MAX-PLANCK-INSTITUT FÜR INFORMATIK

Logic Program Synthesis via Proof Planning

Ina Kraan
David Basin
Alan Bundy

MPI–I–92–244 October 1992

Im Stadtwald
W 6600 Saarbrücken
Germany

Authors' Addresses

Ina Kraan, Department of Artificial Intelligence, University of Edinburgh, Edinburgh, Scotland, U.K. inak@ai.ed.ac.uk
David Basin, Max-Planck-Institut für Informatik, Saarbrücken, Germany. basin@mpi-sb.mpg.de
Alan Bundy, Department of Artificial Intelligence, University of Edinburgh, Edinburgh, Scotland, U.K. bundy@ai.ed.ac.uk

Publication Notes

A version of this paper will appear in the proceedings of LoPSTr-92, published in Springer Verlag LNCS series.

Acknowledgements

David Basin was supported by the German Ministry for Research and Technology (BMFT) under grant ITS 9102. Responsibility for the contents of this publication lies with the authors. Alan Bundy was supported under SERC grant GR/E/44598, Esprit BRA grant 3012, Esprit BRA grant 3245, and an SERC Senior Fellowship.

Abstract

We propose a novel approach to automating the synthesis of logic programs: Logic programs are synthesized as a by-product of the planning of a verification proof. The approach is a two-level one: At the object level, we prove program verification conjectures in a sorted, first-order theory. The conjectures are of the form $\forall \overrightarrow{args}.\ prog(\overrightarrow{args}) \leftrightarrow spec(\overrightarrow{args})$. At the meta-level, we plan the object-level verification with an unspecified program definition. The definition is represented with a (second-order) meta-level variable, which becomes instantiated in the course of the planning.

This technique is an application of the Clam proof planning system. Clam is currently powerful enough to plan verification proofs for given programs. We show that, if Clam's use of middle-out reasoning is extended, it will also be able to synthesize programs.

1 Introduction

The aim of the work presented here is to automate the synthesis of logic programs. This is done by adapting techniques from areas such as *middle-out reasoning* in *explicit proof plans* [Bundy 88, Bundy *et al* 90a], *proofs-as-programs* [Bates & Constable 85] and *deductive synthesis* [Bibel 80]. We synthesize *pure logic programs* [Bundy *et al* 90b] from specifications in sorted, first-order theories. The approach encompasses two levels of reasoning: An object level, which is a sorted, first-order predicate logic with equality, and a meta-level, which reasons explicitly with object-level proofs. At the object level, we prove that the specification and the program are logically equivalent, which ensures the partial correctness and completeness of the program [Hogger 81]. At the meta-level, we construct a plan for the object-level proof. While planning, we represent the body of the program we are synthesizing with a meta-level variable. The use of meta-level variables in proof planning is called *middle-out reasoning*. Synthesis takes place when, in the course of planning, the meta-level variable representing the body of the program is instantiated to an object-level term. However, this term may not always correspond to a pure logic program. If it does not, an auxiliary synthesis is required.

The approach is embedded within the framework of the Clam proof planner [Bundy *et al* 90c]. Clam is currently powerful enough to conduct verification proofs for conjectures containing no meta-level variables. To synthesize programs in the way we are proposing here, however, Clam's use of middle-out reasoning will have to be extended.

The remainder of this paper is organized as follows: Section 2 discusses related work. Section 3 contains a definition of pure logic programs. Section 4 provides a brief introduction to proof planning, middle-out reasoning and rippling. Section 5 shows how verification proofs for a given specification and a given program can be planned, and Section 6 shows how programs can be synthesized by leaving the program unspecified when planning a verification proof. Section 7 contains a summary and suggestions for future work.

2 Related Work

In program synthesis from specifications[1], there are two main approachs, i.e., *proofs-as-programs* [Bates & Constable 85] and *deductive synthesis* [Bibel 80, Biundo 88].

Proofs-as-programs is based on what is known as the *Curry-Howard isomorphism* [Howard 80], whereby a proposition is identified with a type of terms in the λ-calculus that represent evidence for its truth. Under this isomorphism, a proposition is true if and only if the corresponding type has members. A proof of a proposition will construct such a member. Since terms in the λ-calculus

[1] As opposed to synthesis from input-output tuples, for example.

may be evaluated, proofs give rise to functional programs. For example, given the proposition[2]

$$\forall \overrightarrow{input}.\exists output.\ spec(\overrightarrow{input}, output)$$

a proof of the proposition will construct a program f such that, for all inputs, f yields an output that satisfies the specification, i.e., $spec(\overrightarrow{input}, f(\overrightarrow{input}))$ holds. These ideas underlie the Nuprl system [Constable *et al* 86] and its Edinburgh reimplementation Oyster [Bundy *et al* 90c], which are interactive proof development systems for a variant of Martin-Löf type theory[Martin-Löf 79].

Adapting proofs-as-programs to logic program synthesis is not straightforward. The main problem is that proofs-as-programs synthesizes total functions. Logic programs, however, are partial and multivalued [Bundy *et al* 90b]. They may return no value, i.e., fail, or they may return more than one value on backtracking. Moreover, they may not terminate.

One adaptation of proofs-as-programs to logic program synthesis is presented in [Fribourg 90]. Fribourg synthesizes programs from Prolog-style proofs. He extends standard Prolog goals to goals of the form $\forall \vec{x}.\exists \vec{y}.\ q(\vec{x}, \vec{y}) \Leftarrow r(\vec{x})$, where $q(\vec{x}, \vec{y})$ and $r(\vec{x})$ are conjunctions of atoms, and he extends standard Prolog SLD-resolution to the rules of *definite clause inference*, *simplification* and *restricted structural induction*, each of which is associated with a program construction rule. Given an appropriate specification, extended Prolog execution returns a program to compute $\vec{y}$ in terms of $\vec{x}$. However, the program is only correct if it is called with the variables $\vec{x}$ ground and the variables $\vec{y}$ unbound. Also, it will return exactly one answer. It is thus a functional program in the guise of a logic program.

To overcome these disadvantages, [Bundy *et al* 90b] suggests viewing logic programs in all-ground mode as functions returning a boolean value. A specification of a logic program is then:

$$\forall \overrightarrow{args}.\exists boole.\ spec(\overrightarrow{args}) = boole$$

If such specification theorems are proved in type theory, e.g., with the Oyster system, the programs are higher-order and functional. Such programs are difficult to translate into equivalent logic programs. Therefore, [Bundy *et al* 90b] suggests working with a constructive first-order logic in which the extract terms are pure logic programs.

This idea was pursued in [Wiggins *et al* 91] and has been implemented in Whelk, an interactive proof editor for logic program synthesis. The Whelk system distinguishes between the logic of the specification and the logic of the program. The two are related by a mapping from the program logic to the specification logic. Each inference rule in the specification logic corresponds to a program construction rule in the program logic. A major concern is proving the correctness of the rules [Wiggins 92].

[2]Here, and in the following, we often omit sort or type information to avoid notational clutter.

In *deductive synthesis*, a set of transformation rules is applied to a given specification to derive a program. For instance, [Biundo 88] starts with a specification formula $\forall\vec{x}.\exists y.\forall\vec{z}.\ \Phi[\vec{x}, y, \vec{z}]$, where Φ is a quantifier-free first-order formula. Biundo Skolemizes the formula to $\forall\vec{x}.\forall\vec{z}.\ \Phi[\vec{x}, f(\vec{x}), \vec{z}]$ and applies transformation rules to the Skolemized specification until a program is obtained that computes the Skolem function $f(\vec{x})$. Her rules include *evaluation*, *substitution*, *case analysis* and *induction*. Transformation rules must be proved sound if the correctness of the program is to be guaranteed.

Our approach to synthesis can be related both to proof-as-programs and deductive synthesis. On one hand, we are proving

$$\forall\vec{args}.\ prog(\vec{args}) \leftrightarrow spec(\vec{args})$$

where the definition of *prog* is unknown. This is similar to proving the (higher-order) specification

$$\forall\vec{args}.\exists prog.\ prog(\vec{args}) \leftrightarrow spec(\vec{args})$$

constructively, since a constructive proof requires showing how a witness for an existentially quantified variable can be constructed. Thus our approach can be seen as proofs-as-programs. On the other hand, proof planning consists of the successive application of methods to a conjecture, where each method transforms the conjecture into another one. Each method can thus be perceived as a transformation rule.

3 Pure Logic Programs

Our notion of pure logic programs is similar to pure logic programs as defined in [Bundy *et al* 90b] and to logic descriptions as defined in [Deville 90]. In Deville's approach, logic program development is a two-stage process. First, a pure logic description is obtained from a specification in a subset of natural language. Then, the program is derived from the logic description. Deville's reasons for choosing logic descriptions as an intermediate representation are the same as ours for synthesizing pure logic programs. Pure logic programs are a subset of first-order predicate logic and thus share its purely declarative semantics. Pure logic programs are not meant to be directly executed, yet their syntax is sufficiently restricted that they are straightforward to translate into executable programs in logic programming languages, e.g., Prolog or Gödel [Hill & Lloyd 91]. We are thus not restricted to any particular logic programming language.

For the purpose of this paper, pure logic programs are collections of sentences of the form

$$\forall x_1{:}t_1, \ldots, x_n{:}t_n.\ pred(x_1, \ldots, x_n) \leftrightarrow body$$

where *pred* is a predicate symbol, the x_i are distinct variables of sorts t_i and *body* is a pure logic program body. Only one definition per predicate symbol is allowed. Pure logic program bodies are defined recursively:

- The predicates $true$ and $false$ are pure logic program bodies.
- A member of a predefined set of decidable atomic relations is a pure logic program body[3].
- A call to a previously defined predicate is a pure logic program body.
- If P and Q are pure logic program bodies, then
  - $P \wedge Q$
  - $P \vee Q$
  - $\exists x.\ P$

  are pure logic program bodies.

Other connectives such as negation or implication can be added. Avoiding those, however, largely eliminates floundering, without restricting the expressive power of the language.

An example of a pure logic program is:

$$\begin{array}{rcl} \forall x, l.\ member(x, l) & \leftrightarrow & \exists h, t.\ l = [h|t] \wedge (x = h \vee member(x, t)) \\ \forall i, j.\ subset(i, j) & \leftrightarrow & i = [\,] \vee \\ & & \exists h, t.\ i = [h|t] \wedge member(h, j) \wedge subset(t, j) \end{array}$$

The predicate $member(x, l)$ is true if x is a member of the list l, the predicate $subset(i, j)$ is true if i is a subset of j. Translated into Prolog, for instance, they become:

$$\begin{array}{l} member(X, [X|\_]). \\ member(X, [\_|T]) \leftarrow member(X, T). \\ \\ subset([\,], \_). \\ subset([H|T], J) \leftarrow member(H, J), subset(T, J). \end{array}$$

The pure logic program is the completion of the Prolog program.

4 Proof Planning

The central problem of automated theorem proving is the enormous search space for proofs. Some theorem provers, e.g., NQTHM [Boyer & Moore 88], use heuristics to decide when to apply which inference rule. These heuristics are often built-in, which makes them inflexible and difficult to understand. To avoid this, [Bundy 88] suggests using a meta-logic to reason about and to plan proofs.

[3]For the purpose of this paper, the set consists of equality (=) and inequality (≠).

Proof plans are combinations of *methods*, which are specifications of *tactics*. A tactic is a program that applies a number of object-level inference rules to a goal formula. A method is a specification of a tactic in the sense of the assertion: If a goal formula matches the input pattern and if the preconditions are met, the tactic is applicable, and, if the tactic succeeds, the output conditions (or effects) will be true of the resulting goal formulae. These ideas are the basis of the proof planner Clam [Bundy *et al* 90c]. Clam constructs proof plans that can be executed in Oyster.

Middle-out reasoning [Bundy *et al* 90a] extends the meta-level reasoning of proof planning in that it allows the meta-level representation of object-level entities to contain meta-level variables. This allows proof planning to proceed even though an object-level entity is not fully specified. Thus, it is possible to postpone a decision about the entity's real identity. Clam currently uses middle-out reasoning to synthesize tail-recursive programs from non-tail-recursive specifications and to generalize inductive theorems. We will extend Clam's use of middle-out reasoning significantly. In particular, we will use meta-level variables to represent unspecified parts of logic programs.

Clam is particularly good at proving theorems by induction. Its power stems from the *rippling* method, which is central to proving the step case(s) of inductive proofs. In the step case, the overall strategy is to manipulate the induction conclusion in such a way that it is possible to exploit the induction hypothesis. Rippling does this by keeping track of the differences between the induction hypothesis and the induction conclusion and applying rewrites to the induction conclusion to reduce these differences.

Rippling is best illustrated by an example. Clam would represent the step case of the proof of the associativity of plus as

$$\begin{array}{l}(x+y)+z=x+(y+z)\\ \vdash\\ (\boxed{s(\underline{x})}^{\uparrow}+y)+z=\boxed{s(\underline{x})}^{\uparrow}+(y+z)\end{array}$$

where s represents the successor function. The boxes and underlining are meta-level annotations. The non-underlined parts in the boxes are *wave fronts*—they do not appear in the induction hypothesis. The underlined parts in the boxes are *wave holes*. The wave holes and the remaining parts of the induction conclusion are called the skeleton—strung together they form the induction hypothesis. The arrows indicate the direction in which the wave fronts are moving, in this case up the term tree of the induction conclusion. Rippling is the exhaustive application of a set of rewrite rules called *wave rules*. Wave rules are also annotated. They are applied only if the wave rule and a subexpression of the induction conclusion match, including annotations. The annotation on the wave rule ensures that applying it will move the wave front up in the term tree of the induction conclusion. Often, all wave fronts can be rippled to the top of the term tree of the induction conclusion, which means that the induction hypothesis can

be exploited. The wave rules required for our example proof are

$$\boxed{s(\underline{M})}^{\uparrow} + N \quad \Rightarrow \quad \boxed{s(\underline{M+N})}^{\uparrow} \tag{1}$$

$$\boxed{s(\underline{M})}^{\uparrow} = \boxed{s(\underline{N})}^{\uparrow} \quad \Rightarrow \quad M = N \tag{2}$$

where M and N are free variables. Clam generates these wave rules automatically from the definition of $+$ and the substitution axiom for s. The rippling of the example consists of three applications of wave rule (1) (two on the left- and one on the right-hand side) and one of wave rule (2):

$$\begin{aligned}
(\boxed{s(\underline{x})}^{\uparrow} + y) + z &= \boxed{s(\underline{x})}^{\uparrow} + (y+z) \\
\boxed{s(\underline{x+y})}^{\uparrow} + z &= \boxed{s(\underline{x})}^{\uparrow} + (y+z) \\
\boxed{s(\underline{(x+y)+z})}^{\uparrow} &= \boxed{s(\underline{x})}^{\uparrow} + (y+z) \\
\boxed{s(\underline{(x+y)+z})}^{\uparrow} &= \boxed{s(\underline{x+(y+z)})}^{\uparrow} \\
(x+y)+z &= x+(y+z)
\end{aligned}$$

Not only has the wave front moved to the top of the induction conclusion, but it has also disappeared. The induction conclusion is now identical to the induction hypothesis, and the step case is complete. This final step is called *strong fertilization.*

Rippling will be the key method in planning the step cases of the verifications proofs. Other methods we will use in the following sections are induction, symbolic evaluation, tautology checking and unblocking. What these methods do will become apparent in the discussion of the proofs.

5 Verification

In this section, we show how Clam's existing methods can be used to plan the verification proof for a given program. Our verification conjectures, which we prove classically, are first-order sentences of the form:

$$\forall \overrightarrow{args}.\ prog(\overrightarrow{args}) \leftrightarrow spec(\overrightarrow{args})$$

The logical equivalence of the specification and the program guarantees the partial correctness and completeness of the program with respect to the specification [Hogger 81].

We show how Clam plans proofs for such conjectures using the example conjecture

$$\forall i, j.\ subset(i,j) \quad \leftrightarrow \quad (\forall x.\ member(x,i) \rightarrow member(x,j)) \tag{3}$$

where the program *subset* is defined as

$$\begin{array}{lcl} \forall i, j.\ subset(i, j) & \leftrightarrow & i = [\,] \vee \\ & & \exists h, t.\ i = [h|t] \wedge member(h, j) \wedge subset(t, j) \end{array}$$

and *member* in the program and the specification is defined as:

$$\forall x, l.\ member(x, l) \quad \leftrightarrow \quad \exists h, t.\ l = [h|t] \wedge (x = h \vee member(x, t))$$

The definitions of *subset* and *member* give rise to the following wave rules:

$$subset(\boxed{[H|\underline{T}]}^{\uparrow}, J) \quad \Rightarrow \quad \boxed{member(H, J) \wedge \underline{subset(T, J)}}^{\uparrow} \tag{4}$$

$$member(X, \boxed{[H|\underline{T}]}^{\uparrow}) \quad \Rightarrow \quad \boxed{X = H \vee \underline{member(X, T)}}^{\uparrow} \tag{5}$$

We also need the following wave rules, which are derived from lemmas:

$$\boxed{P \vee \underline{Q}}^{\uparrow} \rightarrow R \quad \Rightarrow \quad \boxed{P \rightarrow R \wedge \underline{Q \rightarrow R}}^{\uparrow} \tag{6}$$

$$\forall x.\ \boxed{P \wedge \underline{Q}}^{\uparrow} \quad \Rightarrow \quad \boxed{\forall x.\ P \wedge \underline{\forall x.\ Q}}^{\uparrow} \tag{7}$$

$$\boxed{P \wedge \underline{Q}}^{\uparrow} \leftrightarrow \boxed{P \wedge \underline{R}}^{\uparrow} \quad \Rightarrow \quad Q \leftrightarrow R \tag{8}$$

Wave rules such as (6)–(8) that are stated in terms of logical connectives only are called *propositional* wave rules.

For conjecture (3), based on wave rules (4)–(8), Clam suggests one-step structural induction on the list i[4]. The annotated step case is then:

$$\begin{array}{l} subset(t, j) \leftrightarrow \forall x.\ member(x, t) \rightarrow member(x, j) \\ \vdash \\ subset(\boxed{[h|\underline{t}]}^{\uparrow}, j) \leftrightarrow \forall x.\ member(x, \boxed{[h|\underline{t}]}^{\uparrow}) \rightarrow member(x, j) \end{array}$$

Rippling with wave rules (4) and (5) on the left and right, respectively, gives us:

$$\begin{array}{l} \boxed{member(h, j) \wedge \underline{subset(t, j)}}^{\uparrow} \leftrightarrow \\ \qquad \forall x.\ \boxed{x = h \vee \underline{member(x, t)}}^{\uparrow} \rightarrow member(x, j) \end{array}$$

Rippling with wave rule (6) on the right results in:

$$\begin{array}{l} \boxed{member(h, j) \wedge \underline{subset(t, j)}}^{\uparrow} \leftrightarrow \\ \qquad \forall x.\ \boxed{x = h \rightarrow member(x, j) \wedge \underline{member(x, t) \rightarrow member(x, j)}}^{\uparrow} \end{array}$$

[4]Clam uses a technique called recursion analysis [Bundy *et al* 89] to choose an induction schema. Explaining recursion analysis is beyond the scope of this paper.

Rippling with wave rule (7) on the right gives us:

$$\boxed{member(h, j) \wedge \underline{subset(t, j)}}^{\uparrow} \leftrightarrow$$
$$\boxed{\forall x.\ x = h \rightarrow member(x, j) \wedge \underline{\forall x.\ member(x, t) \rightarrow member(x, j)}}^{\uparrow}$$

Now, we cannot continue rippling because none of the wave rules applies, but we cannot yet exploit the induction hypothesis either. We say that the rippling is *blocked*. We can unblock the rippling by simplifying the wave front on the right-hand side, i.e., by rewriting $\forall x.\ x = h \rightarrow member(x, j)$ to $member(h, j)$:

$$\boxed{member(h, j) \wedge \underline{subset(t, j)}}^{\uparrow} \leftrightarrow$$
$$\boxed{member(h, j) \wedge \underline{\forall x.\ member(x, t) \rightarrow member(x, j)}}^{\uparrow}$$

Wave rule (8) applies and yields:

$$subset(t, j) \leftrightarrow \forall x.\ member(x, t) \rightarrow member(x, j)$$

We strong fertilize to complete the step case. The base case is:

$$\vdash subset([\,], j) \leftrightarrow \forall x.\ member(x, [\,]) \rightarrow member(x, j)$$

Symbolic evaluation of $subset([\,], j)$ and $member(x, [\,])$ gives us:

$$\vdash true \leftrightarrow \forall x.\ false \rightarrow member(x, j)$$

which further simplifies to the tautology:

$$\vdash true$$

Our proof plan is thus complete. It is identical to the proof plan that Clam produces automatically, except that Clam does the base case before the step case.

In the following section, we will show how the planning of verification proofs carries over to the synthesis of logic programs.

6 Synthesis

Verification can be extended to synthesis by introducing middle-out reasoning in the proof planning. Middle-out reasoning involves representing object-level entities with meta-level variables, thus enabling the proof planning to continue even though the identity of the object-level entity is unknown. We will represent the body of the program to be synthesized with a meta-level variable. One

might expect that middle-out reasoning would significantly increase the amount of search in planning, but we will show that this is not case, due to the tight control that rippling provide.

If we inspect the planning of Section 5 to determine which steps depend directly on the definition of the program, we see that there are only two: The application of wave rule (4), since the rule was derived from the program, and the symbolic evaluation of $subset([\,], j)$. Not having wave rule (4) means that, in the step case, the rippling would be blocked after the application of wave rules (5)–(7). It is precisely the use of middle-out reasoning which will allow us to continue planning even though we do not have wave rule (4).

We begin our synthesis with the same conjecture, wave rules (5)–(8), and with a program whose body is undefined, i.e.,

$$\forall i, j.\ subset(i, j) \leftrightarrow \mathcal{P}(i, j)$$

($\mathcal{P}$ is a second-order meta-level variable representing the program body). As before, we proceed by one-step structural induction on the list i. Because of the duality between induction and recursion, we know what the recursive structure of the body of the program will be: A base case where the list i will be empty, and a step case where the list i consists of a head and a tail and which may contain a recursive call. Thus $\mathcal{P}(i, j)$ can already be partially instantiated such that

$$\begin{array}{lcl} \forall i, j.\ subset(i, j) & \leftrightarrow & i = [\,] \wedge \mathcal{B}(j) \vee \\ & & \exists h, t.\ i = [h|t] \wedge \mathcal{S}(h, t, j, subset(t, j)) \end{array}$$

($\mathcal{B}$ and $\mathcal{S}$ are again second-order meta-level variables). Moreover, if the step case contains a recursive call, there will be a wave rule for *subset* of the form:

$$subset(\boxed{[H|\underline{T}]}^{\uparrow}, J) \Rightarrow \boxed{\mathcal{S}(H, T, J, \underline{subset(T, J)})}^{\uparrow} \tag{9}$$

The rippling proceeds as in Section 5 using wave rule (9) instead of (4). Applying wave rules (5) and (9) yields:

$$\begin{array}{l} \boxed{\mathcal{S}(h, t, j, \underline{subset(t, j)})}^{\uparrow} \leftrightarrow \\ \quad \forall x.\ \boxed{x = h \vee \underline{member(x, t)}}^{\uparrow} \rightarrow member(x, j) \end{array}$$

Applying wave rules (6), (7) and the unblocking step to the right-hand side of the equivalence as before gives:

$$\begin{array}{l} \boxed{\mathcal{S}(h, t, j, \underline{subset(t, j)})}^{\uparrow} \leftrightarrow \\ \quad \boxed{member(h, j) \wedge \underline{\forall x.\ member(x, t) \rightarrow member(x, j)}}^{\uparrow} \end{array}$$

We now apply wave rule (8), which instantiates

$$\mathcal{S}(h, t, j, subset(t, j))$$

with:

$$member(h, j) \wedge \mathcal{S}'(h, t, j, subset(t, j))$$

We obtain the subgoal:

$$\boxed{\mathcal{S}'(h, t, j, \underline{subset(t, j)})}^{\uparrow} \leftrightarrow \forall x.\ member(x, t) \rightarrow member(x, j)$$

Finally, strong fertilization, which is now applicable, matches the conclusion with the induction hypothesis, which was

$$subset(t, j) \leftrightarrow \forall x.\ member(x, t) \rightarrow member(x, j)$$

thus instantiating $\mathcal{S}'(h, t, j, subset(t, j))$ with $subset(t, j)$.

To complete the proof plan, we need to deal with the base case:

$$\vdash subset([\,], j) \leftrightarrow \forall x.\ member(x, [\,]) \rightarrow member(x, j)$$

Symbolic evaluation of $subset([\,], j)$ and $member(x, [\,])$ gives us

$$\vdash \mathcal{B}(j) \leftrightarrow \forall x.\ false \rightarrow member(x, j)$$

which simplifies to:

$$\vdash \mathcal{B}(j) \leftrightarrow true$$

This is a tautology if we take $\mathcal{B}(j)$ to be $true$.

The proof plan is complete, and the fully instantiated *subset* program is:

$$\begin{array}{lcl} \forall i, j.\ subset(i, j) & \leftrightarrow & i = [\,] \wedge true \vee \\ & & \exists h, t.\ i = [h|t] \wedge member(h, j) \wedge subset(t, j) \end{array}$$

To summarize the synthesis process, we can say that synthesis equals planning verification proofs using middle-out reasoning. Whether we are doing verification or synthesis, the schema of the proof plan is the same:

1. Choosing an induction schema
2. Base case(s): Symbolic evaluation and tautology checking
3. Step case(s): Rippling and strong fertilization

In the *subset* example, the instantiation of the initial meta-level variable representing the program body met the definition of a pure logic program in Section 3. However, this is not necessarily true of all instantiations in general. We discuss this problem briefly in the following.

Auxiliary Syntheses In the course of planning, a meta-level variable may become instantiated with a program body that violates the definition of pure logic programs of Section 3. Thus, we must check the synthesized program. We need to run an auxiliary synthesis for any part of the program that constitutes a violation; the part itself becomes the specification. We replace any part for which we run an auxiliary synthesis with a call to the auxiliary predicate, and we add the auxiliary predicate to our program.

An example where an auxiliary synthesis is necessary is the specification:

$$\forall m, l.\ max(m, l) \leftrightarrow m \in l \wedge (\forall x.\ x \in l \rightarrow x \leq m)$$

The element m is the maximum element of the list l. The initial synthesized program is:

$$\begin{array}{lll} \forall m, l.\ max(m, l) & \leftrightarrow & l = [] \wedge false \vee \\ & & \exists h, t.\ l = [h|t] \wedge ((m = h \wedge \forall x.\ x \in t \rightarrow x \leq m) \vee \\ & & \qquad (h \leq m \wedge max(m, t))) \end{array}$$

The part $\forall x.\ x \in t \rightarrow x \leq m$ in the program body violates the definition of pure logic program bodies, since it contains a universal quantifier and an implication. We therefore run the auxiliary synthesis:

$$\forall m, l.\ aux(m, l) \leftrightarrow (\forall x.\ x \in l \rightarrow x \leq m)$$

The auxiliary specification states that m is greater than any element of the list l. Unlike the original max specification, however, m does not have to be an element of l. The final program with the auxiliary predicate is:

$$\begin{array}{ll} \forall m, l. & max(m, l) \leftrightarrow \\ & l = [\,] \wedge false \vee \\ & \exists h, t.\ l = [h|t] \wedge ((m = h \wedge aux(m, t)) \vee (h \leq m \wedge max(m, t))) \\ \forall m, l. & aux(m, l) \leftrightarrow \\ & l = [\,] \wedge true \vee \\ & \exists h, t.\ l = [h|t] \wedge h \leq m \wedge aux(m, t) \end{array}$$

7 Summary and Future Work

We have shown how pure logic programs can be synthesized by using middle-out reasoning in the planning of verification proofs. The approach provides a basis for the automatic synthesis of partially correct and complete programs from specifications in sorted, first-order predicate logic. The only synthesis step that lies outside of the proof planning proper is the syntactic check whether the instantiation of the body of the program is acceptable as a pure logic program.

The current methods of the proof planner Clam are a solid foundation to start with. A version of Clam which works with sorted first-order predicate logic

with equality (the original Clam was written for a variant of Martin-Löf type theory) is able to verify the *subset* and *max* programs in Sections 5 and 6. The main change to Clam to enable the corresponding syntheses is the extension of middle-out reasoning.

There are other extensions to Clam which are needed to cope with problems that arise in synthesis proofs. One problem is posed by nested quantifiers in the body of the specification. This occurs, for example, in the proof planning for:

$$\forall k.\ no\_duplicates(k) \leftrightarrow (\forall l, m.\ append(l, m) = k \rightarrow (\forall x.\ x \in l \rightarrow x \notin m))$$

The annotated induction conclusion is:

$$no\_duplicates(\boxed{[h|\underline{t}]}^{\uparrow}) \leftrightarrow$$
$$(\forall l, m.\ append(l, m) = \boxed{[h|\underline{t}]}^{\uparrow} \rightarrow (\forall x.\ x \in l \rightarrow x \notin m))$$

Here, the rippling on the right-hand side of the equivalence is immediately blocked. The wave rule we would like to apply is

$$\boxed{[H1|\underline{T1}]}^{\uparrow} = \boxed{[H2|\underline{T2}]}^{\uparrow} \Rightarrow \boxed{H1 = H2 \wedge \underline{T1 = T2}}^{\uparrow}$$

but in order to do so we need to unfold the *append* first. This is obstructed by the universal quantification of l. Clam's current unblocking techniques will have to be extended to deal with such cases.

Another difficult problem arises, for example, in the proof planning for:

$$\forall x.\ even(x) \leftrightarrow (\exists y.\ y \cdot s(s(0)) = x)$$

Here, the problem is that Clam is unable to suggest the appropriate type of induction, namely two-step induction on x. Clam's technique to choose an induction schema, i.e., recursion analysis [Bundy *et al* 89], works well for conjectures containing universal quantifiers only, but breaks down in the presence of existential quantifiers. The alternative to recursion analysis is again to use middle-out reasoning, this time to postpone the choice of induction schema until the rippling in the step case determines the type of induction.

Finally, in Sections 5 and 6, we assumed that Clam had available the lemmas necessary to derive the propositional wave rules (6)–(8). Given the large number of conceivable propositional wave rules, Clam should be able to generate the lemmas and wave rules on demand.

References

[Bates & Constable 85] Joseph L. Bates and Robert L. Constable. Proofs as programs. *ACM Transactions on Programming Languages and Systems*, 7(1):113–136, January 1985.

[Bibel 80] W. Bibel. Syntax-directed, semantics-supported program synthesis. *Artificial Intelligence*, 14:243–261, 1980.

[Biundo 88] S. Biundo. Automated synthesis of recursive algorithms as a theorem proving tool. In Y. Kodratoff, editor, *Eighth European Conference on Artificial Intelligence*, pages 553–8. Pitman, 1988.

[Boyer & Moore 88] R.S. Boyer and J.S. Moore. *A Computational Logic Handbook*. Academic Press, 1988. Perspectives in Computing, Vol 23.

[Bundy 88] A. 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* 89] A. Bundy, F. van Harmelen, J. Hesketh, A. Smaill, and A. Stevens. A rational reconstruction and extension of recursion analysis. In N.S. Sridharan, editor, *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, pages 359–365. Morgan Kaufmann, 1989. Also available from Edinburgh as DAI Research Paper 419.

[Bundy *et al* 90a] 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, 1990. Also available from Edinburgh as DAI Research Paper 448.

[Bundy *et al* 90b] 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* 90c] 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.

[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.

[Deville 90] Y. Deville. *Logic Programming. Systematic Program Development*. International Series in Logic Programming. Addision-Wesley, 1990.

[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.

[Hill & Lloyd 91] P. Hill and J. Lloyd. The Gödel Report. Technical Report TR-91-02, Department of Computer Science, University of Bristol, March 1991. Revised in September 1991.

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

[Howard 80] W.A. Howard. The formulae-as-types notion of construction. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry; Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490. Academic Press, 1980.

[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.

[Wiggins 92] G. A. Wiggins. Synthesis and transformation of logic programs in the Whelk proof development system. In K. R. Apt, editor, *Proceedings of JICSLP-92*, 1992.

[Wiggins *et al* 91] G. A. Wiggins, A. Bundy, H. C. 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.