Middle-Out Reasoning for Synthesis and Induction

Ina Kraan* David Basin† Alan Bundy‡

July 10, 1995

Abstract

We develop two applications of middle-out reasoning in inductive proofs: Logic program synthesis and the selection of induction schemes. Middle-out reasoning as part of proof planning was first suggested by Bundy *et al* [Bundy *et al* 90a]. Middle-out reasoning uses variables to represent unknown terms and formulae. Unification instantiates the variables in the subsequent planning, while proof planning provides the necessary search control.

Middle-out reasoning is used for synthesis by planning the verification of an unknown logic program: The program body is represented with a meta-variable. The planning results both in an instantiation of the program body and a plan for the verification of that program. If the plan executes successfully, the synthesized program is partially correct and complete.

Middle-out reasoning is also used to select induction schemes. Finding an appropriate induction scheme during synthesis is difficult, because the recursion of the program, which is unknown at the outset, determines the induction in the proof. In middle-out induction, we set up a schematic step case by representing the constructors that are applied to induction variables with meta-variables. Once the step case is complete, the instantiated variables correspond to an induction appropriate to the recursion of the program.

We have implemented these techniques as an extension of the proof planning system *CLAM* [Bundy *et al* 90c], called *Periwinkle*, and synthesized a variety of programs fully automatically.

keywords: Automated theorem proving, proof planning, induction, logic program synthesis, meta-variables, higher-order unification

1 Introduction

We develop techniques based on *proof planning* and *middle-out reasoning* that enable the automatic synthesis of logic programs. Proof planning entails explicit reasoning about how to construct proofs. Middle-out reasoning allows proof planning to progress even though an object being reasoned about is not yet fully known. Middle-out reasoning represents unspecified objects in the proof with variables and instantiates them using unification. Thus we can plan proofs while leaving certain

*Supported by the Swiss National Science Foundation and ARC Project BC/DAAD Grant 438. The work described in this paper was carried out while the first author was at the Department of Artificial Intelligence of the University of Edinburgh.

†Supported by the German Ministry for Research and Technology (BMFT) under grant ITS 9102 and ARC Project BC/DAAD Grant 438. Responsibility for the contents of this publication lies with the authors.

‡Supported by SERC grant GR/J/80702, ESPRIT BRP grant 6810, ESPRIT BRP grant EC-US 019-76094, and ARC Project BC/DAAD Grant 438.

unknown terms or formulae to be filled in at a later stage. In program synthesis based on inductive proofs, there are two things that are unknown: First, most obviously, the program to be synthesized, but second the type of induction used in the proof. This is because the appropriate type of induction depends on the type of recursion of the program to be synthesized.

Middle-out reasoning for program synthesis and induction has been implemented as an extension of the proof planning system *CIAM* [Bundy *et al* 90c]. The extended system, called *Periwinkle*, has been used to synthesize a variety of programs. *Periwinkle* is available on request from the first author.

This paper elaborates and extends [Kraan *et al* 93a, Kraan *et al* 93b]; more detail can be found in [Kraan 94]. The paper is organized as follows: Section 2 is an introduction to proof planning. Sections 3 and 4 present middle-out reasoning for logic program synthesis and the selection of induction schemes. Section 5 presents new methods that proved necessary for synthesis. Section 6 reports on the implementation and on practical results. Section 7 presents ideas for further research, and section 8 draws conclusions.

2 Proof Planning

To use the built-in heuristics common in theorem provers more flexibly, Bundy [Bundy 88] suggests using a meta-logic to reason about and to plan proofs. Proof plans are constructed in the meta-logic by successively applying *methods* to a conjecture until a combination of methods has been found that forms a complete plan. A method is a partial specification of a tactic [Gordon *et al* 79] in the following sense: If a sequent matches the input pattern, and the pre-conditions are met, the tactic is applicable; if the tactic succeeds, the output conditions will be true of the resulting sequents. Explicit proof planning has been implemented in *CIAM* [Bundy *et al* 90c], which constructs plans for inductive proofs in a variant of Martin-Löf type theory [Martin-Löf 79]. The plans are executable in *Oyster* [Bundy *et al* 90c], a sequent-style interactive proof checker.

The advantages of the meta-logic approach are that the search for proofs takes place at the meta-level rather than the object level. The search is less expensive, since methods capture the effects of the corresponding tactics, while avoiding the possibly considerable cost of executing them. More importantly, however, the meta-level representation of the proof can be augmented with additional information on the proof to restrict the search space. The information is passed from method to method, which gives a global rather than a local view of the proof.

Proof planning has concentrated on inductive proofs. The central method for inductive proofs is `ind_strat`, a composite method capturing the structure of such proofs. It is composed of the methods `induction`, `base_case`, and `step_case`. The `induction` method selects a set of induction variables and an induction scheme for a given conjecture, a step crucial to the success of proof planning. The `induction` method uses *recursion analysis* to select an induction. Recursion analysis is a rational reconstruction and extension of the heuristics used in NQTHM to select induction variables and schemes [Boyer & Moore 88, Stevens 88, Bundy *et al* 89]. Recursion analysis prefers induction variables that occur in the recursive positions of the function or relation dominating them (i.e., are smaller in the recursive calls) and which can be rewritten using an axiom or a lemma. It selects a scheme which corresponds to the recursion of the dominating function or relation. In essence, recursion analysis is a look-ahead into the rewriting of the step case.

The `induction` method applies recursion analysis to the input sequent. It succeeds if the analysis suggests a suitable induction scheme and fails otherwise. For the associativity of $+$, for example,

$$\forall x, y, z.\ (x + y) + z = x + (y + z) \tag{1}$$

recursion analysis using

$$\begin{aligned} \forall x, y.\ s(x) + y &= s(x + y) && (2) \\ \forall x, y.\ x = y &\rightarrow s(x) = s(y) && (3) \end{aligned}$$

suggests structural induction on x. Outputs are the base and step cases for the selected induction. Step cases are annotated for the `ripple` method (see below).

The `base_case` method iterates over a symbolic evaluation method `sym_eval` and a simplification and tautology-checking method `elementary`.

In the step case, the main objective is to rewrite the induction conclusion so that the induction hypothesis can be exploited. The `step_case` method applies the `ripple` method to rewrite the induction conclusion and the `fertilize` method to exploit the induction hypothesis. The `ripple` method embodies the *rippling* heuristic [Bundy *et al* 93]. This heuristic uses rewrite rules to eliminate the differences between the induction hypothesis and the induction conclusion so that the induction hypothesis can be exploited. The function symbols that appear in the conclusion, but not in the hypothesis, are called *wave fronts*. Initially, the wave fronts immediately dominate the induction variables. The role of rippling is to move them outwards—just like ripples on a lake—until a perfect reflection of the induction hypothesis is left. The rippling heuristic has been shown to terminate [Bundy *et al* 93, Basin & Walsh]. We represent wave fronts as boxes with holes. The holes are indicated by underlinings. For the step case of the proof of (1), the `induction` method sets up the annotated sequent

$$(x + y) + z = x + (y + z) \quad \vdash \quad (\boxed{s(\underline{x})} + y) + z = \boxed{s(\underline{x})} + (y + z)\ .$$

If we remove the structure in the non-underlined parts of the boxes from the conclusion, we obtain the *skeleton*, i.e., a copy of the induction hypothesis.

Rippling consists of applying annotated rewrite rules called *wave rules*. The annotations on wave rules ensure that applying a wave rule will move at least one wave front up in the term tree of the induction conclusion if the annotations in the rule are compatible with those of the conclusion. The (simplified) schematic format of a wave rule that moves one wave front is

$$F[\boxed{S[\underline{U}]}] :\Rightarrow \boxed{T[\underline{F[U]}]}\ .$$

The effect of applying a wave rule is to move the wave front S on the left-hand side outwards past the F and to turn it into a wave front T on the right-hand side, whose position is higher up the term tree. Note that $:\Rightarrow$ indicates rewriting, not implication. In inductive theorem proving, rippling reasons backwards from the induction conclusion to the induction hypothesis. Thus, rewrite rules may be based on equality, equivalence, or implication from right to left. Wave rules based on (2) and (3), for example, are

$$\begin{aligned} \boxed{s(\underline{M})} + N \quad &:\Rightarrow \quad \boxed{s(\underline{M + N})} \\ \boxed{s(\underline{M})} = \boxed{s(\underline{N})} \quad &:\Rightarrow \quad M = N\ , \end{aligned}$$

where M and N are free variables.

Rippling as presented so far is known as *rippling out.* It is an extension of the *ripple-out* heuristic developed by Aubin [Aubin 76]. For a complete description of all variations of rippling see [Bundy *et al* 93].

The fertilization methods exploit the induction hypothesis. If, after rippling, the wave front surrounds the entire induction conclusion (or has disappeared), the `strong fertilization` method appeals to the induction hypothesis directly. If the wave fronts do not yet surround the entire induction conclusion, the `weak fertilization` method uses the induction hypothesis as a rewrite rule.

Rippling may terminate before the induction hypothesis can be exploited. We then say that the rippling is blocked. There are various techniques to *unblock* the rippling, which modify the conclusion in some way that makes a wave rule or fertilization applicable. A common unblocking step is simplifying a wave front.

3 Middle-Out Synthesis

3.1 Pure Logic Programs

The logic programs we synthesize are the completions of a subset of normal programs (see Lloyd [Lloyd 87]), which we call *pure logic programs.* They are similar to pure logic programs as defined by Bundy *et al* in [Bundy *et al* 90b] and the logic descriptions of Deville [Deville 90]. Formally, we define them as finite sets of pure logic program clauses. A *pure logic program clause* is a closed, typed formula of the form

$$\forall \overline{x}\!:\!\overline{T}.\; A(\overline{x}) \leftrightarrow H$$

where $\overline{x}$ is a vector of distinct variables with types given by $\overline{T}$ (generally left implicit in the following), $A(\overline{x})$ is an atom, called the head of a clause, and H is a *Horn body.* A formula H is a Horn body if, in Backus-Naur notation,

$$H ::= A \mid H_1 \wedge H_2 \mid H_1 \vee H_2 \mid \exists x.H \;\;,$$

where A is an atom whose name is a known relation (such as $=$ or $\neq$) or whose name is among the names of the heads of previously defined clauses (including the one being defined). An example of a pure logic program is

$$\begin{aligned}
\forall x, l.\; member(x,l) \;&\leftrightarrow\; \exists h, t.\; l = h :: t \wedge (x = h \vee member(x,t)) &(4)\\
\forall i, j.\; subset(i,j) \;&\leftrightarrow\; i = nil \;\vee \\
&\qquad \exists h, t.\; i = h :: t \wedge member(h,j) \wedge subset(t,j) \;. &(5)
\end{aligned}$$

We synthesize pure logic programs because they are a suitable intermediate representation between non-executable specifications and executable programs. In particular, the definition of pure logic programs guarantees that

$$\forall \overline{x}\!:\!\overline{T}.\; A(\overline{x}) \leftarrow H$$

corresponds to a set of definite program clauses (see Lloyd [Lloyd 87]).

The class of pure logic programs is very general: It captures the semantics of pure logic programming languages. It also captures the basic recursive structure of algorithms, while avoiding non-logical aspects such as order of execution and non-logical primitives, which are normally specific to the implementation of a logic programming language. This enables us to break down the formidable task of synthesis: First, we synthesize the basic structure of the algorithm, independent of any particular programming language. In a second step, we can translate the pure logic

$$
\boxed{
\begin{array}{ll}
\vdash Prog[Base] \leftrightarrow Spec[Base] & \\
\vdots & \quad symbolic\ evaluation \\
\vdash Formula_1 \leftrightarrow Formula_2 & \\
\vdots & \quad simplification \\
\vdash true & \\
 & \quad tautology
\end{array}
}
$$

Figure 1: Schematic base case in verification

program into a logic programming language of our choice and introduce non-logical primitives as desired. Synthesizing pure logic programs has another advantage: The intended meaning of the program coincides with its logical meaning. Thus, we can reason within the well-understood framework of (many-sorted) first-order logic (with induction) and bring knowledge in theorem-proving to bear.

Using pure logic programs, we can prove partial correctness and completeness[1] by showing that $A(\overline{x})$ as defined by $A(\overline{x}) \leftrightarrow H$ is equivalent to S

$$\forall \overline{x} : \overline{T}.\ A(\overline{x}) \leftrightarrow H \quad \vdash \quad \forall \overline{x} : \overline{T}.\ A(\overline{x}) \leftrightarrow S \ ,$$

where $A(\overline{x})$ and H are as above and S is the specified relation. The proof of equivalence is conducted in an appropriate first-order theory containing axioms and induction principles for recursively defined data-types. For instance, to verify the *subset* program with respect to the specification

$$\forall x.\ member(x, i) \rightarrow member(x, j) \ ,$$

we prove, in a standard theory of lists,

$$\forall i, j.\ subset(i, j) \leftrightarrow (\forall x.\ member(x, i) \rightarrow member(x, j)) \tag{6}$$

from the definitions (4) and (5). Such proofs underlie our synthesis approach.

3.2 Planning Logic Program Verification Proofs

To illustrate verification proof planning, we verify *subset*, i.e., we prove that the logic program (5) verifies conjecture (6). Recursion analysis suggests structural induction on i. The base case is

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

Symbolic evaluation using the base cases of *subset* and *member* yields

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

which simplifies to *true*. The sequence of symbolic evaluation, simplification, and tautology checking is typical of the base cases of verification proofs (see figure 1).

[1] In logic programming, partial correctness means that the computed relation is a subset of the specified relation, completeness that the specified relation is a subset of the computed relation [Clark & Tärnlund 77, Clark 79].

The annotated step case for induction on i is:

$$\begin{aligned} & \mathit{subset}(t,j) \leftrightarrow (\forall x.\ \mathit{member}(x,t) \rightarrow \mathit{member}(x,j)) \vdash \\ & \qquad \mathit{subset}(\boxed{h::\underline{t}},j) \leftrightarrow \Big(\forall x.\ \mathit{member}(x,\boxed{h::\underline{t}}) \rightarrow \mathit{member}(x,j)\Big) \end{aligned} \tag{7}$$

To ripple, we need the wave rules

$$\mathit{subset}(\boxed{H::\underline{T}},L) \;:\Rightarrow\; \boxed{\mathit{member}(H,L) \land \underline{\mathit{subset}(T,L)}} \tag{8}$$

$$\mathit{member}(X,\boxed{H::\underline{T}}) \;:\Rightarrow\; \boxed{X = H \lor \underline{\mathit{member}(X,T)}} \tag{9}$$

$$\boxed{P \lor \underline{Q}} \rightarrow R \;:\Rightarrow\; \boxed{P \rightarrow R \land \underline{Q \rightarrow R}} \tag{10}$$

$$\forall x.\ \boxed{P \land \underline{Q}} \;:\Rightarrow\; \boxed{(\forall x.\ P) \land \underline{\forall x.\ Q}} \tag{11}$$

$$\boxed{P \land \underline{Q}} \leftrightarrow \boxed{P \land \underline{R}} \;:\Rightarrow\; Q \leftrightarrow R \tag{12}$$

of which the first two are based on (4) and (5) and the remaining three on theorems of first-order logic. The latter are called *logical* wave rules (see section 5.1).

The rippling of the induction conclusion in *subset* example consists of applying (8)–(11)

$$\begin{aligned} \mathit{subset}(\boxed{h::\underline{t}},j) &\leftrightarrow \forall x.\ \mathit{member}(x,\boxed{h::\underline{t}}) \rightarrow \mathit{member}(x,j) \\ \boxed{\mathit{member}(h,j) \land \underline{\mathit{subset}(t,j)}} &\leftrightarrow \forall x.\ \mathit{member}(x,\boxed{h::\underline{t}}) \rightarrow \mathit{member}(x,j) \\ \boxed{\mathit{member}(h,j) \land \underline{\mathit{subset}(t,j)}} &\leftrightarrow \forall x.\ \boxed{x = h \lor \underline{\mathit{member}(x,t)}} \rightarrow \mathit{member}(x,j) \\ \boxed{\mathit{member}(h,j) \land \underline{\mathit{subset}(t,j)}} &\leftrightarrow \\ \forall x.\ & \boxed{x = h \rightarrow \mathit{member}(x,j) \land \underline{\mathit{member}(x,t) \rightarrow \mathit{member}(x,j)}} \\ \boxed{\mathit{member}(h,j) \land \underline{\mathit{subset}(t,j)}} &\leftrightarrow \\ & \boxed{(\forall x.\ x = h \rightarrow \mathit{member}(x,j)) \land \underline{\forall x.\ \mathit{member}(x,t) \rightarrow \mathit{member}(x,j)}} \end{aligned}$$

and simplifying the wave front on the right-hand side

$$\begin{aligned} & \boxed{\mathit{member}(h,j) \land \underline{\mathit{subset}(t,j)}} \leftrightarrow \\ & \qquad \boxed{\mathit{member}(h,j) \land \underline{\forall x.\ \mathit{member}(x,t) \rightarrow \mathit{member}(x,j)}} \end{aligned}$$

so that (12) can be applied

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

Strong fertilization completes the proof plan. The sequence of rippling both sides of the equivalence, applying a wave rule that removes the wave fronts, and strong fertilizing is typical of the step cases of verification proofs (see figure 2).

3.3 From Verification to Synthesis

Middle-out reasoning can be used to turn verification proof planning into synthesis by planning the verification of a program while leaving the program unknown. We start with a program whose body is represented with a meta-variable. In the course of planning, the variable becomes instantiated to a program. The planning thus

$$
\begin{array}{l}
Prog[Arg] \leftrightarrow Spec[Arg] \vdash Prog[\boxed{Constr[\underline{Arg}]}] \leftrightarrow Spec[\boxed{Constr[\underline{Arg}]}] \\
\vdots \qquad\qquad\qquad \textit{ripple under equivalence} \\
Prog[Arg] \leftrightarrow Spec[Arg] \vdash \boxed{Front[\underline{Prog[Arg]}]} \leftrightarrow \boxed{Front[\underline{Spec[Arg]}]} \\
\vdots \qquad\qquad\qquad \textit{final ripple} \\
Prog[Arg] \leftrightarrow Spec[Arg] \vdash Prog[Arg] \leftrightarrow Spec[Arg] \\
\qquad\qquad\qquad \textit{strong fertilization}
\end{array}
$$

Figure 2: Schematic step case in verification

results both in an instantiation of the program body and a plan for the verification of that program. If the plan executes successfully, the synthesized program is partially correct and complete.

Representing the program body with a meta-variable entails a loss of information, which affects the proof planning. In the verification proof above, there were a number of steps that depended on the program, but also some that did not. In particular, the symbolic evaluation of *subset* in the base case and the ripple with the *subset* wave rule (8) depend on the program. The main difference between verification and synthesis planning is that in verification, the two types of steps tend to be interleaved. In synthesis, the part of the proof that does not depend on the program is planned first, and any step that does is postponed as long as possible. This is because, in synthesis, any step that depends on the program partially instantiates, i.e., commits the program. Postponing such steps is a least commitment strategy.

To illustrate this, we redo the step case of the verification proof in section 3.2, omitting any steps that depend on the program. This rules out rippling with the *subset* wave rule (8). The rippling progresses as follows, using wave rules (9)–(11) and unblocking.

$$
\begin{array}{lcl}
subset(\boxed{h::\underline{t}}, j) & \leftrightarrow & \forall x.\ \boxed{x = h \vee \underline{member(x,t)}} \rightarrow member(x,j) \\
subset(\boxed{h::\underline{t}}, j) & \leftrightarrow & \forall x.\ \boxed{x = h \rightarrow member(x,j) \wedge \underline{member(x,t) \rightarrow member(x,j)}} \\
subset(\boxed{h::\underline{t}}, j) & \leftrightarrow & \\
 & & \boxed{(\forall x.\ x = h \rightarrow member(x,j)) \wedge \underline{\forall x.\ member(x,t) \rightarrow member(x,j)}} \\
subset(\boxed{h::\underline{t}}, j) & \leftrightarrow & \boxed{member(h,j) \wedge \underline{\forall x.\ member(x,t) \rightarrow member(x,j)}}
\end{array}
$$

The lack of wave rule for *subset* now prevents us from further rippling. However, we can apply weak fertilization: We use the induction hypothesis (7) as a rewrite rule. This yields

$$subset(\boxed{h::\underline{t}}, j) \leftrightarrow \boxed{member(h,j) \wedge \underline{subset(t,j)}}\ .$$

Now we have applied all possible steps that do not depend on the program. In fact, the residual conjecture is precisely the part of the proof that in a verification, would have been proved using the *subset* wave rule (8). It is thus the step clause of our program. By appealing to the as yet uninstantiated program, we commit it to correspond to this residual conclusion. The details of this process are presented below. Figure 3 shows schematically how a typical step case of a synthesis proof

$$
\begin{array}{ll}
Prog[Arg] \leftrightarrow Spec[Arg] \vdash Prog[\boxed{Constr[\underline{Arg}]}] \leftrightarrow Spec[\boxed{Constr[\underline{Arg}]}] & \\
\vdots & \quad \textit{ripple under equivalence} \\
Prog[Arg] \leftrightarrow Spec[Arg] \vdash Prog[\boxed{Constr[\underline{Arg}]}] \leftrightarrow \boxed{Front[\underline{Spec[Arg]}]} & \\
\vdots & \quad \textit{weak fertilization} \\
Prog[Arg] \leftrightarrow Spec[Arg] \vdash Prog[\boxed{Constr[\underline{Arg}]}] \leftrightarrow \boxed{Front[\underline{Prog[Arg]}]} & \\
 & \quad \textit{appeal to program}
\end{array}
$$

Figure 3: Schematic step case in synthesis

$$
\begin{array}{ll}
\vdash Prog[Base] \leftrightarrow Spec[Base] & \\
\vdots & \quad \textit{symbolic evaluation, simplification} \\
\vdash Prog[Base] \leftrightarrow Formula & \\
 & \quad \textit{appeal to program}
\end{array}
$$

Figure 4: Schematic base case in synthesis

progresses: The specification side of the induction conclusion is rippled until weak fertilization is possible, and the proof is completed by appealing to the program.

The base case is similar. In synthesis, only the specification side is symbolically evaluated and simplified, and the residual conclusion corresponds to part of the program (see figure 4).

3.4 An Example Synthesis

We synthesize the program we verified in section 3.2. The subset program is now

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

where $\mathcal{P}$ is the meta-variable that represents the program body. We again do structural induction on i. The type of induction immediately determines the recursive structure of the program. Each induction scheme is associated with the corresponding recursive structure, and the program body is unified (see section 4.2) with this structure

$$
\begin{array}{rcl}
\forall i, j.\ subset(i, j) & \leftrightarrow & i = nil \wedge \mathcal{B}(j) \vee \\
 & & \exists h, t.\ i = h :: t \wedge \mathcal{S}(h, t, j) \ .
\end{array} \tag{13}
$$

The base case for induction on i is

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

Symbolic evaluation using the base case of member and simplification yield

$$\vdash subset(nil, j) \leftrightarrow true \ . \tag{14}$$

We are now left with what will become the base case of the program. By appealing to the as yet uninstantiated program definition, we complete the base case of the proof and at the same time instantiate the base case of the program. This is done by the `synthesis` method. To appeal to the program (13), the `synthesis` method instantiates it appropriately and simplifies it

$$\begin{aligned} subset(nil, j) &\leftrightarrow nil = nil \wedge \mathcal{B}(j) \vee \\ &\quad \exists h', t'.\ nil = h' :: t' \wedge \mathcal{S}(h', t', j) \\ subset(nil, j) &\leftrightarrow \mathcal{B}(j)\ . \end{aligned} \tag{15}$$

The conclusion (14) and the program (15) are unified (see section 4.2), which yields the instantiation $\lambda u.\ true$ for $\mathcal{B}$ and completes the base case. The (normalized) partially instantiated program so far is thus

$$\begin{aligned} \forall i, j.\ subset(i, j) &\leftrightarrow i = nil \wedge true \vee \\ &\quad \exists h, t.\ i = h :: t \wedge \mathcal{S}(h, t, j)\ . \end{aligned} \tag{16}$$

In section 3.3, we showed that the residual conclusion in the step case is

$$subset(\boxed{h :: \underline{t}}, j) \leftrightarrow \boxed{member(h, j) \wedge \underline{subset(t, j)}}\ . \tag{17}$$

We must establish that this follows from the program definition. The `synthesis` method instantiates (16) appropriately and simplifies it

$$\begin{aligned} subset(h :: t, j) &\leftrightarrow h :: t = nil \wedge true \vee \\ &\quad \exists h', t'.\ h :: t = h' :: t' \wedge \mathcal{S}(h', t', j) \\ subset(h :: t, j) &\leftrightarrow \mathcal{S}(h, t, j)\ . \end{aligned} \tag{18}$$

Unifying (see section 4.2) the conclusion (17) (with annotations removed) and the program (18) instantiates $\mathcal{S}$ with $\lambda u, v, w.\ member(u, w) \wedge subset(v, w)$ and completes the step case. We get the (normalized) fully instantiated program

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

3.5 Auxiliary Syntheses

In the course of a synthesis, we need to prevent a meta-variable from becoming instantiated with a program body that violates the definition of pure logic programs (see section 3.1). Instead of directly checking the instantiation, *Periwinkle* parses the program on completion of a synthesis, marking any subformulae that violate the syntactic restrictions on pure logic programs. The universal closure of each such subformula is taken as the specification for an auxiliary synthesis. In the initial program, any subformula for which an auxiliary synthesis was run is substituted with a call to the corresponding auxiliary predicate, and all auxiliary predicates are added to the program. Note that an auxiliary synthesis may require further auxiliary syntheses. Though the process is not guaranteed to terminate, non-termination has not been a problem in practice. An example where an auxiliary synthesis is necessary is the specification

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

which states that m is the maximum element of l. The synthesized program is

$$
\begin{aligned}
\forall m, l.\, max(m,l) \quad \leftrightarrow \quad & l = nil \wedge false \vee \\
& \exists h, t.\, l = h :: t \wedge (m = h \wedge (\forall x.\, member(x,t) \rightarrow x \leq m) \vee \\
& \qquad h \leq m \wedge max(m,t))\ .
\end{aligned}
$$

The subformula $\forall x.\ member(x,t) \rightarrow x \leq m$ violates the definition of pure logic programs, since it contains a universal quantifier and an implication. The auxiliary specification is

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

which states that m is greater than any element of l. The final program is

$$
\begin{aligned}
\forall m, l.\, max(m,l) \quad \leftrightarrow \quad & l = nil \wedge false \vee \\
& \exists h, t.\, l = h :: t \wedge (m = h \wedge aux(m,t) \vee \\
& \qquad h \leq m \wedge max(m,t)) \\
\forall m, l.\, aux(m,l) \quad \leftrightarrow \quad & l = nil \wedge true \vee \\
& \exists h, t.\, l = h :: t \wedge h \leq m \wedge aux(m,t)\ .
\end{aligned}
$$

3.6 Related Work in Program Synthesis

Most program synthesis approaches have originated in the field of functional programming. There has, however, been increased interest in adapting these to logic program synthesis. For a detailed overview of logic program synthesis, see [Deville & Lau 94].

Fribourg [Fribourg 90] and Wiggins [Wiggins 92] both adapt the *proofs-as-programs* approach to logic program synthesis. Fribourg, however, uses $\forall\exists$ specifications. The synthesized programs are thus not truly relational. Wiggins develops a synthesis logic for relational program synthesis with a decidability operator, implemented in a system called *Whelk*. Both systems are interactive. LOPS [Bibel 80, Bibel & Hörnig 84] transforms first-order specifications into logic clauses. LOPS also uses $\forall\exists$ specifications and thus is not really relational. The semi-automatic system of Lau and Prestwich [Lau & Prestwich 88, Lau & Prestwich 90] is based on unfold/fold transformations of logic programs. Our approach synthesizes truly relational programs, like Wiggins and Lau and Prestwich, and unlike Fribourg and LOPS. It is also fully automatic. A more detailed comparison of our approach with those of Wiggins and Lau and Prestwich follows.

The emphasis in *Whelk* [Wiggins 92] is to develop a logic in which relational programs can be synthesized via proofs-as-programs-style extraction. Thus, in the *Whelk* system, synthesis takes place at the object level, not the meta-level, and correctness and executability are ensured in the object-level logic. By contrast, we synthesize and ensure executability at the meta-level, while establishing partial correctness and completeness by a verification proof at the object level. The difference between the two approaches lies in emphasis. While the *Whelk* project focuses more on the logical issues of logic program synthesis, we have put more emphasis on automation. We have therefore chosen as our object-level logic a well-understood formal system, i.e., many-sorted first-order logic with induction, and have taken a perhaps pragmatic approach by using middle-out reasoning for synthesis and by ensuring executability using extralogical means. In the *Whelk* project, on the other hand, a special logic with a decidability operator was developed to synthesize guaranteed executable programs, while automation was a secondary priority.

The proof planning system *CIAM* is currently being adapted to plan proofs in the *Whelk* logic. The techniques developed here will be directly applicable, in particular middle-out induction (see section 4) and extensions to rippling (see section 5). On the other hand, results in the *Whelk* logic could be used to ensure executability at the meta-level without extra-logical means, thus improving our handling of auxiliary syntheses.

The system of Lau and Prestwich [Lau & Prestwich 88, Lau & Prestwich 90] is semi-automatic and unfold/fold-based. It synthesizes partially correct, but not necessarily complete programs. It solves a synthesis problem by bringing it into a normal form and decomposing it top-down into subproblems until the subproblems are easily solved. The program is then composed bottom-up from the solutions of the subproblems. User interaction is required to limit the search space by specifying the desired recursive calls of the program and by deciding which subproblems to solve. The main strategies are definition, implication and matching. The *definition* strategy selects a definition and uses the *if* part of the definition to unfold and the *only if* part to fold. The *implication* strategy exploits known recursive implications. The *match* strategy solves trivial folding problems.

The *subset* example is taken from [Lau & Prestwich 88]. It is thus a good candidate for comparison. The input to the system of Lau and Prestwich are the *subset* specification, the definition of *member*, and a goal specifying the initial unfold/fold problem. The definition strategy is applied twice, which results in two subproblems. The first is solved with the implication, the second with the match strategy. There are no remaining subproblems, and the solution can be composed. This involves the actual unfolding and folding, interleaved with steps to bring intermediate formulae into various types of normal forms. Setting the initial fold problem corresponds to selecting the type of induction. The initial unfolding then corresponds to induction, and the last folding to fertilization. The remaining decomposition, normalizing and composition steps correspond to rippling. In fact, we believe that approach of Lau and Prestwich could be improved by exploiting rippling to guide the folding. This would obviate the need for normal forms.

Lau and Prestwich synthesize partially correct, but not necessarily complete programs, whereas we insist on both partial correctness and completeness. Not requiring completeness has the advantage that the body of the program being synthesized can be strengthened. Although strengthening allows greater flexibility in synthesis, it also increases the search space, which in Lau and Prestwich's work, translates into a need for user interaction. Nevertheless, our approach could well benefit from the strategies of Lau and Prestwich that strengthen formulae to allow folding. This may well be essential when synthesizing larger, more complex programs, or synthesizing programs from partial specifications.

4 Middle-Out Induction

Determining the appropriate type of induction for a given conjecture is a difficult task. The most widely used technique is *recursion analysis* (see section 2). However, recursion analysis works poorly in the presence of existential quantifiers, which arise in $\forall\exists$ specifications of functions. This is because the appropriate induction scheme is bound to 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 and may lead to an unintuitive or inefficient program.

An example where recursion analysis breaks down is quotient remainder

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

Only x and y are available as induction variables, and given the standard definitions of $\times$, $+$, and $<$, recursion analysis cannot find the appropriate induction, which is induction on y, where the induction term is $y + x$.

Recursion analysis works better for the relational conjectures in our approach. The conjecture for a quotient remainder relation qr is

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

where qr is undefined. Since the conjecture is universally closed, we can choose any of x, y, q, and r as induction variables, not only x and y. Hence, recursion analysis stands a better chance of success. For this conjecture, recursion analysis does suggests an appropriate induction: one-step structural induction on q.

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

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

where $even$ is undefined. The natural recursion of the program is two-step recursion, which is not suggested by the standard definition of $\times$. Therefore, we need a more powerful technique.

Middle-out reasoning can provide such a technique. It can be used to postpone the first, crucial step in the planning of inductive proofs, namely the selection of an induction. This was first suggested by Bundy *et al* [Bundy *et al* 90a], but had not been elaborated or implemented. Using meta-variables, we can set up a schematic step case representing many possible inductions. This is achieved by using meta-variables to represent constructors applied to potential induction variables in the induction conclusion. We can then ripple this schematic step case. The application of wave rules successively instantiates the meta-variables. Once fertilization has taken place, the meta-variables are fully instantiated and correspond to a type of induction. We do need to ensure that the induction is a valid one, i.e., that the induction ordering is well-founded. The most general approach would be to prove that the order is in fact well-founded. This is a difficult task, undecidable in general. We currently use a simpler approach, which is to check whether the ordering is among a set of orderings known to be well-founded. Once we have determined that the induction is valid, we can set up the corresponding base cases and complete the proof using standard proof planning methods.

Middle-out induction has two main advantages over recursion analysis: First, it is a more general approach. It can find an appropriate induction even in cases where recursion analysis fails. Second, recursion analysis essentially performs a look-ahead into the rippling process, whereas middle-out induction requires no such look-ahead. However, there are two problems to overcome in middle-out induction: It requires some kind of higher-order unification, and rippling is no longer terminating. These problems are discussed below.

4.1 An Example Synthesis with Middle-Out Induction

We present a variation of the *even* specification above[2]. The conjecture is

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

where *double* is defined as

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

The wave rules for *double* and the replacement axiom for s are

$$double(\boxed{s(\underline{U})}) \ :\Rightarrow \ \boxed{s(s(\underline{double(U)}))} \tag{19}$$

$$\boxed{s(\underline{U})} = \boxed{s(\underline{V})} \ :\Rightarrow \ U = V \ . \tag{20}$$

The schematic step case is

$$\begin{aligned} & even(x) \leftrightarrow (\exists y.\, double(y) = x) \vdash \\ & \quad even(\boxed{\mathcal{C}(\underline{x})}) \leftrightarrow \left(\exists y.\, double(y) = \boxed{\mathcal{C}(\underline{x})}\right) \end{aligned} \tag{21}$$

where $\mathcal{C}$ is the meta-variable standing for the constructor applied to the potential induction variable, and the dashed boxes indicate potential wave fronts, i.e., $\mathcal{C}$ may be instantiated to some function which becomes the wave front or to the identity function $\lambda x.\, x$. The latter means that there is no wave front. Initially, no wave rule applies. To make a wave rule applicable, we need to introduce a case split on the existential variable y. This is done by the `unrolling` method, which is presented in detail in section 5.2. Unrolling on y yields

$$even(\boxed{\mathcal{C}(\underline{x})}) \leftrightarrow \boxed{double(0) = \mathcal{C}(x) \vee \underline{\exists y'.\, double(\boxed{s(\underline{y'})}) = \boxed{\mathcal{C}(\underline{x})}}} \ .$$

Applying wave rule (19) results in

$$even(\boxed{\mathcal{C}(\underline{x})}) \leftrightarrow \boxed{double(0) = \mathcal{C}(x) \vee \underline{\exists y'.\, \boxed{s(s(\underline{double(y')}))} = \boxed{\mathcal{C}(\underline{x})}}} \ .$$

Applying wave rule (20) twice then results in

$$even(\boxed{s(s(\underline{\boxed{\mathcal{C}''(\underline{x})}}))}) \leftrightarrow \boxed{double(0) = s(s(\mathcal{C}''(x))) \vee \underline{\exists y'.\, double(y') = \boxed{\mathcal{C}''(\underline{x})}}} \ .$$

We can simplify and weak fertilize, i.e., apply the induction hypothesis (21) as a rewrite rule. This yields

$$even(s(s(x))) \leftrightarrow even(x) \ .$$

Weak fertilization instantiates $\mathcal{C}$ to $\lambda u.\, s(s(u))$. The step case and the base cases are now completed as described in section 3.

[2] We use the variation here because *Periwinkle* fails on the original, but succeeds for the variation. Though it does find the appropriate induction in the original, i.e., two-step induction on x, it fails on the auxiliary synthesis in the second base case

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

since it is not yet able to simplify $\exists y.\, y \times s(s(0)) = s(0)$ to false.

4.2 Unification

Since we use higher-order meta-variables in our middle-out reasoning, we are confronted with the problem of higher-order unification, which is only semi-decidable. Moreover, there is no unique most general unifier of higher-order terms. When using higher-order terms, therefore, one either accepts this and uses, for instance, the procedure of Huet [Huet 75], combined with backtracking over or selection of possible unifiers, or one uses a restricted subset of higher-order terms with tractable unification, e.g., *higher-order patterns*. The former approach has been taken, for instance, by Hesketh [Hesketh 91] and Ireland [Ireland 92]. The latter approach is taken here.

Higher-order patterns [Miller 91, Nipkow 91] are expressions whose free variables have no arguments other than bound variables. 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 unique most general unifier of unifiable terms.

We have restricted ourselves to higher-order patterns for the terms in which we use meta-variables because they fall naturally into the class of higher-order patterns. 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 satisfied by the definition of pure logic programs. Thus, what we start out with as our program is already a higher-order pattern. 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. Furthermore, the instantiation of the meta-variables occurs via the application of wave rules, which are also higher-order patterns.

4.3 A More General Representation of the Step Case

The representation of the schematic step case used above does not cover more complex induction schemes where the induction term for a variable refers also to other variables. This is the case, for instance, in the quotient remainder example

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

where the induction term for y is $y + x$. Above, we represented the induction term for a variable with a meta-variable applied to the variable, e.g., $\mathcal{D}(y)$ for y. Since the potential induction variables are bound, the instantiation of $\mathcal{D}(y)$ cannot refer to the variables x, q, or r. To allow this, we must generalize the representation of the schematic step case by representing the induction term for a potential induction variable as an application of a meta-variable to *all* potential induction variables. Thus, the induction term for y is represented as $\mathcal{D}(x, y, q, r)$, which, properly annotated, becomes $\boxed{\mathcal{D}(x, \underline{y}, q, r)}$. The schematic step case is then

$$
\begin{aligned}
& qr(x, y, q, r) \leftrightarrow q \times x + r = y \wedge r < x \vdash \\
& qr(\boxed{\mathcal{C}(\underline{x}, y, q, r)}, \boxed{\mathcal{D}(x, \underline{y}, q, r)}, \boxed{\mathcal{E}(x, y, \underline{q}, r)}, \boxed{\mathcal{F}(x, y, q, \underline{r})}) \leftrightarrow \\
& \quad \boxed{\mathcal{E}(x, y, \underline{q}, r)} \times \boxed{\mathcal{C}(\underline{x}, y, q, r)} + \boxed{\mathcal{F}(x, y, q, \underline{r})} = \boxed{\mathcal{D}(x, \underline{y}, q, r)} \wedge \\
& \quad \boxed{\mathcal{F}(x, y, q, \underline{r})} < \boxed{\mathcal{C}(\underline{x}, y, q, r)} .
\end{aligned}
$$

While dealing with this representation is not a problem for *Periwinkle*, it is not particularly fit for human consumption. The implementation supports both the simpler representation in section 4.1 and the more complex representation here.

4.4 Controlling Rippling

Two of the main advantages of rippling are that it gives a tight control on rewriting and that it terminates. The termination proof [Bundy *et al* 93, Basin & Walsh] makes some restrictions, i.e., existential rippling (see section 5.2) and meta-variables are excluded, precisely because they can lead to non-termination. Since middle-out synthesis and induction require meta-variables, we must contend with the possibility of non-termination and devise strategies to avoid it.

Non-termination is in fact more likely than not in rippling in middle-out induction. In terms of the rippling search tree in the schematic step case, where each node corresponds to the application of a wave rule, we can differentiate between two basic types of non-termination: Non-termination in success branches and non-termination in failure branches.

Non-termination in success branches can be avoided by distinguishing between speculative and non-speculative steps. Applying a wave rule to potential wave fronts only, for instance, is a speculative step. Fertilization and applying a wave rule to at least one definite wave front are non-speculative steps. By preferring non-speculative to speculative steps, non-termination on success branches can be avoided. However, this does not avoid non-termination in failure branches. If there were always at least one success branch in the rippling search tree, breadth-first search would solve the problem. Unfortunately, however, this is not the case. A simple example of a rippling search tree with failure branches only is a variant of the associativity of plus[3]

$$\forall x.\ x + (x + x) = (x + x) + x\ .$$

To avoid non-termination in failure branches, we allow only one speculative step, which can be a speculative ripple or an unrolling step (see section 5.2), and then ripple while trying to fertilize as soon as possible. This does mean that *Periwinkle* cannot find a proof for theorems which depend on more than one speculative ripple. This is in fact rare, so that it does not appear to be a severe limitation.

4.5 Related Work in Selection of Induction Schemes

There has been little work on techniques to select induction schemes beyond recursion analysis, except within the framework of the Inka theorem-prover [Biundo *et al* 86], a theorem prover based on resolution and rippling with destructor-style induction.

[3] Here, we would need to generalize before doing induction. In the ordering of methods, however, induction comes before generalization, and the depth-first planner will select induction. This particular example can be solved by a best-first planner with an evaluation function to determine whether induction or generalization should be preferred (see Manning [Manning 92]).

The synthesis system of Biundo [Biundo 88] applies a heuristic called *most-nested function* to specifications in the form of skolemized $\forall\exists$ formulae. A most-nested function is one that occurs at an innermost position in the specification. The recursive arguments of the most-nested function are selected as the induction variables, and the recursion of the most-nested function as the type of induction. The recursive arguments of the most-nested function should also be among the set of variables that are arguments of the skolem function. The type of recursion of the program will then correspond to the type of recursion of the most-nested function. While this heuristic is sufficient for some simple examples, it performs as poorly as recursion analysis in more complex examples. For example, for the (skolemized) quotient remainder specification from [Biundo 89]

$$\forall x, y.\ y \neq 0 \rightarrow plus(times(car(f(x,y)),x), cdr(f(x,y))) = y \ \wedge\ cdr(f(x,y)) < x \ ,$$

the heuristic selects structural induction on y. While the system does find a proof, the resulting program is unintuitive and inefficient.

Protzen [Protzen 94] presents a calculus for constructing induction proofs without committing to an induction scheme at the outset. The ideas are closely related to our middle-out reasoning approach, first presented in [Kraan *et al* 93b], except that they are couched within a destructor style setting and a specialized proof system. The idea is to generate induction hypotheses by rewriting the induction conclusion until all structure introduced by the rewriting is in *tolerable positions*, i.e., surrounds the conclusion or potential induction variables. Tolerable positions basically correspond to our meta-variables. Additionally, Protzen's calculus forces the induction hypotheses to be smaller relative to a well-founded ordering. The approach has not, to the best of our knowledge, been fully implemented yet. We have two concerns with Protzen's techniques. First, in our experience, the search space in such proofs is very large and rather sensitive to how the rewriting is constrained. Hence carefully designed search control is essential, and it is likely that, in practice, Protzen will have to introduce heuristics to restrict the search to have a usable system. Second, it is not clear whether the approach applies to universally quantified conjectures only or also to existentially quantified and synthesis conjectures.

Hutter [Hutter 94] suggests a technique to select induction schemes for $\forall\exists$ formulae which exploits the close relationship between the induction variables, the instantiation of existential variables and the type of induction. Instead of selecting induction variables and the type of induction and then finding instantiations of existential variables, Hutter picks induction variables and instantiations of existential variables, leaving the type of induction to be determined in the course of the proof. Hutter's approach involves two steps: First, the selection of an induction variable and an existential variable and second, the selection of the induction scheme. The selection of the pair of variables is done in a preprocessing step bearing some similarity to recursion analysis. First, all available wave rules are abstracted in that the only information retained is the dominating function symbol and the direction in which the wave front moves—up, down, or across. These abstracted rules are called *labeled fragments*. The term tree of the conjecture is then searched to find a path of labeled fragments such that all instances of a universal and an existential variable are connected, but none of the fragments overlap. Such a path ensures that there is wave rule that can move a wave front in the desired direction at every relevant node. It does not consider the actual form of the wave front, however. Once the variables have been selected, the actual proof is carried out. In the base case, the existential variable is instantiated to the base of the corresponding type. Then, symbolic evaluation is applied. The remaining formula is assumed as the condition of the base case, which completes its proof. The negation of this formula becomes the condition of the step case. In the step case, the existential variable is again instantiated, now to the compound case of the type, and the conclusion is rippled. Once the rippling has terminated, the structure that has accumulated around the induction variable determines its predecessor. As in our approach, the well-foundedness of the induction order remains to be established. To the best of our knowledge, the approach has not yet been implemented.

Hutter's approach and ours are related. Both rely on the rippling of the step case to determine the type of induction. Both require a certain amount of search, Hutter's in the preprocessing step, ours in the rippling. The main difference lies in the fact that Hutter's approach is divided into two steps. The preprocessing corresponds to a lookahead into the rippling, albeit a simplified version. The trade-off between our one-step and Hutter's two-step approach is thus that Hutter's does some of the search in a simplified setting, which reduces the amount of search in the actual rippling, but involves some duplication of effort. We search in the actual rippling, which is more expensive, but we have no duplication of effort. Finally, the preprocessing step of Hutter simply fails if a lemma is missing, since it cannot find a path. This would pose a serious problem in proofs requiring propositional wave rules, when, as in our system, these wave rules are generated on demand.

5 Extensions to Rippling

Synthesizing logic programs is a new application of proof planning and poses new problems to the proof planner. While many of them are specific to program synthesis, some of them can occur in proof planning in general. Methods developed to solve these more general problems are presented together in this section.

5.1 Generating Logical Wave Rules

Initially, *Periwinkle* used a library of around sixty wave rules based on schematic lemmas about logical connectives. Rules (10) and (11) in section 3.4 are examples of such rules. Considering the large number of such wave rules, it would be preferable for *Periwinkle* to recognize the need for one and generate it on demand. A method that does this has been implemented for a large subclass of logical wave rules, i.e., wave rules expressed in terms of propositional connectives only. Wave rule (10) is propositional, whereas wave rule (11) is not, since it involves quantifiers.

The idea underlying the generation of propositional wave rules is that we can conjecture a partially specified lemma that gives rise to the desired wave rule. We then try to fill in the missing part of the lemma by generating the truth table for that part and finding a formula that satisfies that truth table. In the conclusion

$$\ldots \leftrightarrow \neg \boxed{x = h \vee \underline{member(x,t)}}\ ,$$

for instance, the rippling is blocked. To ripple the right-hand side further, we need a wave rule that pushes the negation down over the disjunction. The wave rule we want is thus of the form

$$\neg \boxed{P \vee \underline{Q}} :\Rightarrow \boxed{\mathcal{F}(\underline{\neg Q})}\ ,$$

based on a lemma

$$\neg(P \vee Q) \leftrightarrow \mathcal{F}(\neg Q)\ ,$$

where $\mathcal{F}$ represents the unknown part. We need to find an expression $\mathcal{F}$ with the same truth values as $\neg(P \vee Q)$ containing a subterm $\neg Q$. We proceed as follows: First, we try the simple cases: $\neg Q$ itself and its negation $\neg\neg Q$. This fails, and we create a set of candidate expressions whose top-level connective is some binary connective, with $\neg Q$ as its first argument and an unknown expression E as its second argument. In this example, we consider only conjunction and disjunction, though the implementation also considers equivalence and implication. We construct the candidate expressions and derive the truth tables for the second argument of the top-level connective. The truth table for E in $\neg Q \vee E$ is

P	Q	$\neg(P \vee Q)$	$\neg Q$	E
T	T	F	F	F
T	F	F	T	X
F	T	F	F	F
F	F	T	T	T/F

and the truth table for E in $\neg Q \wedge E$ is

P	Q	$\neg(P \vee Q)$	$\neg Q$	E
T	T	F	F	T/F
T	F	F	T	F
F	T	F	F	T/F
F	F	T	T	T

In the column for E, T indicates that E must be true for the given values of the variables P and Q, F that E must be false, and T/F that E may be either true or false. The symbol X indicates a conflict; no possible value is logically consistent. Any formula with a conflict is discarded. We try to complete the surviving candidates by finding a variable or a negation of a variable that satisfies the derived truth table. In this example, the truth table for $\neg Q \wedge E$ is satisfied by $\neg P$. If no (negated) variable had satisfied the truth table for E, we would have constructed more complex expressions using binary connectives. To cut down on the search space, we restrict the first argument of any binary connective to a propositional variable or its negation, but allow the second argument to be further expanded.

This straightforward approach to generating propositional wave rules sufficed to generate all propositional wave rules in our original database. However, it is not clear whether it is complete. The problem lies in the syntactic restrictions that were made to cut down the search space, and the question is whether there are wave rules that cannot be expressed in the restricted syntactic form.

5.2 Existential Rippling and Unrolling

Rippling as portrayed in section 2 applies to theorems containing universal quantifiers only, and cannot cope with existential quantifiers. To allow a wave rule to apply to existential variables, we introduce existential versions of wave rules. In essence, what an existential wave rule does is to partially instantiate an existential variable to a wave term. A simple wave rule of the form

$$F[\boxed{S[\underline{U}]}] :\Rightarrow \boxed{T[\underline{F[U]}]}$$

can be turned into an existential wave rule

$$\exists \boxed{u}.G[F[\boxed{u}]] :\Rightarrow \exists \boxed{u'}.G[\boxed{T[\underline{F[\boxed{u'}]}]}]\ .$$

To indicate that existential variables can be instantiated to wave terms, they are annotated with potential wave fronts, represented as dashed boxes.

There is a problem when applying existential rippling in logic program synthesis. Wave rules used to rewrite subexpressions of equivalences must be based on equivalence or equality. Unfortunately, existential versions of wave rules are not necessarily equivalence-preserving. An example where the existential wave rule is not equivalence-preserving is the synthesis conjecture that a list k occurs at the back of a list l

$$\forall k, l.\ back(k, l) \quad \leftrightarrow \quad \exists x.\ app(x, k, l)\ .$$

Applying structural induction on l yields the step case

$$\forall k.\ back(k, \boxed{h :: \underline{t}}) \leftrightarrow \exists \boxed{x}.\ app(\boxed{x}, k, \boxed{h :: \underline{t}})\ . \tag{22}$$

Now, we would like to apply the wave rule

$$app(\boxed{H_1 :: \underline{T_1}}, L, \boxed{H_2 :: \underline{T_2}}) :\Rightarrow \boxed{H_1 = H_2 \wedge \underline{app(T_1, L, T_2)}} \tag{23}$$

in its existential version

$$\exists \boxed{x}.\ app(\boxed{x}, L, \boxed{H :: \underline{T}}) :\Rightarrow \boxed{\exists \boxed{x_1}.\ \exists \boxed{x_2}.\ \boxed{x_1} = H \wedge \underline{app(\boxed{x_2}, L, T)}}\ . \tag{24}$$

While the lemma underlying (23)

$$\forall h_1, t_1, l, h_2, t_2.\ app(h_1 :: t_1, l, h_2 :: t_2) \leftrightarrow h_1 = h_2 \wedge app(t_1, l, t_2)$$

holds, the equivalence that would justify (24)

$$\forall l, h, t.\ (\exists x.\ app(x, l, h :: t)) \leftrightarrow (\exists x_1, x_2.\ x_1 = h \wedge app(x_2, l, t))$$

is a non-theorem (The left side is true, but the right side false for $l = h :: t$).

Thus, before we apply an existential version of wave rule, we must establish that it is equivalence-preserving. (Dis-)Proving the underlying lemma, however, is a difficult and expensive task and requires the full power of proof planning. Instead of implementing a method for equivalence-preserving existential rippling, we have developed a less expensive and more generally applicable alternative, *unrolling*.

In the step case (22), with (23) as the only available wave rule, rippling is blocked from the outset by the existentially quantified variable x. Applying wave rule (23) is clearly what is called for. For that, x must be brought into the form $\boxed{x_1 :: \underline{x_2}}$. This can be done by introducing an appropriate case split, i.e., one where one of the cases is $x = x_1 :: x_2$. The case split that lends itself is the one where x is either the empty or a composite list.

When we apply the case split, we must preserve the skeleton of the conclusion and put any additional structure introduced by the case split into a wave front. This is achieved by annotating the case split accordingly. Schematically, a case split on x in $\exists x : nat.\ P(x)$ is annotated as

$$\boxed{P[0] \vee \underline{\exists x : nat.\ P[\boxed{s(\underline{x})}]}}\ .$$

In the *back* example, introducing a case split on x leads to the conclusion

$$\forall k.\ back(k, \boxed{h :: \underline{t}}) \leftrightarrow \boxed{app(nil, k, h :: t) \vee \underline{\exists x_1.\ \exists x_2.\ app(\boxed{x_1 :: \underline{x_2}}, k, \boxed{h :: \underline{t}})}}\ ,$$

to which wave rule (23) applies, yielding

$$\forall k.\ back(k, \boxed{h :: \underline{t}}) \leftrightarrow \boxed{app(nil, k, h :: t) \lor \underline{\exists x_1.\ \exists x_2.\ \boxed{x_1 = h \land \underline{app(x_2, k, t)}}}}\ .$$

The step case can be completed with a further ripple and weak fertilization.

Currently, only case splits on the structure of recursive data types are considered, both for existential and universal quantifiers. The method can be extended to cover more complex case splits.

Unrolling is a speculative step and can cause non-termination. It is therefore controlled in the same way as speculative rippling (see section 4.4). It is worth noting the relationship between equivalence-preserving existential rippling and unrolling. Applying an equivalence-preserving existential wave rule to a conclusion

$$\exists y : nat.\ P(\boxed{s(\underline{x})}, y)$$

yields a conclusion

$$\exists y' : nat.\ \boxed{F(\underline{P(x, y')})}\ . \tag{25}$$

Unrolling initially yields

$$\boxed{P(x, 0) \lor \underline{\exists y' : nat.\ P(\boxed{s(\underline{x})}, \boxed{s(\underline{y'})})}}\ .$$

Because the existential rewrite was equivalence-preserving, the case $P(x, 0)$ must be false. Thus, after we simplify the wave front and apply the original version of the wave rule, we also obtain (25).

5.3 Very Weak Fertilization

The techniques presented so far are mainly devised to allow rippling to continue. In some cases, however, the blockage does not prevent further rippling, but fertilization. *Very weak fertilization* is a method that recognizes one particular type of blockage that occurs frequently when synthesizing relations from functions. It exploits the induction hypothesis in a way that takes the blockage into account. Very weak fertilization applies in particular in cases where we would like to weak fertilize, but we cannot because the corresponding side of the conclusion is an equality that is not yet fully rippled. A simple example where this occurs is the synthesis of a reverse relation from a reverse function

$$\forall k, l.\ rrev(k, l) \quad \leftrightarrow \quad frev(k) = l\ ,$$

where $frev$ is defined as

$$\begin{aligned} frev(nil) &= nil \\ \forall h, t.\ frev(h :: t) &= app(frev(t), h :: nil)\ . \end{aligned}$$

From the $frev$ definition, we get the wave rule

$$frev(\boxed{H :: \underline{T}}) :\Rightarrow \boxed{app(\underline{frev(T)}, H :: nil)}\ . \tag{26}$$

By structural induction on k, we obtain the step case

$$\forall l.\ rrev(t, l) \leftrightarrow frev(t) = l \quad \vdash \quad \forall l.\ rrev(\boxed{h :: \underline{t}}, l) \leftrightarrow frev(\boxed{h :: \underline{t}}) = l$$

Applying wave rule (26) yields

$$rrev(\boxed{h::\underline{t}}, l) \leftrightarrow \boxed{app(\underline{frev(t)}, h::nil)} = l \ .$$

The rippling is now blocked. However, the wave hole on the right-hand side of the equivalence, i.e., $frev(t)$, is identical to the left-hand side of the equation on the right-hand side of the equivalence of the induction hypothesis. Thus, if we can pull the wave hole out of its nested position and into an equality, we can weak fertilize. This can be achieved by introducing a new existential variable as a placeholder for the wave hole and adding the equality between the new variable and the wave hole. Schematically, this corresponds to applying the rewrite

$$\boxed{\phi(\underline{\psi(x)})} = y :\Rightarrow \boxed{\exists y'.\ \phi(y') = y \wedge \underline{\psi(x) = y'}} \qquad (27)$$

which is not quite a wave rule because it does not entirely preserve the skeleton. The skeleton of the left-hand side is $\psi(x) = y$, that of the right-hand side is $\psi(x) = y'$, where y' is a bound variable. The notion of rippling is currently being extended to allow such rules as wave rules as well (see Bundy and Lombart [Bundy & Lombart 95]).

In the example, using the existential variable l', this yields

$$rrev(\boxed{h::\underline{t}}, l) \leftrightarrow \boxed{\exists l'.\ app(l', h::nil) = l \wedge \underline{frev(t) = l'}} \ .$$

We can now exploit the induction hypothesis, since l, the variable corresponding to l' in the hypothesis, is universally quantified. We rewrite the conclusion to

$$rrev(\boxed{h::\underline{t}}, l) \leftrightarrow \exists l'.\ app(l', h::nil) = l \wedge rrev(t, l') \ .$$

Very weak fertilization solves a problem that arises in reasoning about relations rather than functions. The need for this technique stems from the fact that we are synthesizing a relational program from a non-tail-recursive function. The recursive case of the reverse function $frev$ is non-tail-recursive, i.e., its value is defined by a function applied to the result of the recursive call. The flat structure of relations makes such a nesting in the corresponding relation impossible. In the relational case, such nestings can only be expressed by having the corresponding relations share existential variables. Thus, to make progress in the synthesis of a relation, it is necessary to unpack the nesting of functions by introducing existential variables.

6 Implementation and Results

6.1 Implementation

Periwinkle is an adaptation and extension of the *CIAM-Oyster* system [Bundy *et al* 90c]. *CIAM* was designed as a planner to plan inductive proofs in type theory. Recently, however, there has been an increased interest in using *CIAM* for other types of proofs, logics, and proof-checkers, and there is an ongoing effort into making *CIAM* less logic-dependent. At the object level, *Oyster* is being replaced by an interactive proof checking shell *Mollusc* [Richards *et al* 94], which, given a specification of a logic, becomes a proof checker for that logic. Using *Mollusc*, we implemented a proof checker for many-sorted first-order predicate logic with equality and induction, following the Gentzen System $G_=$ from [Gallier 86] and adding appropriate induction rules.

At the planning level, CL^AM needed to be adapted to first-order predicate logic with equality. Beyond the syntax-related adaptations, CL^AM was extended in a number of ways. First, the middle-out reasoning presented here required higher-order pattern unification and a higher-order representation of annotations. Second, we implemented code to handle auxiliary syntheses. Finally, we implemented the new methods and auxiliary code related to these.

6.2 Synthesized Programs

One of our aims was to improve on the results achieved by known synthesis systems, particularly in terms of automation and selection of induction schemes. Three examples from the literature are of special importance: *subset* from Lau and Prestwich [Lau & Prestwich 88], *delete* from Bundy *et al* [Bundy *et al* 90b] and *qr* from Biundo [Biundo 89]. For the first two examples, the aim was to fully automate the synthesis, which, in the literature, was semi-automatic or interactive. For the last example, the aim was to synthesize a better algorithm by finding a more appropriate induction. *Periwinkle* is able to synthesize good programs for all of the specifications automatically and can thus be considered to have achieved its aims.

The problems taken from the literature can be summarized as follows: The *subset* example [Lau & Prestwich 88]

$$\forall x, y.\ subset(x, y) \quad \leftrightarrow \quad (\forall z. member(z, x) \rightarrow member(z, y))$$

posed the problems of rippling using logical wave rules (see section 5.1) and controlling the rippling to avoid non-termination (see section 4.4). While the synthesis requires three instances of user interaction in [Lau & Prestwich 88], it requires none in *Periwinkle*. The *delete* example [Bundy *et al* 90b]

$$\forall x, y, z.\ delete(x, y, z) \quad \leftrightarrow \quad (\exists k, l.\ fapp(k, l) = y \wedge fapp(k, x :: l) = z)$$

contains nested quantifiers which require unrolling (see section 5.2). This example, which had not yet been automated at all, is also done automatically by *Periwinkle*. A remaining problem with the synthesis of *delete* is that it requires a lemma about the append function *fapp*,

$$\forall k, l.\ fapp(k, l) = nil \quad \leftrightarrow \quad k = nil \wedge l = nil\ .$$

Finally, the quotient remainder example *qr* from [Biundo 89]

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

requires middle-out induction (see section 4) and auxiliary syntheses (see section 3.5). Biundo's heuristic selects structural induction on y, which leads to an inefficient, unintuitive program. *Periwinkle* chooses a more appropriate induction, namely one where the induction terms for w and y are $s(w)$ and $y + x$, respectively, which leads to a simpler, more efficient program. The synthesis of *qr* requires three lemmas, the associativity of *plus*, a variant of the cancellation of *plus* and a lemma on $<$ and *plus*

$$\begin{aligned}
\forall x, y, z.\ plus(x, plus(y, z)) &\quad = \quad plus(plus(x, y), z) \\
\forall x, y, z.\ plus(x, y) = plus(x, z) &\quad \leftrightarrow \quad y = z \\
\forall x, y, z.\ (z < x \rightarrow plus(x, y) = z) &\quad \leftrightarrow \quad false\ .
\end{aligned}$$

Table 1 gives an overview of other problems solved by *Periwinkle*. All plans were constructed using middle-out induction. For more detail on the specifications, programs and proofs, see [Kraan 94]. The examples fall into three categories:

Name	Description	Planning time	Plan execution time
add3	Addition relation for three numbers	9.5	5.8
app_length	Combined length of two lists	10.9	5.8
back	see section 5.2	14.0	5.6
before	An element precedes another in a list	39.1	19.2
between	A number is between two numbers	34.2	4.9
delete	see above	95.0	31.3
even	see section 4.1	7.2	4.7
insert	Insert an element before another in a list	124.8	37.8
max	see section 3.5	30.4	10.4
not_member	Non-membership in a list	9.1	3.0
qr	see above	42.1	12.9
rapp	Append relation	9.4	3.7
rplus	Addition relation	6.0	3.6
rrev	see section 5.3	8.4	4.6
rtimes	Multiplication relation	6.8	5.3
subset	see above	21.2	5.0

Table 1: Programs synthesized by *Periwinkle*. Planning and plan execution times are for main syntheses in seconds of CPU time on a Sparc station 10 using Quintus Prolog Release 3.1.4.

1. Synthesizing relations from functions. Such relational programs have the advantage that they can be run in various modes. Thus, for instance, an addition relation can also be used for subtraction. Examples programs are *rapp*, *rplus*, *rtimes* and *rrev*.

2. Synthesizing programs from specifications containing quantified and negated relations. Examples programs are *delete*, *subset* and *max*.

3. Synthesizing more efficient programs from executable but less efficient programs. An example program is *between*.

It would not be fair to omit examples that we would have liked to synthesize, but failed to. These include sorting and partitioning lists. The former fails because *Periwinkle* lacks relational rippling (see section 7.3), the latter because *Periwinkle* is not able to find a required case split.

7 Future Research

7.1 Synthesis

The work in this paper has concentrated on the fully automatic synthesis of recursive logic programs that are partially correct and complete. The system currently tries to find a recursive program, but does not prefer any one type of recursion over another. A useful extension to the system would be to allow the user to specify constraints on the program, for instance by giving the complexity or type of recursion of the program, or by requiring that the program be tail-recursive. Allowing the user to determine the recursive structure of the program would shift the challenge to the proof planner from finding appropriate inductions to finding wave rules, since the induction of the proof is determined from the outset.

7.2 Middle-Out Induction

In middle-out induction, the instantiation of the meta-variables is used as an index into the database of induction schemes. Currently, middle-out induction fails if the ordering determined in the rippling is not among the set of known orderings. However, it may be possible to "salvage" the induction. For instance, the ordering may be a combination of known orderings. Alternatively, there may be an ordering corresponding to the instantiation of a subset of the meta-variables. In the latter case, the step case would have to be patched by introducing an appropriate case split on the non-induction variable. Ideally, however, the proof planner would not have to rely on a set of known induction schemes, but would use general well-founded induction and prove the well-foundedness of the ordering.

7.3 Rippling

7.3.1 Search Control

The control of rippling in middle-out induction poses a considerable challenge. Most importantly, rippling needs to be controlled so that the search space is kept within reasonable bounds and non-termination is avoided. In section 4.4, this was achieved by a simple mechanism, i.e., allowing only one speculative step. Although allowing no more than one speculative step ensures termination, it also cuts the search space down too far—there are proofs which require more than one speculative step. An example requiring two speculative steps is

$$\forall w, x, y, z.\, plus(w, x) = plus(y, x) \;\leftrightarrow\; plus(w, z) = plus(y, z)\ .$$

With only one speculative step, a blockage occurs that prevents fertilization. What is needed is global control over speculative steps such as speculative rippling, generalization, and lemma conjecturing. Such global control can be provided by the planning critics of Ireland [Ireland & Bundy], which, given a failed proof planning attempt, analyze it and suggest ways of correcting it. In the case of middle-out induction, the method and its critic would work together as follows: The method allows an initial speculative ripple to begin rippling. Then, it allows only definite rippling and fertilization. If the rippling fails, the critic can analyze the rippling and suggest the appropriate measure—another speculative ripple, a lemma or a generalization.

7.3.2 Rippling with Relations

The class of programs we can synthesize so far is limited mainly because rippling as presented by Bundy *et al* [Bundy *et al* 93] is based on nested functions. The idea of a wave rule is precisely that it moves constructors or functions up from a nested position. To extend the class of logic programs we can synthesize, the notion of rippling needs to be extended to relations. Relations cannot be nested like functions, but a similar effect is achieved by existentially quantified variables that are arguments of more than one literal. Take, for instance, the step cases of standard functional and relational definitions of list reversal

$$\forall h, t.\, frev(h :: t) \;=\; append(frev(t), h :: nil) \tag{28}$$

$$\forall h, t.\, rrev(h :: t, l) \;\leftrightarrow\; \exists l'.\, rrev(t, l') \wedge append(l', h :: nil, l)\ . \tag{29}$$

While (28) gives rise to a wave rule

$$frev(\boxed{H :: \underline{T}}) :\Rightarrow \boxed{append(\underline{frev(T)}, H :: nil)}\ ,$$

(29) currently does not. The rule

$$rrev(\boxed{H :: \underline{T}}, L) :\Rightarrow \boxed{\exists l'.\ \underline{rrev(T, l')} \wedge append(l', H :: nil, L)}$$

is not a wave rule in the traditional sense because the skeleton of the left-hand side, $rrev(T, L)$, and the skeleton of the right-hand side, $rrev(T, l')$, are not identical. This "almost" wave rule is very similar to the schematic rewrite (27) in section 5 in that they both preserve the skeleton except for the names of bound variables. Such rules are clearly within the spirit of rippling, and the notion of rippling is being extended to cope with them (see Bundy and Lombart [Bundy & Lombart 95]). Relational rippling would enable us to synthesize a large number of programs that are currently beyond the capabilities of the system.

8 Conclusions

In this paper, we have investigated the application of proof planning to the automatic synthesis of logic programs via inductive proofs. The work developed out of existing work in middle-out reasoning [Bundy *et al* 90a] and in proofs-as-programs for logic program synthesis [Bundy *et al* 90b]. The main goals were to synthesize relational programs and to automate the synthesis process. Our work makes four principal contributions:

1. We have applied middle-out reasoning to new applications and have shown how to extend it to overcome related search problems. For example, we show how search arising in higher-order unification can be restricted by the use of higher-order patterns.
2. We have shown that middle-out reasoning is a mechanism through which proof planning can be used to synthesize logic programs fully automatically.
3. We have provided evidence that middle-out reasoning provides an elegant and effective mechanism to select induction schemes. Middle-out induction has made the class of synthesis theorems, for which existing techniques often fail, amenable to automatic proof.
4. We have provided new techniques for unblocking rippling when the blockage is caused by missing wave rules on propositional connectives or by nested quantifiers.

Although we are still far away from automatically synthesizing complex programs from formal specifications, we have made progress towards that goal by automating some of the steps that are traditionally considered Eureka steps, and are therefore normally left to the user. Middle-out reasoning has played a crucial role in this success by providing a elegant framework for speculation, and we believe that its potential is far from exhausted.

Acknowledgements

We would like to thank Sophie Renault for her careful reading of this paper.

References

[Aubin 76] R. Aubin. *Mechanizing Structural Induction*. Unpublished PhD thesis, University of Edinburgh, 1976.

[Basin & Walsh] David Basin and Toby Walsh. A calculus for and termination of rippling. JAR (this issue).

[Basin *et al* 93] David Basin, Alan Bundy, Ina Kraan, and Sean Matthews. A framework for program development based on schematic proof. In *Proceedings of the 7th International Workshop on Software Specification and Design (IWSSD-93)*, 1993. Also available as Max-Planck-Institut für Informatik Report MPI-I-93-231 and Edinburgh DAI Research Report 654.

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

[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.

[Biundo 89] S. Biundo. Automatische Synthese rekursiver Programme als Beweisverfahren. PhD thesis, Universität Karlsruhe, 1989.

[Biundo *et al* 86] S. Biundo, B. Hummel, D. Hutter, and C. Walther. The Karlsruhe induction theorem proving system. In Joerg Siekmann, editor, *8th Conference on Automated Deduction*, pages 672–674. Springer-Verlag, 1986. Springer Lecture Notes in Computer Science No. 230.

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

[Bundy & Lombart 95] A. Bundy and V. Lombart. Relational rippling: a general approach. In *Proceedings of IJCAI-95*, 1995. To appear.

[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.

[Bundy *et al* 93] A. Bundy, A. Stevens, F. van Harmelen, A. Ireland, and A. Smaill. Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62:185–253, 1993. Also available from Edinburgh as DAI Research Paper No. 567.

[Clark & Tärnlund 77] K. L. Clark and S-Å. Tärnlund. A first order theory of data and programs. In B. Gilchrist, editor, *Information Processing*, pages 939–944. IFIP, 1977.

[Clark 79] K. L. Clark. Predicate Logic as a Computational Mechanism. Technical Report 79-59, Dept. of Computing, Imperial College, 1979.

[Deville & Lau 94] Y. Deville and K.K. Lau. Logic program synthesis. *The Journal of Logic Programming*, 19/20:321–350, May/July 1994.

[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.

[Gallier 86] J. Gallier. *Logic for Computer Science.* Harper & Row, New York, 1986.

[Gordon *et al* 79] M.J. Gordon, A.J. Milner, and C.P. Wadsworth. *Edinburgh LCF - A mechanised logic of computation*, volume 78 of *Lecture Notes in Computer Science*. Springer Verlag, 1979.

[Hesketh 91] J.T. Hesketh. *Using Middle-Out Reasoning to Guide Inductive Theorem Proving.* Unpublished PhD thesis, University of Edinburgh, 1991.

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

[Hutter 94] D. Hutter. Synthesis of induction orderings for existence proofs. In *Proceedings of the 12th Conference on Automated Deduction*, 1994.

[Ireland & Bundy] A. Ireland and A. Bundy. Productive use of failure in inductive proof. JAR (this issue).

[Ireland 92] A. Ireland. The Use of Planning Critics in Mechanizing Inductive Proofs. In A. Voronkov, editor, *International Conference on Logic Programming and Automated Reasoning – LPAR 92, St. Petersburg*, Lecture Notes in Artificial Intelligence No. 624, pages 178–189. Springer-Verlag, 1992. Also available from Edinburgh as DAI Research Paper 592.

[Kraan 94] I. Kraan. *Proof Planning for Logic Program Synthesis*. Unpublished PhD thesis, Department of Artificial Intelligence, University of Edinburgh, 1994. Submitted February 1994.

[Kraan *et al* 93a] 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 available as Max-Planck-Institut für Informatik Report MPI-I-92-244 and Edinburgh DAI Research Report 603.

[Kraan *et al* 93b] I. Kraan, D. Basin, and A. Bundy. Middle-out reasoning for logic program synthesis. In D. S. Warren, editor, *Proceedings of the Tenth International Conference on Logic Programming*. MIT Press, 1993. Also available as Max-Planck-Institut für Informatik Report MPI-I-93-214 and Edinburgh DAI Research Report 638.

[Lau & Prestwich 88] K. K. Lau and S. D. Prestwich. Synthesis of recursive logic procedures by top-down folding. Technical Report UMCS-88-2-1, Dept. of Computer Science, University of Manchester, February 1988.

[Lau & Prestwich 90] K. K. Lau and S. D. Prestwich. Top-down synthesis of recursive logic procedures from first-order logic specifications. In D.H.D. Warren and P. Szeredi, editors, *Proceedings of the 7th International Conference on Logic Programming*, pages 667–684. MIT Press, 1990.

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

[Manning 92] A. Manning. Representing preference in proof plans. Unpublished M.Sc. thesis, Dept. of Artificial Intelligence, Edinburgh, 1992.

[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.

[Miller 91] D. Miller. A logic programming language with lambda abstraction, function variables and simple unification. In *Extensions of Logic Programming*, volume 475 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1991.

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

[Protzen 94] M. Protzen. Lazy generation of induction hypotheses. In *Proceedings of the 12th Conference on Automated Deduction*, 1994.

[Richards *et al* 94] B.L. Richards, I. Kraan, A. Smaill, and G.A. Wiggins. Mollusc: a general proof development shell for sequent-based logics. In A. Bundy, editor, *12th Conference on Automated Deduction*, pages 826–30. Springer-Verlag, 1994. Lecture Notes in Artificial Intelligence, vol 814; Also available from Edinburgh as DAI Research paper 723.

[Stevens 88] A. Stevens. A rational reconstruction of Boyer and Moore's technique for constructing induction formulas. In Y. Kodratoff, editor, *The Proceedings of ECAI-88*, pages 565–570. European Conference on Artificial Intelligence, 1988. Also available from Edinburgh as DAI Research Paper No. 360.

[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*, pages 351–368. M.I.T. Press, Cambridge, MA, 1992.