesised predicates which are not fully instantiated will work.

Given a language with a fixed computation rule, like Prolog, it is easy to give a counterexample to this supposition. Consider the following (logically correct) program and call.

```
append([H|T1],L,[H|T2]) :- append(T1,L,T2).
append([],L,L).

?- append(X,[],Y).
```

This is clearly a logically correct specification of the `append/3` relation, but the program never terminates, because the base case appears after the step case.

A safer specification and call might be given in Gödel:

```
MODULE Append.

IMPORT Lists.

PREDICATE Append:  List(Integer) * List(Integer) * List(Integer).

DELAY Append(X,Y,Z) UNTIL NONVAR(X) \/ NONVAR(Z).

Append([h|t1],l,[h|t2]) :- Append(t1,l,t2).
Append([],l,l).

[] <- Append(x,[],y).
```

in which case the program terminates with the message "`Floundered`". We would like the delay declaration which causes this (desirable) behaviour to be generated automatically.

I will address this problem in Section 4. First, let us consider a related problem to do with the use of negation as failure in logic programming languages.

3 Prevention of Floundering under Negation

3.1 The Problem

In his PhD thesis, [10], Lee Naish discusses the problems with the idea of negation in logic programming languages. It is unnecessary here to repeat that detail; I merely summarise by reiterating that there is a serious problem in languages involving negation but no explicit quantifiers. To borrow and slightly adapt Naish's example, consider the goal

```
?- not(X=a), X=b.
```

in Prolog. If `not` is called first, as one would normally expect, the goal fails – incorrectly – whereas otherwise it succeeds with an instantiation of `b` for `X`. In Gödel, on the other hand, the negated conjunct is *delayed* until `X` is ground, so the rightmost conjunct is executed first, giving the correct result.

The problem is that, by `not(X=a)`, we mean $\exists x.x \neq a$. What actually happens, because of negation as failure, is that we evaluate $\neg\exists x.x = a$ (*ie* $\forall x.x \neq a$). This is only a problem if we bind variables in the goal; for example, a call of `not(X=X)` evaluates to the negation of $\forall x.x = x$, *ie* $\neg\exists x.x \neq x$.

This problem generalises to the floundering of arbitrary negated goals, if any variable in the scope of a negation is bound during evaluation. The worst case is where a variable thus bound is later used outside the negation (where it remains unbound). Some languages partly defuse this problem by use of the unnamed variable "_" to mean a variable which is universally quantified, and therefore unimportant in terms of floundering due to any binding of that variable under negation. I find this solution unsatisfying, especially in a language like Gödel whose syntax already contains the necessary quantifiers.

3.2 The Solution

In [10, p19ff], Naish proposes an elegant means of removing floundering under negation. The solution is simply partially to evaluate and replace the negated predicates with new positive ones which compute the negated original. The $\neq$ predicate is then used to ensure failure where the old (unnegated) version would succeed and *vice versa*.

By removing negated goals other than =, we restrict the problem of floundering under negation to those goals, thus improving things greatly. However, it was shown in Section 3.1 that floundering can still be a problem even with such a restriction. Fortunately, a complete solution to this problem was also proposed in Section 3.1: we simply delay the negated goals (viewing them, if we wish, as constraints) until they can be executed without resulting instantiation of the variables in them. A less refined, but substantially easier to implement, approach is to require that the negated goals are delayed until either they are fully ground or there are no other goals to execute. In this case, we would wish the program to terminate with an error message warning of floundering.

Naish has already shown this approach to be correct. However, its implementation in *Whelk* is of interest as it neatly demonstrates the system at work.

3.3 The Implementation

Negation is *Whelk* is as in other constructive logic systems. The Law of the Excluded Middle does not in general hold, and $\neg P$ means that, if P is true, a contradiction can be derived. The two sequent calculus rules governing negation may be written thus:

$$\neg \text{ introduction } \frac{\Gamma, A \vdash \{\}}{\Gamma \vdash \neg A} \qquad \neg \text{ elimination } \frac{\Gamma, \neg A, \Delta \vdash A}{\Gamma, \neg A, \Delta \vdash \{\}}$$

where Γ, Δ are sequences of formulæ, A is a formula, and $\{\}$ is contradiction.

The point about these rules is that they operate only on conjectures and hypotheses whose negation is the outermost operator. The result of this is that the negated formulæ (A in the rules above) must be evaluated by rewriting or in some other way *before* the operations involving the negation itself. Further, the negation is not reflected into the synthesised program by explicit introduction of $\neg$, but by switching the boolean value associated with the formula from *true* to *false* or *vice versa*. An example will help here.

3.4 Example: `notmember/2` (base case)

[10] gives the example of negating the `member/2` predicate. This example serves nicely both here and in Section 4 so I use it too. We start off with the specification conjecture:

$$\vdash \forall n. \forall l. \partial \neg member(n, l)$$

We also need the definition of $member/2$ (which is logically equivalent to the completion of the more familiar Horn clause `member/2` definition):

$$\forall x. \neg member(x, [\,]) \tag{1}$$

$$\forall x. \forall h. \forall t. member(x, [h|t]) \leftrightarrow x = h \vee member(x, t) \tag{2}$$

The proof is by induction on l. I give the rules of the calculus as I use them. Here, I explain only the base case of the induction; the step case is left for Section 4. I present the full detail of the proof here so that the reader may form an intuition for how the technique works; the reader not needing such an intuition may skip to Section 3.5. Again, I omit types for legibility. The proof is presented in refinement style (*ie* effectively backwards). Note that I use $A\langle x/y \rangle$ to mean "A with x replaced by y" because the more usual notation is ambiguous with Prolog's and Gödel's list notation.

Proof (Base Case):

$$\vdash \forall n. \forall l. \partial \neg member(n, l)$$

Apply $\forall$ introduction $\dfrac{\Gamma, v \vdash A}{\Gamma \vdash \forall v.A}$ on n and l, synthesising the program fragment

$$notmember(n, l \ldots) \leftrightarrow \ldots$$

and leaving us with the subconjecture

$$n, l \vdash \partial\neg member(n, l)$$

Apply list induction $\dfrac{\Gamma, v, \Delta \vdash A\langle [\,]/v\rangle \qquad \Gamma, v, \Delta, v_0, v_1, A\langle v_1/v\rangle \vdash A\langle [v_0|v_1]/v\rangle}{\Gamma, v, \Delta \vdash A}$ on l, to give two cases: base case (3); and step case (4), which I defer to Section 4.4:

$$n, l \vdash \partial\neg member(n, [\,]) \tag{3}$$
$$n, l, v_0, v_1, \partial\neg member(n, v_1) \vdash \partial\neg member(n, [v_0|v_1]) \tag{4}$$

and with the following constructed program fragment:

$$\begin{array}{lcl} notmember(n, l, \ldots) & \leftrightarrow & notmember_l(n, l, \ldots) \\ notmember_l(n, l, \ldots) & \leftrightarrow & n = [\,] \wedge \ldots \vee \exists v_1.\exists v_0.y = [v_0|v_1] \wedge \ldots \end{array}$$

The auxiliary predicate, $notmember_l$, is introduced as a result of the induction rule application; auxiliaries are required in the synthesis of recursive predicates, because, in the absence of explicit higher-order induction terms (such as are used in type theory), we need names by which to refer to them when we come to the recursive call.

Next, on subconjecture (3), above, rewrite using definition (1) to give the subconjecture:

$$n, l, \forall x.\neg member(x, [\,]) \vdash \partial\neg member(n, [\,])$$

Apply $\forall$ elimination $\dfrac{\Gamma, v_0, \forall v_1.A, \Delta, A\langle v_0/v_1\rangle \vdash B}{\Gamma, v_0, \forall v_1.A, \Delta \vdash B}$ with n on the lemma to give:

$$n, l, \forall x.\neg member(x, [\,]), \neg member(n, [\,]) \vdash \partial\neg member(n, [\,])$$

Apply ∂_{true} introduction $\dfrac{\Gamma \vdash A}{\Gamma \vdash \partial A}$ giving subconjecture

$$n, l, \forall x.\neg member(x, [\,]), \neg member(n, [\,]) \vdash \neg member(n, [\,])$$

and constructed program

$$\begin{array}{lcl} notmember(n, l) & \leftrightarrow & notmember_l(n, l) \\ notmember_l(n, l) & \leftrightarrow & n = [\,] \wedge true \vee \exists v_1.\exists v_0.y = [v_0|v_1] \wedge \ldots \end{array}$$

Apply axiom $\dfrac{}{\Gamma, A, \Delta \vdash A}$ which completes this branch of the proof.

3.5 How does the method work?

The proof above may be divided into two distinct sections: before the application of ∂ introduction, and after it. These are called the *synthesis* and *verification* parts of the proof. In the synthesis part, many of the rules contribute to the construction of the new program; in the verification part, they show that synthesised program is correct.

Now, recall from Section 3.3 that the rules for negation are defined thus:

$$\neg \text{ introduction } \frac{\Gamma, A \vdash \{\}}{\Gamma \vdash \neg A} \qquad\qquad \neg \text{ elimination } \frac{\Gamma, \neg A, \Delta \vdash A}{\Gamma, \neg A, \Delta \vdash \{\}}$$

and that I have chosen to give no rule which will allow us to introduce $\neg$ under ∂. Therefore negation must be left until the last step in the synthesis part of the proof; this is the point at which the "witness" for the decidability of our specification is supplied: either *true* or $false$. Appropriate rewrite rules (*eg* de Morgan's Laws) are allowed, so that a conjecture may be rewritten into a suitable form for this to be possible. The choice of *true* or $false$ determines the polarity of the part of the synthesised predicate corresponding with the current branch of the proof, using the ∂ introduction rules (one of which was used above)

$$\partial_{true} \text{ introduction } \frac{\Gamma \vdash A}{\Gamma \vdash \partial A} \qquad \partial_{false} \text{ introduction } \frac{\Gamma \vdash \neg A}{\Gamma \vdash \partial A}$$

The rule will introduce a $\neg$ if appropriate, and proof proceeds with the verification that the "witness" was the correct one. The "witness", and not the $\neg$, will appear in in the synthesised program, controlling its success or failure in the way we want — this is why we need two rules to introduce one operator.

Because of the restrictions on proof rules for negation explained above, it is impossible to introduce a negation into a synthesised program by application of a proof rule. However, it might still be possible to do so by introduction of lemmas, or by cutting in generalisations, and so on, if this were to involve insertion of part of a synthesised program from "outside" a given proof.

To plug this loophole, we require that any formula introduced into the system and used in program construction, unless it is a formula decidable by first order unification without reference to external definitions (*eg* an equality between canonical terms), must be elaborated in the above style, with synthesis and verification proof. This approach is enforced by the proof system and cannot be circumvented. Since the proof system is restricted to handling negation in the way demonstrated above, it is impossible for negations other than $\neq$ (between canonical terms and/or variables) to appear. These goals are acceptable, as we are able to deal with them by delaying (and/or by viewing them as constraints) in any reasonable logic programming language.

Finally, note that the initial conjecture of the above example need not be the top level conjecture of a theorem, but may be produced by prior application of rules to a more complicated conjecture. Similarly, it is possible in many circumstances partially to evaluate such conjectures as parts of larger formulæ; this generally produces interleaving as in [1], which I discuss in [11].

4 Automatic Generation of Delay Declarations

4.1 More General Delay Declarations

Having suggested how we may go about using the built-in delay capability of (*eg*) Gödel, it is now appropriate to ask how we might go about extending the idea to generating our own delay declarations.

The idea is that we want to prevent unbounded recursion in our synthesised programs by preventing predicates from being called before the arguments which control their recursion are sufficiently instantiated so to do.

The approach I will propose in this section is closely related to that suggested in [10, p26ff], in that it uses meta-knowledge about the recursive structure of programs and data-types to infer which arguments to a predicate are significant in controlling recursion. In my approach, however, because of the extra meta-knowledge contained in a *Whelk* proof, we can be definite about which the significant arguments are, and we can reach this conclusion much more easily and reliably, as a side-effect of the proof process, requiring no *post hoc* analysis.

4.2 Detecting Recursive Arguments

Naish's approach to detecting recursive arguments in a predicate is based on *post hoc* analysis of an existing program. This carries with it all the implications of any technique founded on the same precepts: even if the recursive structure is obvious to the informed human reader, an automated

analysis may be laborious and difficult. In arbitrary programs, the significant arguments may be very hard to spot, even for experienced programmers. Naish presents a compact algorithm to carry out the process, but acknowledges that its worst case complexity is exponential with the size of the input (with the comment that this seems in practice not to be significant, and that linear behaviour is the norm; other authors have more efficient algorithms based on the same idea).

Naish's summary of the algorithm runs thus ([10, p34]):

for-each pair L, of unifiable clause heads and recursive calls **do**
  if the head is as general as the call **then**
    terminate with failure
  else
    for-each argument I, less general in the head **do**
      add a wait declaration to wait group L,
      with 0 in argument I and 1 in all other arguments
    end-for
  end-if
end-for
$Allwaits = \{\ W \mid W$ is the intersection of one wait from each group $\}$
$Waits = \{\ W \mid W \in Allwaits \wedge \forall V. V \in Allwaits \rightarrow W \not\subset V\ \}$

Waits is the value we want here. It is a set of *wait declarations*. A *wait declaration* is a specification of which arguments must be non-variable for a given predicate to be called. For example:

```
?- wait p(0,1,1).
```

states that the predicate `p/3` must only be executed when its first argument is at least partly instantiated. The algorithm generates a number of wait declarations for each predicate in the program over which it works - some of these can usually be discarded because they are subsumed by others. When applied to an appropriate logic programming language, the wait declarations change the order of execution, delaying the analysed predicates until their significant arguments are non-variable.

Generation of these declarations is based on the structurally recursive data-types in the arguments to the program being analysed: this can be seen in the inner **for-each** loop – the instantiation of the clause head and the recursive call are compared and it is required that the head argument be *less general* – that is, *more* instantiated. Thus, if the predicate is called with this argument instantiated, the data is broken down by unification; given a well-founded recursive datatype in the argument position, this gives well-founded recursion in the predicate. For example, list destruction involves an argument in the head of the form `[H|T]`, and a corresponding recursive argument of the form `T`. Note, incidentally, that Naish's algorithm does not require that the recursive call be on the same variable as named in the head variable – *ie* in this example, the variable need not be `T`; any variable will do. This causes over-generality in the loop checking and thence over-caution in the wait declarations — sometimes perfectly executable programs are delayed so much that they flounder.

Naish's algorithm cannot deal with recursive datatypes whose constructors are not explicit in a program (*eg* integers); nor is mutual recursion apparently covered (unless this is included in the "recursive calls" in the first **for-each** of the algorithm, which would be begging hard questions).

Naish demonstrates his algorithm working with two quite hard examples: n-queens, and a term-ordering predicate. It works well in these restricted cases.

4.3 Delay Generation from Inductive Proof Structure

While the technique I propose is very similar to Naish's in its theoretical basis, in terms of execution it is fundamentally different. As I mentioned in Section 2, inductive proof and the construction of recursive programs in *Whelk* correspond one-to-one — if we prove a synthesis conjecture by induction, we necessarily get a recursive program.

Now, "recursive" datatypes are defined inductively. Thus, they are known *a priori* to be well-founded, and so can be used to control recursion in the same way as they provide a basis for induction. What is more, when we choose a particular variable as the induction variable (as with l in Section 3), we always construct a corresponding argument position in the synthesised program — as a direct result of applying the induction rule. Therefore, it is trivial to generate a wait declaration (or, better, a more general *delay* declaration as in Gödel), for that argument.

4.4 Example: `notmember/2` (step case)

I now return to the `notmember/2` example, starting from where we left off: conjecture (4) in Section 3.4. I will use the definition of `member/2` given in Section 3 and an axiom about the decidability of equality:

$$\vdash \forall x.\forall y.x = y \vee \neg x = y \tag{5}$$

Recall that the proof of the base case (from Section 3.4) has given this constructed program fragment:

$$\begin{array}{lcl} notmember(n,l) & \leftrightarrow & notmember_l(n,l) \\ notmember_l(n,l) & \leftrightarrow & n = [\,] \wedge true \vee \exists v_1.\exists v_0.y = [v_0|v_1] \wedge \ldots \end{array}$$

We reached this stage by application of induction on l, and proof of the resulting base case. Because our synthesised program arises from a proof by induction on l, its recursion is necessarily controlled by any argument(s) corresponding with l. Therefore, we can generate a simple delay declaration (here in Gödel):

```
DELAY Notmember(n,l) UNTIL NONVAR(l)
```

The proof now proceeds as follows. Again, I present the full detail of the proof; the reader not interested in that detail should skip to Section 4.5.

Proof (step case):

$$n, l, v_0, v_1, \partial\neg member(n, v_1) \vdash \partial\neg member(n, [v_0|v_1])$$

Rewrite according to definition (2):

$$n, l, v_0, v_1, \partial\neg member(n, v_1) \vdash \partial(\neg(n = v_0 \vee member(n, v_1)))$$

Rewrite under de Morgan law:

$$n, l, v_0, v_1, \partial\neg member(n, v_1) \vdash \partial(\neg n = v_0 \wedge \neg member(n, v_1))$$

Apply $\wedge$ introduction under ∂ $\dfrac{\Gamma \vdash \partial C \qquad \Gamma \vdash \partial D}{\Gamma \vdash \partial(C \wedge D)}$ giving subconjectures

$$n, l, v_0, v_1, \partial\neg member(n, v_1) \vdash \partial\neg n = v_0 \tag{6}$$

$$n, l, v_0, v_1, \partial\neg member(n, v_1) \vdash \partial\neg member(n, v_1) \tag{7}$$

and constructed program fragment

$$\begin{array}{lcl} notmember(n,l) & \leftrightarrow & notmember_l(n,l) \\ notmember_l(n,l) & \leftrightarrow & n = [\,] \wedge true \vee \exists v_1.\exists v_0.y = [v_0|v_1] \wedge \ldots \wedge \ldots \end{array}$$

On subconjecture (6) (omitting the induction hypothesis, which we do not need here), introduce decidability axiom (5):

$$n, l, v_0, v_1, \ldots, \forall x.\forall y.x = y \vee \neg x = y \vdash \partial\neg n = v_0$$

Substitute values by $\forall$ elimination, as before:

$$n, l, v_0, v_1, \forall x.\forall y.x = y \vee \neg x = y, n = v_0 \vee \neg n = v_0 \vdash \partial\neg n = v_0$$

Apply $\vee$ elimination $\dfrac{\Gamma, A \vee B, \Delta, A \vdash C \quad \Gamma, A \vee B, \Delta, B \vdash C}{\Gamma, A \vee B, \Delta \vdash C}$ to give subconjectures

$$n, l, v_0, v_1, \forall x.\forall y.x = y \vee \neg x = y, n = v_0 \vee \neg n = v_0, n = v_0 \vdash \partial\neg n = v_0 \quad (8)$$
$$n, l, v_0, v_1, \forall x.\forall y.x = y \vee \neg x = y, n = v_0 \vee \neg n = v_0, \neg n = v_0 \vdash \partial\neg n = v_0 \quad (9)$$

and constructed program fragment

$$\begin{array}{lcl} notmember(n,l) & \leftrightarrow & notmember_l(n,l) \\ notmember_l(n,l) & \leftrightarrow & n = [\,] \wedge true \vee \\ & & \exists v_1.\exists v_0.y = [v_0|v_1] \wedge \\ & & \qquad (n = v_0 \wedge \ldots \vee \neg n = v_0 \wedge \ldots) \wedge \ldots \end{array}$$

Apply ∂ introduction, as before, with $false$ in (8) and $true$ in (9):

$$n, l, v_0, v_1, \forall x.\forall y.x = y \vee \neg x = y, n = v_0 \vee \neg n = v_0, n = v_0 \vdash \neg\neg n = v_0$$
$$n, l, v_0, v_1, \forall x.\forall y.x = y \vee \neg x = y, n = v_0 \vee \neg n = v_0, \neg n = v_0 \vdash \neg n = v_0$$

The constructed program fragment is now:

$$\begin{array}{lcl} notmember(n,l) & \leftrightarrow & notmember_l(n,l) \\ notmember_l(n,l) & \leftrightarrow & n = [\,] \wedge true \vee \\ & & \exists v_1.\exists v_0.y = [v_0|v_1] \wedge \\ & & \qquad (n = v_0 \wedge false \vee \neg n = v_0 \wedge true) \wedge \ldots \end{array}$$

This part of the program is now complete; the rest of this branch of the proof is trivial verification, using rules already demonstrated above.

Finally, to subconjecture (7), above:

$$n, l, v_0, v_1, \partial\neg member(n, v_1) \vdash \partial\neg member(n, v_1)$$

apply axiom as before. Note that in this case, the application of the induction rule causes an appropriate recursive call to be associated with the induction hypothesis. This is now inserted into the constructed program, to give the finished article:

$$\begin{array}{lcl} notmember(n,l) & \leftrightarrow & notmember_l(n,l) \\ notmember_l(n,l) & \leftrightarrow & n = [\,] \wedge true \vee \\ & & \exists v_1.\exists v_0.y = [v_0|v_1] \wedge \\ & & \qquad (n = v_0 \wedge false \vee \neg n = v_0 \wedge true) \wedge \\ & & \qquad notmember_l(n, v_1) \end{array}$$

4.5 The `Notmember/2` Module

The proof gives rise, automatically, to the Gödel module shown in Figure 1. Consider the behaviour of the program called with the goal

```
[] <- Notmember(0,[0|t]).
```

We wish this to fail, and indeed it does so. Now, suppose we give the following goal, which we would like explicitly to flounder, with an error message:

```
[] <- Notmember( 0, [h|t] ).
```

```
MODULE Notmember.

IMPORT Lists.
IMPORT Naturals.

PREDICATE Notmember:  Natural * List(Natural).

Notmember(n,l)      <-  Notmember_l(n,l)

PREDICATE Notmember_l:  Natural * List(Natural).

DELAY Notmember(n,l) UNTIL NONVAR(l).

Notmember_l(n,l)    <-  l=[] & true \/
                        Some [v1] Some [v0] y=[v0|v1] & ~n=v0 &
                                              Notmember_l(n,v1)
```

Figure 1: The Gödel Module for `Notmember/2`

Our delay declaration admits this, as far as the first call to `Notmember_l/2`, because the second argument is non-variable. However, at this point there will be a call to `=/2` with one argument uninstantiated. Then, the default behaviour is to delay, so the recursive call is made to `Notmember/2`. This time, however, the second argument is a fully uninstantiated variable, so this call too is delayed. Therefore, the whole computation flounders, explicitly, as we would wish, and an error is reported.

4.6 More Subtle Control Generation

While this approach will work well for many cases, some similar problems are harder. Let us return now to the `append/3` definition I gave in Section 2. I gave the delay declaration

```
DELAY Append(x,y,z) UNTIL NONVAR(x) \/ NONVAR(z).
```

requiring that *either* the first *or* the last argument be instantiated before the predicate is executed.

The `append/3` predicate is different from `member/2` in that its recursion is controlled by either of two variables – `x` or `z` in the declaration above. How can we spot this? Simply enough, it arises from the use of a more powerful induction scheme: simultaneous primitive recursion on two variables, as defined by the following rule (writing the preconditions of the rule vertically):

$$\frac{\begin{array}{rcl}\Gamma, u, v, \Delta & \vdash & A\langle [\,]/u\rangle \\ \Gamma, u, v, \Delta, u_0, u_1 & \vdash & A\langle [u_0|u_1]/u\rangle\langle [\,]/v\rangle \\ \Gamma, u, v, \Delta, u_0, v_0, u_1, v_1, A\langle u_1/u\rangle\langle v_1/v\rangle & \vdash & A\langle [u_0|u_1]/u\rangle\langle [v_0|v_1])/v\rangle\end{array}}{\Gamma, u, v, \Delta \vdash A}$$

This scheme is equivalent to primitive induction on x, followed by primitive induction on y in x's step case. Use of such a scheme begs the question: how do we know which scheme to use? This is outside the scope of this paper, but is addressed in [4].

There follows a sketch of the proof. Note that we are synthesising the program from an equivalent definition of `append/3` here. This is not a failure of the technique — it is impossible to specify the simple example program in any other logical terms. We gain from the proof process because the delay declarations are generated for us, and because our specification is shown to be realisable.

4.7 Example: append/3

Lemmas:

$$\vdash \forall x.app([\,],x) = x \tag{10}$$
$$\vdash \forall h.\forall t.\forall y.app([h|t],y) = [h|app(t,y)] \tag{11}$$
$$\vdash \forall x.\forall y.x = y \vee \neg x = y \tag{12}$$
$$\vdash \forall h.\forall t.\neg[h|t] = [\,] \tag{13}$$
$$\vdash \forall h_1.\forall h_2.\forall t_1.\forall t_2.[h_1|t_1] = [h_2|t_2] \leftrightarrow h_1 = h_2 \wedge t_1 = t_2 \tag{14}$$

Proof:

$$\vdash \forall x.\forall y.\forall z.\partial(app(x,y) = z)$$

Introduce x, y and z. Apply simultaneous primitive induction on x and z:

$$x,y,z \vdash \partial(app([\,],y) = z) \tag{15}$$
$$x,y,z,x_0,x_1 \vdash \partial(app([x_0|x_1],y) = [\,]) \tag{16}$$
$$x,y,z,x_0,x_1,z_0,z_1,\partial(app(x_1,y) = z_1) \vdash \partial(app([x_0|x_1],y) = [z_0|z_1]) \tag{17}$$

In (15), and use (12) above to case split on equality:

$$x,y,z,y = z \vdash \partial(app([\,],y) = z) \tag{18}$$
$$x,y,z,\neg y = z \vdash \partial(app([\,],y) = z) \tag{19}$$

(18) is completed by ∂_{true} introduction, (19) by ∂_{false}, verified using (10).

For (16), rewrite under $=$ using (11); then ∂_{false} introduction can be verified by (13). In (17), the step case, rewrite the left hand side of the equation in the conjecture according to (11):

$$x,y,z,x_0,x_1,z_0,z_1,\partial(app(x_1,y) = z_1) \vdash \partial([x_0|app(x_1,y)] = [z_0|z_1])$$

Next, rewrite using (14) to give:

$$x,y,z,x_0,x_1,z_0,z_1,\partial(app(x_1,y) = z_1) \vdash \partial(x_0 = z_0 \wedge app(x_1,y) = z_1)$$

Finally, $\wedge$ introduction gives us two subgoals:

$$x,y,z,x_0,x_1,z_0,z_1,\partial(app(x_1,y) = z_1) \vdash \partial(x_0 = z_0) \tag{20}$$
$$x,y,z,x_0,x_1,z_0,z_1,\partial(app(x_1,y) = z_1) \vdash \partial(app(x_1,y) = z_1) \tag{21}$$

(20) is solved with a case split on (12), as for conjecture (15). (21) is identical to the induction hypothesis, and so we have finished the whole proof. The finished program then looks like this:

$$\begin{array}{lcl} app(x,y,z) & \leftrightarrow & app_l(x,y,z) \\ app_l(x,y,z) & \leftrightarrow & x = [\,] \wedge (y = z \wedge true \vee \neg y = z \wedge false) \vee \\ & & \exists v_0.\exists v_1.x = [v_0|v_1] \wedge \\ & & \quad (z = [\,] \wedge false \vee \\ & & \quad \exists v_2.\exists v_3.z = [v_2|v_3] \wedge \\ & & \quad\quad (v_0 = v_2 \wedge true \vee \neg v_0 = v_2 \wedge false) \wedge \\ & & \quad\quad app_l(v_1,y,v_3) \end{array}$$

Since we performed the proof by simultaneous induction on x and z, we can generate the delay declaration we need in the same way as before, *dis*joining the requirements that each argument be non-variable since clearly *either* induction variable, and not necessarily *both* will be enough to control the recursion. Thus, the general form of delay declarations generated by *Whelk* will be disjunctive. *Con*joined delays will never arise, because of the form of the programs; each argument is represented by exactly one variable, and each induction corresponds with the introduction of a

new recursive predicate. Since it is not possible to apply two proof rules at once, only disjunctive delays can be generated.

Finally, it is worth mentioning that the technique will still work even if the proof is elaborated by two separate applications of ordinary primitive induction on x and then z – though the program produced will be slightly uglier. This, however, is not important, as we still have the specification to work with.

5 Conclusion and Further Work

In this paper, I have explained how an existing technique may be applied in a new way within my program synthesis system to allow the automatic generation of delay declarations, thus allowing my synthesised programs to benefit from facilities for control within modern logic programming languages.

The new approach to the technique relies on meta-knowledge about the synthesised program encorporated in the synthesis proof, and not on *post hoc* analysis. Thus, the information it has to work with is complete and correct, and need never be guessed by analysing the syntactic form of a program. Therefore, the technique applies to datatypes which do not have explicit constructor functions, unlike earlier approaches. It is trivial to generate delay declarations for many recursive programs synthesised by the proofs as programs technique, modulo the question of choosing the right induction scheme which is addressed elsewhere.

This approach is possible only because of the initial decision to reason with specifications instead of programs. It is just one of several areas (see *eg* [8]) where this approach facilitates program construction and manipulation.

The next step in this work will be to build these ideas into the existing *Whelk* system. This begs a question of the level at which such reasoning should take place: either at the object level, in *Whelk*, or at the meta-level, in the CLaM proof planner, with which we will automate the search for synthesis proofs over the next few years. It seems likely that the latter option is best, in which case a full implementation will be some time away.

6 Acknowledgements

This work is funded by ESPRIT Basic Research Action #3012, "Computational Logic". Thanks to Alan Bundy and the DReaMers for their interest and support; and to John Lloyd and Wolfgang Bibel for raising the problems addressed here.

References

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

[2] A. Bundy, A. Smaill, and J. Hesketh. Turning eureka steps into calculations in automatic program synthesis. In S. L.H. Clarke, editor, *Proceedings of UK IT 90*, pages 221–6. IEE, 1990. Also available from Edinburgh as DAI Research Paper 448.

[3] 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.

[4] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. Research Paper 567, Dept. of Artificial Intelligence, University of Edinburgh, 1991. In the Journal of Artificial Intelligence.

[5] 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.

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

[7] P. M. Hill and J. W. Lloyd. The Gödel Programming Language. Technical Report CSTR-92-27, Department of Computer Science, University of Bristol, 1992. Revised May 1993. To be published by MIT Press.

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

[9] Per Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis, Naples, 1984. Notes by Giovanni Sambin of a series of lectures given in Padua, June 1980.

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

[11] 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.

[12] G. A. Wiggins. Synthesis and transformation of logic programs in the Whelk proof development system. In K. R. Apt, editor, *Proceedings of JICSLP-92*, pages 351–368. M. I.T. Press, Cambridge, MA, 1992.

[13] G. A. Wiggins, Alan Bundy, I. Kraan, and J. Hesketh. Synthesis and transformation of logic programs through constructive, inductive proof. In K-K. Lau and T. Clement, editors, *Proceedings of LoPSTr-91*, pages 27–45. Springer Verlag, 1991. Workshops in Computing Series.