MAX-PLANCK-INSTITUT FÜR INFORMATIK

Middle-Out Reasoning for Logic Program Synthesis

Ina Kraan
David Basin
Alan Bundy

MPI–I–93–214 January 1993

INFORMATIK

Im Stadtwald
D 66123 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 at ICLP 93 in Budapest Hungary, June 21–24, 1993.

Acknowledgements

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

Abstract

Logic programs can be synthesized as a by-product of the planning of their verification proofs. This is achieved by using higher-order variables at the proof planning level, which become instantiated in the course of planning. We illustrate two uses of such variables in proof planning for program synthesis, one for synthesis proper and one for the selection of induction schemes. We demonstrate that the use of these variables can be restricted naturally in such a way that terms containing them form a tractable extension of first-order terms.

Keywords

Logic program synthesis, proof planning, automated theorem proving

1 Introduction

The framework of the work presented here is the automatic synthesis of logic programs via *middle-out reasoning* in *explicit proof plans* [Bundy 88, Bundy *et al* 90a]. We synthesize *pure logic programs* from specifications in sorted, first-order theories. The approach encompasses two levels of reasoning: An object level, which is sorted, first-order predicate logic with equality, and a meta-level, at which object-level proofs are reasoned about explicitly. At the object level, we prove that specification and program are logically equivalent. At the meta-level, we construct a plan for the object-level proof. We represent the body of the program to be synthesized with a second-order meta-variable, which becomes instantiated to a program during planning.

Normally, the first and crucial step within proof planning is the selection of an induction scheme. It is crucial for synthesis in particular because the type of induction will correspond to the type of recursion of the synthesized program. By using meta-variables to represent the constructor functions applied to induction variables in the step case, we can plan the step case without committing to a particular induction scheme. We thus postpone the selection of an induction scheme to a later stage of the proof, when more information has been gained from the proof planning.

The contributions of this paper are the following: First, it develops the idea that middle-out reasoning, i.e., the use of the variables at the planning level, can be used to synthesize programs. Middle-out reasoning for program synthesis was first proposed in [Kraan *et al* 93]. Here, approach is improved by limiting the higher-order unification to a decidable case where most general unifiers exist. Second, we demonstrate a new application of middle-out reasoning, namely, the selection of induction schemes.

The ideas presented in this paper have been implemented as an application of the proof planner *CIᴬM* [Bundy *et al* 90c]. The system has been used to synthesize the examples in this paper.

Section 2 of this paper discusses related work in program synthesis and selecting induction schemes for synthesis proofs. Section 3 contains a definition of pure logic programs. Sections 4 and 5 provide brief introductions to higher-order patterns and to proof planning, respectively. Section 6 shows how middle-out reasoning can be used for synthesis proper, and Section 7 how middle-out reasoning can be used to select induction schemes for synthesis proofs. Section 8 contains conclusions and suggestions for future work.

2 Background and Related Work

2.1 Logic Program Synthesis

The approach we take to program synthesis is similar to *proofs-as-programs*. *Proofs-as-programs* evolved from ideas in constructive logic, e.g., the *Curry-Howard isomorphism* [Howard 80], where a proposition is identified with the type of terms in the λ-calculus representing 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 constructs such a member. Since terms in the λ-calculus can be evaluated, proofs give rise to functional programs.

Adapting proofs-as-programs to logic program synthesis is not straightforward. The proofs-as-programs approach synthesizes total functions, whereas logic programs are partial and multivalued [Bundy *et al* 90b]. Logic programs 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 in Prolog-style proofs. He extends standard Prolog goals to implicational 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. He also extends standard Prolog SLD-resolution to the rules of *definite clause inference*, *simplification* and *restricted structural induction*. Each of these rules is associated with a program construction rule. Given a specification, extended Prolog execution will return a program to compute $\vec{y}$ in terms of $\vec{x}$. However, the program will only be correct when the variables in $\vec{x}$ are ground and the variables in $\vec{y}$ are unbound. Also, it will return only 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. Specifications of logic programs are thus of the form:

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

The programs resulting from proofs of such specifications are still higher-order and functional and thus 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, however, is proving the correctness of the rules [Wiggins 92].

In our approach, at the object level, we are not proving $\forall\exists$ specifications, but verification conjectures of the form[1]

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

in a first-order theory. However, at the planning level, we represent the program body with a meta-variable. Thus, what we do is closely related to proving the higher-order conjecture

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

where P represents a pure logic program. The universal quantification of the arguments allows the program to be truly relational, i.e., to run in any mode. Since we are proving a verification conjecture, the synthesized programs are partially correct and complete, if the execution of the proof plan succeeds.

2.2 Selecting Induction Schemes for Synthesis Proofs

Determining the appropriate type of induction for a given conjecture is a difficult task. The most widely used technique is *recursion analysis* [Boyer & Moore 79,

[1]Here, and in the following, we omit sort information to avoid notational clutter.

Bundy *et al* 89]. Recursion analysis selects an appropriate induction scheme by examining the recursion schemes of the functions and relations in the conjecture to determine which of the variables available for induction occur in the recursive positions of these functions and relations.

Recursion analysis works poorly in the presence of existential quantifiers, which are inherent in $\forall\exists$ specifications of functions. This is because the appropriate induction scheme usually depends on the recursion scheme of the witnessing function—which is precisely what we want to synthesize and therefore do not know. Using an inappropriate induction scheme may make it difficult to find a proof or may lead to an unintuitive or inefficient program.

A simple example where recursion analysis breaks down is the specification of a quotient and remainder function:

$$\forall x, y.\ \exists q, r.\ \ x \neq 0\ \rightarrow\ q \times x + r = y \wedge r < x$$

Only x and y are available as induction variables, and, given the standard definitions of addition and multiplication, recursion analysis cannot find the appropriate induction, which is induction on y from $y - x$ to y.

Recursion analysis works better for the conjectures in our approach than for $\forall\exists$ specifications. This is because the relations are universally quantified over all arguments. We thus have more variables to choose from as induction variables and hence stand a better chance of success. The conjecture for quotient and remainder in our approach is

$$\forall x, y, q, r.\, quotient\_remainder(x, y, q, r)\ \leftrightarrow\ q \times x + r = y\ \wedge\ r < x$$

where, for synthesis, the relation *quotient_remainder* would remain unspecified. Since x, y, q and r are all universally quantified, they are all candidate induction variables. For this conjecture, recursion analysis will indeed suggest an appropriate induction, namely one-step structural induction on q.

However, recursion analysis is always limited to a type of induction which is based on the recursion schemes present in the specification. Even for relational conjectures, the recursion of the program may not be among them. An example of this is the specification:

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

The natural recursion scheme for the program would be two-step recursion, which is not suggested by the standard definition of multiplication.

[Hutter 92] suggests a technique which can be used to select induction schemes for $\forall\exists$ formulae. Hutter recognizes the close relationship between the instantiation of the existential variables, the induction variables and the type of induction. Instead of selecting the induction variable and type of induction and then finding the instantiation of the existential variable, he picks an induction variable and an instantiation of an existential variable, leaving the type of induction open until a later stage of the proof. In doing so, he is not limited to induction schemes suggested by the specification, but can also find ones suggested by the proof.

Our approach is related to that suggested in [Hutter 92]. We also postpone a decision on the induction scheme until a later stage of the proof. However, where Hutter commits to an induction variable and an instantiation of an existential variable, we leave that choice open as well. Moreover, whereas Hutter's approach is geared towards $\forall\exists$ formulae, ours is generally applicable.

3 Pure Logic Programs

Our notion of pure logic programs is related to pure logic programs in [Bundy *et al* 90b] and similar to logic descriptions [Deville 90] and completions of normal programs [Lloyd 87]. Pure logic programs are a suitable intermediate representation on the borderline between non-executable specifications and executable programs; pure logic programs are a subset of first-order predicate logic and can thus be reasoned about within that framework. Their syntax, however, is sufficiently restricted that, even if they are not meant to be directly executed, they are straightforward to translate into executable programs in languages such as Prolog or Gödel.

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. There can be no more than one definition per predicate symbol. 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[2].
- A call to a previously defined predicate (including the predicate being defined) is a pure logic program body.
- If P and Q are pure logic program bodies, then

  $$P \wedge Q \qquad P \vee Q \qquad \exists x.\ P$$

  are pure logic program bodies.

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

An example of a pure logic program is:

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

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, this becomes:

```
member(X, [X|_]).
member(X, [_|T]) :- member(X, T).

subset([], _).
subset([H|T], J) :- member(H, J), subset(T, J).
```

[2]For the purpose of this paper, the set consists of equality (=) and inequality ($\neq$).

4 Higher-Order Patterns

Higher-order terms are normally difficult to deal with, since unification is undecidable and there is no most general unifier. When using higher-order terms, one either accepts this and uses, for instance, the pre-unification procedure of [Huet 75], or one restricts oneself to a subset of higher-order terms which is tractable.

Higher-order patterns form a tractable subset of higher-order terms. They are expressions whose free variables have no arguments other than bound variables. The class of higher-order patterns was first investigated by [Miller 90], and followed up among others by [Nipkow 91]. Formally, following [Nipkow 91], a term t in β-normal form is called a *(higher-order) pattern* if every free occurrence of a variable F is in a subterm $F(u_1, \ldots, u_n)$ of t such that each u_i is η-equivalent to a bound variable and the bound variables are distinct.

Higher-order patterns are akin to first-order terms in that unification is decidable and there exists a most general unifier of unifiable terms. Also, the unification of two higher-order patterns is again a higher-order pattern. Both [Miller 90] and [Nipkow 91] give unification algorithms. [Qian 92] shows that the unification of higher-order patterns can be done in linear time. Higher-order patterns are thus as tractable as first-order terms.

The higher-order meta-variables we use in the proof planning can be restricted by letting them represent functions or predicates applied to distinct bound variables only (see Sections 6 and 7). Thus the terms containing them are higher-order patterns. This restriction is natural for the applications suggested here. For synthesis proper, we are creating programs that represent relations and that are therefore developed in the context of a collection of universally bound variables. The distinctness requirement is already present in the definition of pure logic programs. Thus, what we start out with as our program is already a higher-order pattern. Furthermore, any step that further instantiates the higher-order pattern does so via unification with another higher-order pattern. For middle-out induction, we use meta-variables to represent the constructor function applied to the induction variable. Since the variable on which we induce must be universally bound to begin with, the expressions we obtain are again higher-order patterns. The instantiation of the meta-variables occurs via the application of rewrite rules, which are themselves also inherently higher-order patterns.

5 Proof Planning

To avoid the built-in heuristics common in theorem provers, which are often inflexible and difficult to understand, [Bundy 88] suggests using a meta-logic to reason about and to plan proofs. 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 such that, 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 will be true of the resulting goal formulae. The proof planner *CIAM* [Bundy *et al* 90c] incorporates these ideas.

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-variables. This enables proof planning to proceed even though an object-level entity is not fully specified and thus allows a decision about its identity to be postponed.

The proof planner $CI\!AM$ is geared towards proving theorems by induction. Its central method is *rippling* [Bundy *et al* 91], used in the step case of inductive proofs. Rippling keeps track of the differences between the induction hypothesis and the induction conclusion and reduces them by applying special rewrite rules called *wave rules* until the induction hypothesis can be exploited. To this end, rippling uses annotations on the induction conclusion and on the wave rules. Schematically, the step case of a (constructor-style) induction is annotated as follows:

$$p(x) \vdash p(\boxed{c(\underline{x})})$$

Non-underlined parts in boxes (called *wave fronts*) do not appear in the induction hypothesis and thus need to be eliminated before we can appeal to the induction hypothesis. Underlined parts in boxes (called *wave holes*) and remaining parts of the conclusion form a copy of the induction hypothesis. Wave rules are annotated similarly:

$$p(\boxed{c(\underline{X})}) \Rightarrow \boxed{c'(\underline{p(X)})}$$

They are applied only if the rule and a subexpression of the conclusion match, including annotations. The annotations on the wave rule ensure that it will move wave fronts outwards. Once all wave fronts have been eliminated or moved to surround the conclusion, the induction hypothesis can be exploited.

Other methods are induction, symbolic evaluation, simplification and fertilization. Their use will become apparent in the following sections.

6 Middle-Out Reasoning for Synthesis

We synthesize programs via middle-out reasoning by planning verification proofs for programs with initially unspecified bodies. The verification conjectures, which we prove classically, are first-order sentences of the form:

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

Proving 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].

To synthesize a program, we plan the proof of the verification conjecture while representing the body of the program with a second-order meta-variable. During the proof planning, the variable will become instantiated to a program. To illustrate the synthesis process, we will work through the synthesis of the *subset* program from Section (3).

Our conjecture is

$$\forall i, j.\ subset(i, j) \quad \leftrightarrow \quad (\forall x.\ member(x, i) \rightarrow member(x, j)) \qquad (1)$$

where *member* is defined as:

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

and the unspecified *subset* program is represented as:

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

$\mathcal{P}$ is the variable representing the program body.

The definition of *member* gives rise to the following wave rule[3]:

$$member(X, \boxed{[H|\underline{T}]}) \quad \Rightarrow \quad \boxed{X = H \vee \underline{member(X,T)}} \tag{2}$$

We also need the following wave rules derived from lemmas:

$$\boxed{P \vee \underline{Q}} \rightarrow R \quad \Rightarrow \quad \boxed{P \rightarrow R \wedge \underline{Q \rightarrow R}} \tag{3}$$

$$\forall x.\ \boxed{P \wedge \underline{Q}} \quad \Rightarrow \quad \boxed{\forall x.\ P \wedge \underline{\forall x.\ Q}} \tag{4}$$

For simplicity, we will rely on recursion analysis to suggest an appropriate induction. For conjecture (1), based on wave rules (2)–(4), recursion analysis suggests one-step structural induction on i. The annotated step case is then:

$$subset(t,j) \leftrightarrow (\forall x.\ member(x,t) \rightarrow member(x,j)) \tag{5}$$

$$\vdash$$

$$subset(\boxed{[h|\underline{t}]}, j) \leftrightarrow (\forall x.\ member(x, \boxed{[h|\underline{t}]}) \rightarrow member(x,j)) \tag{6}$$

The duality between induction and recursion determines the recursive structure of the body of the program: There will be a base case where i is empty, and a step case where i consists of a head and a tail and which may contain a recursive call. There are several ways to represent this. Previously, in [Kraan *et al* 93] we suggested instantiating $\mathcal{P}(i,j)$ in the following way:

$$\begin{aligned} \forall i,j.\ subset(i,j) \quad \leftrightarrow \quad & i = [\,] \wedge \mathcal{B}(j) \vee \\ & \exists h,t.\ i = [h|t] \wedge \mathcal{S}(h,t,j,subset(t,j)) \end{aligned} \tag{7}$$

$\mathcal{B}$ and $\mathcal{S}$ are again meta-variables. This particular representation for the program was suggested because it gives rise to a wave rule for the *subset* program of the form:

$$subset(\boxed{[H|\underline{T}]}, J) \Rightarrow \boxed{\mathcal{S}(H,T,J,\underline{subset(T,J)})} \tag{8}$$

However, with this representation, neither program nor wave rule are higher-order patterns, since the last argument of $\mathcal{S}$ is not a bound variable.

Here, we show that it is possible to obtain a synthesis proof plan without wave rule (8) and with a simpler representation of the structure of the program. Instead of (7), we use

$$\begin{aligned} \forall i,j.\ subset(i,j) \quad \leftrightarrow \quad & i = [\,] \wedge \mathcal{B}(j) \vee \\ & \exists h,t.\ i = [h|t] \wedge \mathcal{S}(h,t,j) \end{aligned} \tag{9}$$

which no longer contains an explicit recursive call. This representation does not give rise to a wave rule.

We now proceed with the step case and ripple the induction conclusion (6). Applying wave rule (2) on the right gives us:

$$subset(\boxed{[h|\underline{t}]}, j) \leftrightarrow (\forall x.\ \boxed{x = h \vee \underline{member(x,t)}} \rightarrow member(x,j))$$

Applying wave rule (3) on the right results in:

$$\begin{aligned} & subset(\boxed{[h|\underline{t}]}, j) \leftrightarrow \\ & \quad (\forall x.\ \boxed{x = h \rightarrow member(x,j) \wedge \underline{member(x,t) \rightarrow member(x,j)}}) \end{aligned}$$

[3] Wave rules are generated automatically from definitions and lemmas provided to CL^AM.

Applying wave rule (4) on the right leads to:

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

Now, none of the wave rules apply. However, the expression in the wave hole on the right-hand side, i.e., $\forall x.\ member(x, t) \rightarrow member(x, j)$, matches the right-hand side of the induction hypothesis (5). Thus, we can exploit the induction hypothesis itself as a rewrite rule. This is called *weak fertilization.* We obtain the conclusion:

$$subset(\boxed{[h|\underline{t}]}, j) \leftrightarrow \boxed{\forall x.\ x = h \rightarrow member(x, j) \wedge \underline{subset(t, j)}} \tag{10}$$

Once we have appealed to the induction hypothesis, we are left with a conclusion that corresponds to the step case of the program we are synthesizing. Thus, from the perspective of synthesis, we are done. Remember, however, that we are synthesizing a program and planning its verification proof at the same time. In order to complete the verification, we must show that (10) follows from the program definition. This step will further instantiate the body of the program.

To appeal to the program definition (9), we instantiate it appropriately

$$\begin{aligned} subset([h|t], j) \quad \leftrightarrow \quad & [h|t] = [\,] \wedge \mathcal{B}(j) \vee \\ & \exists h', t'.\ [h|t] = [h'|t'] \wedge \mathcal{S}(h', t', j) \end{aligned}$$

and simplify it to:

$$subset([h|t], j) \leftrightarrow \mathcal{S}(h, t, j) \tag{11}$$

Unifying (10) and (11) instantiates S with the expression $\lambda u, v, w.\ (\forall x.\ x = h \rightarrow member(x, w) \wedge subset(v, w))$.

We thus obtain the partially instantiated program:

$$\begin{aligned} &\forall i, j.\ subset(i, j) \leftrightarrow \\ &\quad i = [\,] \wedge \mathcal{B}(j) \vee \\ &\quad \exists h, t.\ i = [h|t] \wedge \forall x.\ x = h \rightarrow member(x, j) \wedge subset(t, j) \end{aligned}$$

This instantiation is not yet acceptable. In Section 3, we imposed syntactic restrictions on pure logic programs. The program we have synthesized so far does not quite comply with them, since it contains a universal quantifier. In this case, the offending expression $\forall x.\ x = h \rightarrow member(x, j)$ can easily be simplified to $member(h, j)$. In other cases, an auxiliary synthesis may be required. For a brief discussion of auxiliary syntheses, see [Kraan *et al* 93].

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 $member(x, [\,])$ gives us

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

which simplifies to:

$$\vdash subset([\,], j) \leftrightarrow true$$

After simplification, we are left with a conclusion that corresponds to the base case of the program. We appeal to the program definition as before in the step case.

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

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

The example of the *subset* synthesis closely follows the general schema of synthesis proof plans, which can be summarized as follows:

1. Apply induction
2. Step case(s): Apply rippling and weak fertilization, appeal to program
3. Base case(s): Apply symbolic evaluation and simplification, appeal to program
4. Run auxiliary syntheses if necessary

7 Middle-Out Reasoning for Induction

The fundamental problem in selecting induction schemes for synthesis is that the appropriate induction scheme is closely related to the recursion scheme of the program—which is precisely part of what we want to synthesize. Yet selecting an induction scheme is normally the first step of the proof planning, and we therefore have no information beyond the specification itself. Using middle-out reasoning we can escape this problem: We use meta-variables to represent the constructor functions applied to potential induction variables. Thus, we can proceed with rippling in the step case without having to commit to any particular induction scheme. Once the rippling has been completed and the induction hypothesis has been appealed to, the meta-variables will be fully instantiated. We can either check whether the instantiations of the meta-variables correspond to one of a set of given induction schemes, or we can prove that the ordering defined by the instantiations is well-founded. In the current implementation, we take the first approach.

To illustrate this, we work through an example similar to that in Section 2

$$\forall x.\ even(x) \leftrightarrow (\exists y.\ double(y) = x)$$

where *double* is defined as follows:

$$\begin{aligned}double(0) \quad &= \quad 0 \\ \forall x.\ double(s(x)) \quad &= \quad s(s(double(x)))\end{aligned}$$

The wave rules for *double* and equality are

$$double(\boxed{s(\underline{U})}) \quad \Rightarrow \quad \boxed{s(s(\underline{double(U)}))} \tag{12}$$

$$\boxed{s(\underline{U})} = \boxed{s(\underline{V})} \quad \Rightarrow \quad U = V \tag{13}$$

The step case is

$$\begin{aligned}&even(x) \leftrightarrow (\exists y.\ double(y) = x) \\ &\vdash \\ &even(\boxed{\mathcal{C}(\underline{x})}) \leftrightarrow (\exists y.\ double(\boxed{y}) = \boxed{\mathcal{C}(\underline{x})})\end{aligned}$$

where $\mathcal{C}$ is the meta-variable standing for the constructor function applied to the induction variable. The dashed wave front around the variable y indicates that it can be involved in the rippling. Note that, because we have no knowledge of the induction scheme until we complete the step case, we cannot determine the structure of the program yet.

We can apply wave rule (12) to the induction conclusion. This instantiates the existentially quantified variable y with $s(y')$, where y' is a new existentially quantified variable. Rippling with existentially quantified variables is explained in detail in [Bundy *et al* 91]. Applying (12) yields the conclusion:

$$even(\boxed{\mathcal{C}(\underline{x})}) \leftrightarrow (\exists y'.\ \boxed{s(s(\underline{double(\boxed{y'})}))} = \boxed{\mathcal{C}(\underline{x})})$$

Applying wave rule (13) twice then results in:

$$even(\boxed{s(s(\mathcal{C}'(\underline{x})))}) \leftrightarrow (\exists y'.\ double(\boxed{y'}) = \boxed{\mathcal{C}'(\underline{x})})$$

The applications of wave rule (13) partially instantiate $\mathcal{C}$ to $\lambda u.s(s(\mathcal{C}'(u)))$. We can now weak fertilize, i.e., apply the induction hypothesis as a rewrite rule. This leaves us with the conclusion

$$even(\boxed{s(s(\underline{x}))}) \leftrightarrow even(x)$$

which corresponds to the step case of the program. This also instantiates $\mathcal{C}'$ to $\lambda u.u$ and $\mathcal{C}$ to $\lambda u.s(s(u))$. Thus, the induction scheme is two-step induction, and the program structure is:

$$\begin{aligned}\forall x.\ even(x) \quad \leftrightarrow \quad & x = 0 \wedge \mathcal{B}_1 \vee \\ & x = s(0) \wedge \mathcal{B}_2 \vee \\ & \exists y.\ x = s(s(y)) \wedge \mathcal{S}(y)\end{aligned}$$

We can now finish the step case and the bases cases following the general schema in Section 6. The final *even* program is:

$$\begin{aligned}\forall x.\ even(x) \quad \leftrightarrow \quad & x = 0 \wedge true \vee \\ & x = s(0) \wedge false \vee \\ & \exists y.\ x = s(s(y)) \wedge even(y)\end{aligned}$$

The *even* example was chosen for its simplicity and does not show the full power of middle-out reasoning for induction. The technique is particularly powerful in examples involving many candidate variables to select from.

8 Conclusions and Future Work

In this paper, we have shown how pure logic programs can be synthesized with the help of middle-out reasoning in the planning of verification proofs. First, a meta-variable is used to represent the body of the program to be synthesized. Second, meta-variables are used to represent the constructor functions applied to candidate induction variables in the step case of the induction. This facilitates finding appropriate induction schemes. By restricting the meta-variables such that we need only work with higher-order patterns, we obtain a formal system which is a tractable extension of first-order terms. The approach provides a sound and computationally viable basis for the automatic synthesis of partially correct and complete programs.

The ideas presented in this paper have been implemented as an application of the proof planner *CIAM*. We have adapted *CIAM* to first-order predicate logic, added higher-order pattern unification and developed synthesis methods. The system has synthesized a number of examples, including those presented here.

To scale the approach up to larger and more difficult problems, there are issues that need to be addressed. Difficult issues, often caused by the use of meta-variables, are that rippling may not terminate or may stop before it the induction hypothesis can be exploited. To cope with non-termination, additional heuristic control is needed. The need for additional control arises from the fact that working with meta-variables means working with less knowledge, which usually leads to more choices. When rippling stops before the induction hypothesis can be used, the cause is often a missing lemma. In Section 6, for instance, we assumed that we had the lemmas necessary to derive wave rules (3) and (4). In general, we cannot expect to have all necessary lemmas, and should therefore be able to generate them on demand.

Another important issue is leading the proof planning to particular types of programs, e.g., divide-and-conquer algorithms or tail-recursive programs. In this context, it is worth investigating whether mode information could help guide the proof planning.

References

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

[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: Edinburgh DAI Research Paper 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 Edinburgh 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 Edinburgh 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 Edinburgh 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 Edinburgh DAI Research Paper 507.

[Bundy *et al* 91] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. Edinburgh DAI Research Paper 567, 1991. To appear in Artificial Intelligence.

[Deville 90] Y. Deville. *Logic Programming. Systematic Program Development.* International Series in Logic Programming. Addison-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.

[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.

[Huet 75] G. Huet. A unification algorithm for lambda calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[Hutter 92] D. Hutter. Synthesis of induction orderings for existence proofs. 1992. Forthcoming.

[Kraan *et al* 93] I. Kraan, D. Basin, and A. Bundy. Logic program synthesis via proof planning. In K.K. Lau and T. Clement, editors, *Logic Program Synthesis and Transformation*, pages 1–14. Springer-Verlag, 1993. Also Max-Planck-Institut für Informatik Research Paper MPI-I-92-244.

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

[Miller 90] D. Miller. A logic programming language with lambda abstraction, function variables and simple unification. Technical Report MS-CIS-90-54, Department of Computer and Information Science, University of Pennsylvania, 1990.

[Nipkow 91] T. Nipkow. Higher-order critical pairs. In *Proc. 6th IEEE Symp. Logic in Computer Science*, pages 342–349, 1991.

[Qian 92] Z. Qian. Unification of higher-order patterns in linear time and space. Technical Report 5/92, FB 3 Informatik, Universität Bremen, 1992.

[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.