Proof Planning

for

Logic Program Synthesis

Ina Kraan

ARTIFICIAL INTELLIGENCE LIBRARY
UNIVERSITY OF EDINBURGH
80 South Bridge
Edinburgh EH1 1HN

Ph.D.

University of Edinburgh

1994

MAIN
LIBRARY
STORE 1

Proof Planning

for

Logic Program Synthesis

Ina Kraan

ARTIFICIAL INTELLIGENCE LIBRARY
UNIVERSITY OF EDINBURGH
80 South Bridge
Edinburgh EH1 1HN

Ph.D.

University of Edinburgh

1994

30150 013446684

Abstract

The area of logic program synthesis is attracting increased interest. Most efforts have concentrated on applying techniques from functional program synthesis to logic program synthesis. This thesis investigates a new approach: Synthesizing logic programs automatically via middle-out reasoning in proof planning.

[Bundy *et al* 90a] suggested middle-out reasoning in proof planning. Middle-out reasoning uses variables to represent unknown details of a proof. 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 in synthesis is difficult, because the recursion in 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 applied to the 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.

The results reported in this thesis are encouraging. The approach has been implemented as an extension to the proof planner CLAM [Bundy *et al* 90c], called *Periwinkle*, which has been used to synthesize a variety of programs fully automatically.

I declare that this thesis has been composed by myself and that the work described in it is my own. Portions of the work described here have been published previously in [Kraan *et al* 93a] and [Kraan *et al* 93b].

Ina Kraan

Acknowledgements

First and foremost I would like to thank my supervisors Prof. Alan Bundy, Dr. David Basin and Dr. Ian Green. Their support, advice and encouragement has been invaluable. I would also like to express my appreciation for the members of the Mathematical Reasoning Group, who provided such a productive working and research environment. Finally, I gratefully acknowledge the financial support of Sandoz AG and the Schweizerischer Nationalfonds.

Table of Contents

1. Introduction **7**

1.1 Overview 7

1.2 Motivation 8

1.3 Contributions 9

1.4 Organization 10

2. Background **11**

2.1 Logic Programming 11

2.1.1 Logic Programming Languages 12

2.2 Program Synthesis 13

2.2.1 Synthesizing Functional Programs 15

2.2.2 Synthesizing Logic Programs 19

2.3 Summary 22

3. Automated Theorem Proving and Proof Planning **24**

3.1 NQTHM and Inka 25

3.1.1 NQTHM 25

3.1.2 Inka 26

3.2 Proof Planning 26

3.2.1 Methods and Tactics 27

3.2.2 Proof Planning for Inductive Proofs 28

3.2.3 Middle-Out Reasoning 42

3.3 Summary 43

4. Middle-Out Synthesis **44**

4.1 Pure Logic Programs 44

4.1.1 Program Correctness 47

4.2 Proof Planning for Reasoning about Logic Programs 49

4.2.1 Planning Logic Program Verification Proofs 49

4.3 From Verification to Synthesis 53

4.4 An Example Synthesis 57

4.5 Issues in Middle-Out Synthesis 61

4.5.1 Representation 61

4.5.2 Auxiliary Syntheses 63

4.5.3 Controlling Synthesis 65

4.6 Comparison of Middle-Out Synthesis with Other Approaches . . 67

4.6.1 Comparison with Whelk 67

4.6.2 Comparison with Lau and Prestwich 69

4.7 Summary 74

5. Middle-Out Induction **75**

5.1 Outline 75

5.2 An Example Synthesis with Middle-Out Induction 79

5.3 Issues in Middle-Out Induction 81

5.3.1 Unification 81

5.3.2 A More General Representation of the Step Case 87

5.3.3 Controlling Rippling 88

5.4 Comparison of Middle-Out Induction with Other Approaches . . 93

5.5 Summary 95

6. Extensions to Rippling **96**

6.1 Generating Logical Wave Rules 97

6.2 Equivalence-Preserving Existential Rippling 100

6.3 Unrolling for Unblocking 102

6.4 Very Weak Fertilization 106

6.5 Summary 108

7. Implementation and Results **109**

7.1 Implementation 109

7.2 Synthesized Programs 111

7.2.1 Examples for Method Development 112

7.2.2 Examples from the Literature 112

7.2.3 Test Examples 113

7.3 Summary 115

8. Future Research **116**

8.1 Immediate Improvements 116

8.2 Synthesis 117

8.3 Middle-Out Induction 118

8.3.1 Multiple Step Cases 118

8.3.2 Unknown Induction Orderings 118

8.4 Rippling 119

8.4.1 Control 119

8.4.2 Relational Rippling 121

8.4.3 Logical Wave Rules 121

8.5 Summary 123

9. Conclusions **124**

9.1 Improving the Prospects of Middle-Out Reasoning 125

9.2 Automating Logic Program Synthesis 126

9.3 Induction Beyond Recursion Analysis 128

9.4 Unblocking 128

9.5 Summary 128

A. Examples **130**

B. Traces **138**

B.1 Subset 139

B.2 Even 143

B.3 Reverse 147

C. Methods and Submethods **151**

C.1 Methods 151

C.1.1 Mor_induction 151

C.1.2 Ripple 153

C.1.3 Synthesis 154

C.2 Submethods 156

C.2.1 Speculative_step 156

C.2.2 Prop_wave 157

C.2.3 Unroll 159

C.2.4 Very_weak_fertilize 161

C.2.5 Wave 165

D. Gentzen System $G_=$ for Languages with Equality **169**

E. Higher-Order Pattern Unification Algorithm **171**

Bibliography **172**

List of Figures

3–1 General structure of inductive proofs 29

3–2 The ind_strat method 29

3–3 The base_case method 32

3–4 The sym_eval method 32

3–5 The step_case method 33

3–6 A wave rule for + ... 35

3–7 The ripple method 41

3–8 The fertilization method 41

4–1 Schematic base case in verification 50

4–2 Schematic step case in verification 53

4–3 Schematic step case in synthesis 56

4–4 Schematic base case in synthesis 56

4–5 Overview of verification versus synthesis 57

Chapter 1

Introduction

1.1 Overview

The research presented in this thesis focuses on the application of *proof planning* and *middle-out reasoning* to problems arising in program synthesis. Program synthesis is the process of deriving an executable program from a specification. Here, a specification is a formal description of the problem to be solved: Proof planning entails explicit reasoning about proofs and how to carry them out. Middle-out reasoning allows the proof planning to continue even though an object being reasoned about has not yet been specified. We can thus reason about and plan proofs while leaving certain elements of the proof to be filled in at a later stage. In program synthesis via inductive proofs, there are two things that are initially unknown: First, most obviously, the program to be synthesized, but second also the type of induction of 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 represents unspecified objects in the proof with variables and instantiates them via unification in the course of the planning. The use of such variables causes several problems. First, it signifies a loss of information and thus a loss of search control. Additional means of search control are

therefore needed. Second, the variables can be higher-order. Hence, the issues of higher-order unification and its intractability need to be addressed. We restrict the use of of higher-order variables to higher-order patterns, for which unification is decidable.

We synthesize programs by planning the verification of an initially unknown program. In particular, we synthesize logic programs, and verify them by proving the logical equivalence of specification and program. The proof planning yields both a program and a plan for the verification proof. Providing the plan executes successfully, the synthesized program is partially correct and complete.

Middle-out reasoning in proof planning for program synthesis has been implemented as an extension of the proof planning system CLAM. The extended system, *Periwinkle*, has been used to synthesize a variety of programs.

1.2 Motivation

In software development, a major concern is ensuring that software meets its requirements. To make this task more manageable, the process of software development has traditionally been broken down into various phases, i.e., requirements analysis, specification, design, implementation, testing (and/or verification) and maintenance. Ideally, the end product of each phase would satisfy completely the requirements of those of the previous ones, and there would be no need for maintenance. In reality, however, this is not the case, and software maintenance is the dominant cost factor of software development. If we were able to automatically synthesize programs that would be guaranteed to meet their specifications, errors in the design and implementation phases would be eliminated, and the maintenance costs of software drastically reduced. At the same time, automated program synthesis would reduce software development efforts in two ways. First, since we synthesize programs that are correct, programming and verification collapse into the single task of synthesis. This is in contrast to traditional software development, where programming and verification are two

separate tasks: Programs are coded, and only subsequently checked, either via empirical testing or, more seldom, via formal verification. Second, when a specification is changed and the corresponding program needs to be rewritten, it may be possible to reuse parts of the synthesis process itself.

A second major source of errors is the specification phase, i.e., the process of finding a formal specification that adequately reflects the informal description of requirements. This can be subdivided into two tasks, writing the specification and ensuring that the specification adequately reflects the requirements. The latter is known as *validation*. Validation is easier if the specification is executable, i.e., it serves as a prototype and provides *touch-and-feel* experience. However, the price to pay for having executable specifications is a restricted expressive power of the specification language, which makes the task of writing specifications more difficult. [Hayes & Jones 89], for example, presents a number of examples which can be expressed more naturally in a non-executable way. If we were able to automatically synthesize programs from specifications, we would no longer have to worry about specifications being executable and could thus have a more expressive specification language and touch-and-feel experience nonetheless, provided the synthesis is sufficiently fast.

We thus believe that the automatic synthesis of correct programs from specifications is a worthwhile goal to pursue, and the aim of this thesis is to make progress towards this ultimate goal.

1.3 Contributions

This thesis makes new contributions in two areas: Automatic logic program synthesis and the selection of induction schemes. First, it demonstrates how the combined techniques of middle-out reasoning and proof planning can be exploited to automate logic program synthesis. To our knowledge, most logic program synthesis systems still require some degree of interaction by the user. The approach we present can equally be applied to logic program transformation,

since our programming language, pure logic programs, is a subset of our specification language, first-order predicate logic. Hence, a given program, which is perhaps easy to understand but inefficient, can simply be regarded as a specification, and a new, more efficient program can be synthesized from the original program.

Second, it demonstrates how the technique of middle-out reasoning can be used to solve a difficult problem in automated theorem proving, namely the selection of induction schemes for inductive synthesis proofs. The solution we propose is a limited, but practicable one.

1.4 Organization

This thesis is organized as follows: Section 2 presents the general background against which this thesis is set, by giving a brief introduction to logic programming, reasoning with logic programs and program synthesis, and by presenting an overview of various program synthesis systems. Section 3 gives an overview of automated inductive theorem proving, concentrating in particular on the proof planning work carried out in the Mathematical Reasoning Group at the University of Edinburgh, which is the foundation upon which this thesis is built. Sections 4 and 5 form the heart of the thesis, presenting the application of middle-out reasoning to logic program synthesis and to the selection of induction schemes. Section 6 presents extensions to the proof planning method known as rippling that proved necessary for synthesis. Section 7 reports on the implementation and on the practical results we have achieved. Finally, Section 8 presents ideas for further research and Section 9 draws conclusions.

The appendices include examples that have been synthesized using the system *Periwinkle* (Appendix A) and traces for some of these examples (Appendix B). They also include listings of the methods extended or developed in this thesis (Appendix C). Finally, Appendix D lists the logic, Appendix E the unification algorithm we use.

Chapter 2

Background

2.1 Logic Programming

Logic programming is the use of logic as a computational formalism. Logic is used to express knowledge, and inference is used to manipulate knowledge. A logic program is a set of axioms, and computation is the proof of a goal formula from these axioms. [Kowalski 74] shows that a Horn clause

$$A \leftarrow B_1, \ldots, B_n$$

can be read not only declaratively, but also procedurally: To solve A, solve B_1 through B_n. The two main advantages of declarative programming over imperative programming are that it favors mathematical accountability and that it separates knowledge from its use [Hogger 90].

Logic and functional programming are both based on an abstract model of computation. They differ in that functional programs are single-valued, whereas logic programs are multivalued [Bundy *et al* 90b], returning alternate values on backtracking. Logic programs are also often partial in that they may return no value, i.e., fail.

In the following sections, we will briefly discuss the foundations of some logic programming languages. For more details on logic programming, see [Lloyd 87, Hogger 90, Sterling & Shapiro 86, Deville 90].

2.1.1 Logic Programming Languages

A logic programming language is defined by two main elements: The programming logic and the inference system. In general, the programming logic is a subset of first-order predicate logic, and the inference system is sound, complete and sufficiently efficient for that subset. The language can be typed or untyped. Most programming logics are based on Horn clauses, and most inference systems are resolution-based [Robinson 65]. Logic programming languages may also contain non-logical primitives. Although there is much interest in parallel and constraint-based logic programming languages, these are beyond the scope of this thesis.

Programming Logics and Inference Systems

In this section, we follow the presentation of [Lloyd 87]. The logic programming languages discussed are *definite programs* (pure Prolog), *normal programs* (pure Prolog with negation as failure) and *programs* (e.g., pure Gödel [Hill & Lloyd 92]).

Definite Programs Definite programs are finite sets of definite program clauses. A *definite program clause* is a clause of the form

$$A \leftarrow B_1, \ldots, B_n$$

where $n \geq 0$ and $A, B_1, \ldots, B_n$ are atoms. A commonly used semantics for definite programs is the least Herbrand model. A commonly used inference system that is sound and complete for definite programs is SLD-resolution. Pure Prolog with the occurs-check is an implementation of definite programs with SLD-resolution.

Normal Programs Normal programs are finite sets of program clauses. A *program clause* is a clause of the form

$$A \leftarrow L_1, \ldots, L_n$$

where $n \geq 0$, A is an atom and $L_1, \ldots, L_n$ are literals, i.e., atoms or negated atoms. A commonly used semantics for normal programs is completions of programs [Clark 78]. A commonly used inference system that is sound and complete for definite programs is SLDNF-resolution. Pure Prolog with negation-as-failure and the occurs-check is an implementation of normal programs. Full Prolog is an implementation of normal programs, with a number of non-logical primitives added.

Programs Programs are finite sets of program statements. A *program statement* is a first-order formula of the form

$$A \leftarrow W$$

where A is an atom and W is empty or a first-order formula. A commonly used semantics for programs is again completions of programs [Lloyd 87, page 109]. Programs are translated into normal programs using the Lloyd-Topor transformation [Lloyd 87, pp. 112-115]. Pure Gödel is an implementation of *typed* programs. Full Gödel includes some additional non-logical primitives.

2.2 Program Synthesis

Program synthesis is the process of obtaining a program from a specification. There are many ways of going about program synthesis: It can be automatic or interactive, provably correct or not; the specification can be partial or complete, formal or informal. Here, we are concerned with the fully automatic synthesis of correct programs from complete, formal specifications [1]. We regard program transformation as a special case of program synthesis—one where the specification is already an executable program.

---

[1] A complete specification is one that describes all, a partial specification one that describes some properties of a relation. A program satisfying a partial specification therefore computes a subset of the specified values. Thus, for instance, the positive

Within synthesis from formal specifications, existing approaches are usually described as constructive or deductive. The distinction between the two is somewhat blurred, and whether an approach is classified as constructive or deductive is mainly a matter of perspective.

Constructive approaches are usually called such either because they are embedded within the framework of a constructive logic, or because they emphasize the actual construction of witnesses for existentially quantified variables in the specification. Many constructive approaches fall within the *proofs-as-programs* paradigm (see Section 2.2.1). Deductive approaches are called such because they emphasize that the program can be deduced from the specification. That the distinction is fuzzy is particularly apparent in the approach of Manna and Waldinger [Manna & Waldinger 91]. Although they themselves consider their work to be deductive, since the program is synthesized in the course of a deduction, it would be equally fair to call their approach constructive, since it constructs a witness for an existential variable in the course of the deduction. Manna and Waldinger prefer the term deductive, since they emphasize that the logic in which they work is not constructive [Manna & Waldinger 92].

The approach we take in this thesis can also be viewed as either deductive or constructive. It is deductive in that we prove the equivalence of specification and program, which means that the program is a deductive consequence of the specification. It is constructive in that the meta-variable we use to represent the body of the program at the proof-planning level is implicitly existentially

---

integer square root relation is partially specified by

$$\forall x{:}int, y{:}int.\ x^2 \leq y < (x+1)^2$$

and completely specified by

$$\forall x{:}int, y{:}int.\ x^2 \leq y < (x+1)^2 \wedge x \geq 0\ .$$

Our approach could extend to partial specifications by weakening the proofs from partial correctness and completeness to partial correctness only.

quantified. The successive instantiation of the meta-variable in the course of the proof planning corresponds to the construction of a witness for an existential variable.

Most program synthesis approaches have originated in the field of functional programming. Recently, however, there has been an interest in applying ideas from functional program synthesis to logic program synthesis. In the following sections, we will give a brief overview of various functional program synthesis and transformation systems, followed by a somewhat more detailed presentation of logic program synthesis and transformation systems. This overview is not meant to be complete, but representative of the various possible approaches. We have limited ourselves to synthesis systems that synthesize correct programs from formal specifications.

2.2.1 Synthesizing Functional Programs

Specifications of functional programs are usually of the form

$$\forall x \exists y.\, Spec(x, y)\ .$$

This can be read as: For all input x, there exists an output y such that the specified relation $Spec$ holds. The task of functional program synthesis is to provide a function f that computes a witness for the existential variable y. Thus, we want a function f such that

$$\forall x.\, Spec(x, f(x))$$

is true.

In the following sections, we present various approaches how to find such functions.

Type-Theoretic Approaches

A number of systems incorporate the *proofs-as-programs* principle. This principle is based on the *Curry-Howard isomorphism* and the *propositions-as-*

types principle [Howard 80] [Curry & Feys 58]. The basic idea of the Curry-Howard isomorphism is that each axiom and rule of inference of constructive logic corresponds to a rule of term formation in the typed λ-calculus and vice versa. Every construction of a proof of a formula thus also corresponds to a construction of a term in the typed λ-calculus. The basic idea of the *propositions-as-types* principle is that a proposition can be viewed as a type where the type inhabitants are the proofs of the proposition. Proving a proposition thus corresponds to showing that the type of the proposition is inhabited. These ideas were taken up in type theories such as [Martin-Löf 79] or [Coquand & Huet 88], and implemented in systems such as Nuprl [Constable *et al* 86], LEGO [Luo & Pollack 92] or Coq [Dowek *et al* 91]. Since the λ-calculus is effectively a functional programming language, a term corresponding to a proof can be interpreted as a program whose type is the formula proved.

In the Nuprl system [Constable *et al* 86], for instance, proving that a type is inhabited requires specifying how to build an object of that type. A proof of a specification theorem

$$P : (\forall x : i.\ \exists y : o.\ S(x, y))$$

corresponds to the construction of a term P which, applied to a given x, will compute a witness for y in the course of its evaluation. This term is called the *extract term*. Nuprl and its Edinburgh reimplementation Oyster [Horn 88] are both interactive proof assistants for a variant of Martin-Löf type theory [Martin-Löf 79] and can be used to synthesize functional programs from higher-order ∀∃-formulae. Both systems include a language in which tactics, i.e., programs that apply a series of inference rules, can be built. Proofs can thus be partially automated. The proof planning system CLAM [Bundy *et al* 90c] generates tactics that can be executed in Oyster. Thus, Oyster and CLAM can be combined to synthesize programs automatically.

LEGO [Luo & Pollack 92] is a proof assistant for a variety of related type systems, including the Edinburgh Logical Framework [Harper *et al* 87] and the Calculus of Constructions [Coquand & Huet 88]. While Nuprl and Oyster construct sequent-style proofs, LEGO constructs natural deduction-style proofs.

Coq [Dowek *et al* 91] is a proof assistant for the Calculus of Constructions. It allows, among others, the extraction of ML [Harper *et al* 86] programs from proofs. Coq, like Nuprl and Oyster, supports tactics.

Manna and Waldinger

The program synthesis work described in [Manna & Waldinger 91] is similar to the proofs as programs approach in that every proof step is also a program construction step. The main differences are that Manna and Waldinger's system uses tableaux, synthesizes programs expressed in a simple, functional *if-then-else* programming language, uses an untyped first order classical logic and starts from first-order ∀∃-formulae as specifications. The system is interactive.

A synthesis tableau consists of a column of assertions, a column of goals and a column of outputs for each output variable. Each rule application adds new rows to a tableau while preserving certain of its properties. Rules are applied until the tableau contains a row with the assertion false (or the goal true), and outputs are expressed in the vocabulary of the programming language. The rules include *induction* and several versions of *resolution*, and have been shown to be sound.

Biundo

[Biundo 88, Biundo 89] is an example of synthesis of functional programs via deduction. Specifications are ∀∃-formulae of the form

$$\Psi = \forall \overline{x}.\exists y.\forall \overline{z}.\ \Phi[\overline{x}, y, \overline{z}]$$

where Φ is a quantifier-free first-order formula. The formula is skolemized to

$$\Psi = \forall \overline{x}.\forall \overline{z}.\ \Phi[\overline{x}, f(\overline{x}), \overline{z}].$$

Transformation rules are applied to the skolemized specification until a transformation tree is obtained where the leaf nodes form a set of formulae such that a subset of these formulae forms a complete definition of the skolem function f, and

the remaining formulae can be proved using this definition. The programs are functional programs expressed in first-order predicate logic. The rules include *evaluation*, *substitution*, *case analysis* and *induction*. The system, which is a component of the Inka theorem prover [Biundo *et al* 86] (see Section 3.1.2), is fully automated.

Synthesis is broken down into four stages, within which a number of heuristics guide the synthesis process. The four stages are induction, evaluation, extraction and elimination. The induction stage applies induction and brings the formulae into a normal form. The evaluation stage applies symbolic evaluation. The extraction stage uses equalities and the substitution axiom to extract the skolem function from nested positions. Finally, the elimination stage eliminates superfluous conditions from definition formulae.

KIDS

Transformation of functional programs goes back to [Burstall & Darlington 77], which suggests various transformation rules to derive more efficient programs. Of the many transformation-based systems (see [Partsch & Steinbrüggen 83]), we have selected KIDS [Smith 90] as a powerful and general representative, incorporating most known transformation techniques.

KIDS is a partially automated program transformation system. First, a domain theory is created in which the specification is written. Then, a design tactic is applied to determine the type of algorithm, e.g., divide-and-conquer or global search. Next, a number of optimizing transformations can be applied, including *simplification*, *partial evaluation*, *finite differencing* and *data type refinement*. The main effort in using KIDS is building up the domain theory. The richer the domain theory, i.e., the more lemmas it contains, the more powerful are the transformations that can be applied. Interaction is at a high level—the user simply determines which transformation to apply to which part of the program. The actual transformation is then carried out via tactics in a theorem prover.

2.2.2 Synthesizing Logic Programs

The specification of a logic program is a description of the relation which the program is to compute, usually given as a first order formula. The precise format of the specification varies from approach to approach.

Most of the work in logic program synthesis is an adaptation of existing work in the fields of functional program synthesis. Whereas the adaptation of deductive synthesis and program transformation is fairly straightforward, the adaptation of proofs-as-programs raised a number of problems. The main obstacle is that proofs-as-programs synthesizes total functions only. Logic programs, by contrast, are partial and multivalued [Bundy *et al* 90b]. They can return no value, i.e., fail, or return alternate values on backtracking.

In the following, we present a representative, but not complete, overview of logic program synthesis systems. For early work on the foundations of logic program synthesis, see [Clark & Darlington 80, Clark 81, Hogger 81]. For a more detailed overview, see [Deville & Lau 93].

ExExE

In [Fribourg 90], one way of adapting the proofs-as-programs paradigm to logic program synthesis is presented. Programs are synthesized in Prolog-style proofs. A standard Prolog goal is of the form

$$\exists \overline{y}.\ p(\overline{y})\ ,$$

where $p(\overline{y})$ is a conjunction of atoms. Fribourg extends Prolog goals to those of the form

$$\forall \overline{x}.\exists \overline{y}.\ q(\overline{x},\overline{y}) \Leftarrow r(\overline{x})\ ,$$

where $q(\overline{x},\overline{y})$ and $r(\overline{x})$ are conjunctions of atoms. The Prolog execution rule, SLD resolution, is extended to the rules of *definite clause inference*, *simplification* and *restricted structural induction*. The rules of definite clause inference and structural induction are each associated with a program construction rule.

Given a specification in the form of an extended goal, extended Prolog execution will return a program to compute $\overline{y}$ in terms of $\overline{x}$. The main disadvantages of this approach are: The logic program will only be correct in the modes where the variables corresponding to $\overline{x}$ are ground and the variables corresponding to $\overline{y}$ are variable, and the program will return only one answer. It is thus a functional program in the guise of a logic program. This approach has been implemented in the interactive system ExExE [Bouverot 91].

Whelk

Whelk [Wiggins *et al* 91, Wiggins 92, Wiggins 93] is an interactive proof editor for logic program synthesis which also takes the proofs-as-programs approach, but avoids the disadvantages of the previous approach. Whelk synthesizes pure logic programs and translates them into Gödel or Prolog.

In Whelk, specifications of logic programs are of the form

$$\forall x.\ \partial(\mathrm{Spec}(x))\ , \tag{2.1}$$

where $\mathrm{Spec}(x)$ is an arbitrary first-order formula and ∂ is a decidability operator, read as: "It is decidable whether". The logic developed for the Whelk system allows a logic program $\mathrm{Prog}(x)$ to be extracted from a proof of (2.1) in such a way that

$$\vdash \forall x.\ \partial(\mathrm{Spec}(x)) \text{ iff } \vdash \forall x^{*}.\ \mathrm{Spec}(x)^{*} \leftrightarrow \mathrm{Prog}(x^{*})$$

holds, where $\mathrm{Spec}(x)$ is a formula specifying a program, $\mathrm{Prog}(x^{*})$ is a program which fulfills the specification, and * is a mapping from the logic of the specification to the logic of the program. The Whelk system distinguishes between the logic in which the specification is written, which contains functions and the boolean type $\{true, false\}$, and the logic of the synthesized program, which is similar to that of pure logic programs (see Section 4.1). The Whelk logic is a sorted first-order logic with equality, extended with inference rules for the decidability operator ∂. Each inference rule for ∂ is associated with a program construction rule.

LOPS

LOPS [Bibel 80, Bibel & Hörnig 84] is a system that transforms first order specifications into logic clauses. These clauses are not directly executable, but can easily be translated into executable programs. LOPS is a deductive and transformation-based system. Given a specification of the form

$$\forall x \exists y.\, IC(x) \rightarrow OC(x, y)$$

where IC is the input condition and OC the output condition, a number of strategies are applied to obtain an algorithm. These strategies are GUESS and DOMAIN, GET-G, GET-DNF, GET-REC and GET-PE. GUESS and DOMAIN guess an answer belonging to the set of solutions of a *subset* of the output conditions OC. GET-G rewrites a given formula according to domain and goal-dependent equivalence transformation rules, GET-DNF rewrites the goal into disjunctive normal form, GET-REC finds a suitable recursion, and GET-EP makes predicates recursively evaluable. Though [Bibel 80, Bibel & Hörnig 84] show that the resulting programs can easily be translated into Prolog, the LOPS system synthesizes functional programs, since it uses $\forall\exists$ specifications. [Neugebauer 93], however, has adapted the LOPS approach to logic program synthesis.

Tamaki and Sato, Kanamori and Horiuchi

In the field of logic program transformation, [Tamaki & Sato 84] show how the unfold/fold techniques of [Burstall & Darlington 77] can be applied to pure logic program transformation in such a way that the least Herbrand model is preserved. Their transformation rules are *definition*, *unfolding*, *folding*, *deletion*, *goal merging* and *case splitting*. [Kanamori & Horiuchi 87] adapt the unfold/fold transformation method of Tamaki and Sato to allow definitions which are generalizations of definite clauses. The transformation of the more general formula into definite clauses constitutes the synthesis of a program.

The program synthesis system in [Lau & Prestwich 88b, Lau & Prestwich 88a, Lau & Prestwich 90] is an interactive system that uses unfold/fold transformations to transform a first order specification into a partially correct program. The desired recursive structure of the program is provided by the user in the form of a meta-goal. The initial meta-goal is repeatedly divided into subgoals until all subgoals can be solved, and the overall solution is obtained by composing the solutions of the subgoals appropriately. The user is prompted for assistance in setting up subgoals.

The overall strategy is to unfold only when it enables a subsequent fold. Folding is done automatically using the strategies *match*, *modus ponens*, *implication* and *definition*. The match strategy checks whether the current folding problem is trivial. The modus ponens strategy introduces new predicates. The implication strategy is a decomposition strategy which uses known recursive implications. The definition strategy is another decomposition strategy which decomposes a definition by using its "if" part to unfold, and its "only if" part to fold. The folds are found by setting up the appropriate fold subproblems. The system synthesizes only the recursive clauses of the program, not the base clauses.

2.3 Summary

In this section, we briefly introduced logic programming, and then presented various approaches to program synthesis. In the following, we will be presenting an alternative approach to logic program synthesis. Our main goals are to synthesize truly relational programs, i.e., to avoid mode-sensitivity, and to automate the synthesis fully.

Approaches that avoid mode-sensitivity include Whelk (Section 2.2.2) and Lau and Prestwich (see Section 2.2.2). Approaches that are partially or fully

automated include Biundo (Section 2.2.1) and Lau and Prestwich (Section 2.2.2). A detailed comparison of our work with these approaches can be found in subsequent chapters.

Chapter 3

Automated Theorem Proving and Proof Planning

Automated theorem proving is a subdiscipline of Artificial Intelligence concerned with the automatic derivation of proofs. The central problem within automated theorem proving is controlling the large search space for proofs. At every step in the proof, one must decide which inference rule to apply and how to apply it. Since many inferences are normally possible at any stage in a proof, the branching factor of the search space is large, and exhaustive search is prohibitively expensive.

There are two principal approaches to controlling the search space of automated theorem proving. One approach reduces the search space by using a uniform representation, usually clausal form, and a uniform inference rule, usually resolution. Even so, there is often a choice of possible inferences, and heuristics are needed to guide the proof. However, the inference system is unnatural for many humans, which makes it difficult to follow proofs and devise heuristics.

An alternative approach to automated theorem proving uses more traditional inference systems. Though such systems generally have a higher branching factor, the patterns of proofs are more readily understood by humans. This makes it possible to devise heuristics which correspond to strategies humans use when doing proofs. Among the best-known such systems is the theorem prover NQTHM [Boyer & Moore 88]. One problem with this type of prover is that

the heuristics are often built-in and thus inflexible and difficult to understand. To avoid this, [Bundy 88] suggests using a meta-logic to reason about and plan proofs explicitly. This approach is known as *explicit proof planning.*

The remainder of this chapter is devoted to a brief overview of two inductive theorem proving systems, NQTHM [Boyer & Moore 88] and Inka [Biundo *et al* 86], and a more detailed presentation of the proof planning approach underlying our synthesis approach.

3.1 NQTHM and Inka

3.1.1 NQTHM

NQTHM [Boyer & Moore 88] is a well-known heuristic theorem prover for quantifier-free first-order logic with induction. The theorem prover uses six basic proof techniques, namely *simplification*, *destructor elimination*, *cross-fertilization*, *generalization*, *elimination of irrelevance* and *induction*. Each technique is a program that takes a formula as input and returns a set of formulae as output. The programs are derived rules of inference: The input formula is provable if each of the output formulae are provable. The techniques are tried in turn, in the order listed, on each remaining formula until all formulae have been proved (or all techniques fail). Users can guide the prover by providing lemmas and selecting sets of rewrite rules.

Simplification is the only technique that can return an empty set of formulae, i.e., complete the proof of a formula. It is a complex technique that uses decision procedures, term rewriting and what are known as meta-functions. Destructor elimination uses axioms and lemmas to rewrite destructive terms as constructive terms, which are preferred by a number of heuristics in the system. Cross-fertilization exploits the induction hypothesis (see also Section 3.2.2). Generalization generalizes formulae, usually by strengthening them in such a way that they become amenable to induction. Elimination of irrelevancy identifies

and removes irrelevant hypotheses that could complicate an induction. Finally, induction finds and applies a suitable induction scheme (see also Section 3.2.2).

NQTHM is a powerful theorem-prover that is widely used. It has one drawback for non-experts, however, which is that its heuristics are embedded directly in its derived rules of inference. Because the heuristics are implicit in the code, they can be difficult to understand and extend. Proof planning, which is presented in the following section, is an attempt to address this problem. Also, NQTHM does not lend itself to synthesis very well, since its logic is quantifier-free.

3.1.2 Inka

Inka [Biundo *et al* 86] is an inductive theorem-proving system that is based on and extends the heuristics of NQTHM. Inka differs from NQTHM in several ways. First, it is resolution-based. Second, it has a component which automatically proves termination properties of recursive program. Such proofs often need to be provided as lemmas in NQTHM. Finally, and most importantly, Inka uses a logic which allows existential quantification. The program synthesis system of Biundo [Biundo 89] (Section 2.2.1) is the component of Inka that proves existentially quantified theorems.

3.2 Proof Planning

Proof planning is an attempt to make theorem-proving heuristics more explicit by using a meta-logic to reason about and plan proofs. Proof plans are formed in the meta-logic by successively applying *methods* to a conjecture until a combination of methods has been found that forms a complete plan of the proof. Each method is a partial specification of a *tactic* [Gordon *et al* 79], which is a program that applies a number of object-level inference rules to a goal formula. Explicit proof planning has been implemented in the proof planner CLAM

[Bundy *et al* 90c]. CLAM constructs plans for inductive proofs in a variant of Martin-Löf type theory [Martin-Löf 79]. The proof plans can be executed in Oyster, a sequent-style interactive proof checker for the type theory.

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

3.2.1 Methods and Tactics

As mentioned in the previous section, the basic elements of proof planning are methods. Methods are relations of the form *method(Name, SequentIn, PreCond, PostCond, SequentsOut, Tactic)*, where

Name is the name of the method.

SequentIn is the pattern of the input sequent.

PreCond is a set of conditions that must be met by the input sequent for the method to be applicable.

PostCond is a set of conditions that will hold for the output sequents if the tactic succeeds.

SequentsOut is a set of patterns for the output sequents that will result if the tactic succeeds.

Tactic is the name of the tactic associated with the method.

Pre- and post-conditions are expressed in the language of the meta-logic. Although the tactic associated with a method often goes by the same name, this is not necessarily the case. There are, for instance, methods that only have an effect at the meta-level, not at the object-level. The tactic associated with these methods is the trivial tactic that does nothing.

Methods are partial specifications of tactics 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 (or effects) will be true of the resulting sequents. Each method captures a particular notion of progress towards a complete proof, while leaving the tedious details of how to achieve the progress to the tactic.

3.2.2 Proof Planning for Inductive Proofs

The proof planning work in Edinburgh has concentrated mainly on planning inductive proofs. There are two challenges specific to automating inductive proofs: The selection of an appropriate induction scheme and the exploitation of the induction hypothesis in the step case of the proof.

The overall structure of inductive proofs follows the schema of Figure 3–1. The first step is the selection of an induction scheme. Then, symbolic evaluation, simplification and tautology checking are applied to the base case. In the step case, the induction conclusion is rewritten in such a way that the induction hypothesis can be exploited. This rewriting is called *rippling* and is explained in detail in the following sections. Finally, the induction hypothesis is exploited using a technique called *fertilization*, which is also explained below.

The proof planning methods presented in this section are those of CLAM 2.0 for type theory. Some modifications of these methods are required to use CLAM for first-order predicate logic. These are discussed as they arise in the following sections. Section 7.1 contains some of the implementational details involved.

The central method for inductive proofs is ind_strat (see Figure 3–2), which is a compound method embodying the overall structure of the family of inductive

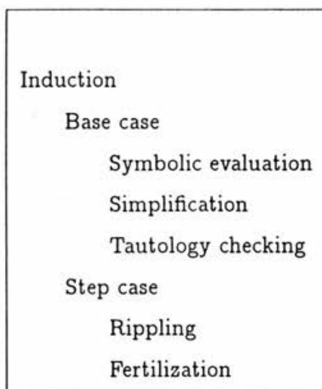

Figure 3–1: General structure of inductive proofs

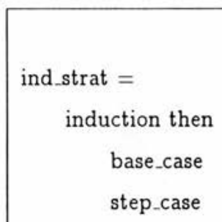

Figure 3–2: The ind_strat method

proofs. The methods it is composed of, induction, base_case and step_case, are discussed in the following sections. They are illustrated throughout using the proof of the associativity of +

$$\forall x, y, z. (x + y) + z = x + (y + z) . \tag{3.1}$$

Induction

The purpose of the induction method is to select a set of induction variables and an induction scheme for a given conjecture. Selecting an appropriate induction scheme is one of the main challenges in inductive theorem proving: It is one

of the first steps, yet it is crucial to the success of the proof planning. The technique the induction method uses to select an induction scheme is known as *recursion analysis.*

Recursion analysis is the rational reconstruction and extension of the heuristics of the induction technique of NQTHM (see Section 3.1.1) to select induction variables and schemes [Boyer & Moore 88, Stevens 88, Bundy *et al* 89]. Recursion analysis basically prefers induction variables which occur in the recursive position of the function or relation dominating them 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. For example (3.1), the available axioms are

$$\forall u, v.\ s(u) + v = s(u + v) \tag{3.2}$$

$$\forall u, v.\ s(u) = s(v) \leftrightarrow u = v\ , \tag{3.3}$$

i.e., the recursive part of the definition of $+$ and the replacement axiom for s. Each of the universally quantified variables x, y, z in the conjecture is analyzed with respect to these definitions. Both occurrences of the variable x are in the recursive position of $+$, while only one occurrence of the variable y and no occurrence of the variable z are in the recursive position of either axiom. The variable z is labeled *unsuitable* as an induction variable. The variable y is labeled as *flawed* because only one of its occurrences is in a recursive position. The variable x is labeled *unflawed*:

Variable	Definition	Recursion	Occurrences			Status
			total	unflawed	flawed	
x	$+$	$s(x)$	2	2	0	unflawed
y	$+$	$s(y)$	2	1	1	flawed
z	none	none	2	0	2	unsuitable

On the basis of this analysis, the induction method selects induction variables, preferring unflawed to flawed candidates.

In essence, recursion analysis is a look-ahead into the rewriting of the step case. If, for instance, we choose one-step induction on x for (3.1), the step case

becomes

$$(x + y) + z = x + (y + z)$$
$$\vdash$$
$$(s(x) + y) + z = s(x) + (y + z) \; ,$$

to which we can apply at least two rewrites based on the definition of $+$. If we choose one step induction on y, however, the step case is

$$(x + y) + z = x + (y + z)$$
$$\vdash$$
$$(x + s(y)) + z = x + (s(y) + z)$$

to which we can apply only one rewrite based on the definition of $+$, namely on the second occurrence of y.

This presentation of recursion analysis is an oversimplification in that recursion analysis also considers combinations of induction variables, combinations of recursion schemes as well as subsumption between them.

The main disadvantage of recursion analysis is that it can only find schemes that are dual to recursion schemes present in the conjecture. This is not always sufficient, particularly not in the case of synthesis, because the appropriate induction may depend on the recursion of the program to be synthesized, which is unknown at the outset. An alternative to recursion analysis that avoids this restriction, and also avoids the look-ahead into the step case, is the technique of *middle-out induction* (Section 5), which forms a central part of this thesis.

To summarize, the induction method applies recursion analysis to the input sequent. It succeeds if the analysis suggests a suitable induction scheme, and fails if all variables are labeled unsuitable. Its output are the base and step sequents for the selected induction scheme. The induction method also annotates the step sequents for the subsequent rippling method, which is described further down.

```
base_case =
    iterate(
        sym_eval,
        elementary
    )
```

Figure 3–3: The base_case method

```
sym_eval =
    iterate(
        equal,
        reduction,
        eval_def,
        existential
    )
```

Figure 3–4: The sym_eval method

Base Case

The base_case method is a compound method which iterates over a symbolic evaluation method sym_eval and a simplification and tautology-checking method elementary (see Figure 3–3).

Symbolic Evaluation The sym_eval method is again a compound method. It iterates over the submethods equal, reduction, eval_def and existential (see Figure 3–4).

The equal method uses any existing equality to rewrite all hypotheses and goals towards the alphabetically lower term.

```
step_case =
    ripple then
    fertilize
```

Figure 3–5: The step_case method

The reduction method rewrites the conclusion using a reduction rule. A reduction rule is a rewrite rule that removes a constant expression or wave rule (see below) where the wave front is a type constructor.

The eval_def method rewrites the conclusion using defining equations.

The existential method deals with existentially quantified base case proof obligations

Elementary The elementary method is a terminating method. It is a propositional tautology checker, but also does some simple non-propositional simplifications.

Step Case

In the step case of inductive proofs, the main objective is to manipulate the induction conclusion in such a way that the induction hypothesis can be exploited. The step_case method (see Figure 3–5) applies the ripple method to rewrite the induction conclusion, and then the fertilize method to exploit the induction hypothesis.

Rippling The ripple method embodies what is known as 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. To visualize rippling, following

[Bundy *et al* 93], imagine a loch (a Scottish lake) which reflects the induction hypothesis. The reflection, i.e., the induction conclusion, is not a perfect image of the induction hypothesis, because where there are induction variables in the hypothesis, there are induction terms in the conclusion. The terms that appear in the induction conclusion, but not in the induction hypothesis, are *wave fronts*. They are the ripples on the surface of the loch that spoil the reflection. Initially, the wave fronts immediately dominate the induction variables. The role of rippling is thus to move them outwards—just like ripples on a loch—until they leave behind a perfect reflection of the induction hypothesis. The rippling heuristic has been shown to terminate [Bundy *et al* 93].

We will represent wave fronts as boxes with holes. The holes are indicated by underlinings. Thus, for example, for the step case of the proof of (3.1), the induction method sets up the annotated sequent

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

If we remove the structure in the non-underlined parts of the boxes from the conclusion, we obtain what is called the *skeleton*, i.e., a copy of the induction hypothesis. The skeleton is thus the part of the induction conclusion we want to preserve.

Rippling now consists of applying annotated rewrite rules called *wave rules*. The annotation on the wave rule ensures that applying it will move the wave front up in the term tree of the induction conclusion, if the annotations in the rule match those of the conclusion. The (simplified) schematic format of a wave rule[1] that moves one wave front outwards is

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

The effect of applying such 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

[1] We use the symbol $:\Rightarrow$ as the rewrite arrow.

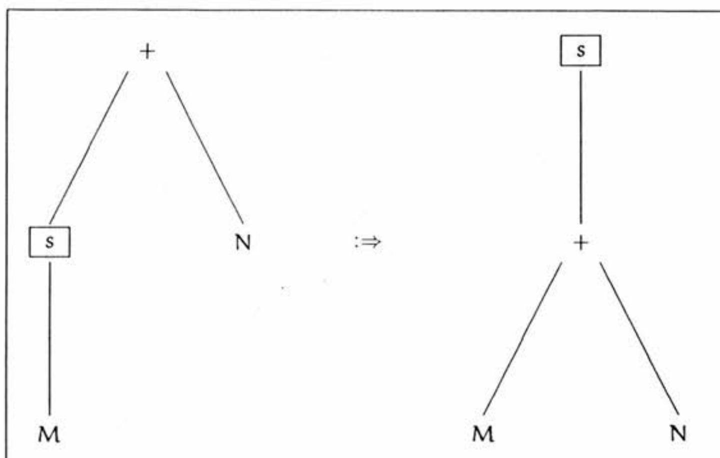

Figure 3–6: A wave rule for +

right-hand side, whose position is higher up the term tree. A simple example of a wave rule is

$$\boxed{s(\underline{M})} + N :\Rightarrow \boxed{s(\underline{M + N})},$$

which is based on the recursive definition of +. Here, S and T happen to be the same, i.e., the constructor s. Figure 3–6 shows this wave rule in terms of the term trees of the left- and right-hand sides, emphasizing the upward movement of the wave front. CL^AM generates wave rules automatically from the axioms, definitions, and lemmas it is given.

To apply a wave rule to a subexpression of the induction conclusion, the following pre-conditions [Bundy *et al* 93] must hold:

- The left-hand side of the wave rule must match the subexpression.
- The subexpression must contain at least one wave-front.
- Each wave front in the rule must match a wave front in the subexpression.

To illustrate the rippling process, we will work through the step case of (3.1). The wave rules we will use are:

$$\boxed{s(\underline{M})} + N \;:\Rightarrow\; \boxed{s(\underline{M + N})} \tag{3.4}$$

$$\boxed{s(\underline{M})} = \boxed{s(\underline{N})} \;:\Rightarrow\; M = N\,, \tag{3.5}$$

where M and N are free variables. These wave rules are derived from the definition of + and the replacement axiom for s. Rippling applies wave rule (3.4) once on the right side and twice on the left side and then applies wave rule (3.5) once:

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

The applications of rules (3.4) successively move the wave fronts outwards, and the application of rule (3.5) removes all wave fronts from the conclusion. The induction conclusion is now identical to the induction hypothesis. The method that exploits the induction hypothesis is called *fertilization*.

Before discussing fertilization, however, we present some variations on rippling. Rippling as presented so far is known as *rippling out*. It is an extension of the ripple-out heuristic developed in [Aubin 76]. We do not cover all variations of rippling, but only the ones that occur in subsequent sections. For a complete description of rippling, see [Bundy *et al* 93].

A first extension of rippling out is to allow *conditional rippling*. The format of a simple conditional wave rule is

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

A simple example of a set of conditional wave rules is

$$\begin{aligned}
X = H \rightarrow count(X, \boxed{H :: \underline{T}}) &\;:\Rightarrow\; \boxed{s(\underline{count(X, T)})} \\
X \neq H \rightarrow count(X, \boxed{H :: \underline{T}}) &\;:\Rightarrow\; count(X, T)
\end{aligned}$$

where count is a function which counts the number of occurrences of an element in a list.

When we apply a conditional wave rule, we must ensure that the condition holds. There are two ways this can be done: One is to try to prove the condition of the wave rule from the current hypotheses, the other is to introduce an appropriate case split. The method that introduces such case splits is called case_split. The strategy that CL^AM uses is to introduce the case split if the precondition of the wave rule is not already present among the hypotheses and if there is a complete set of rules for all conditions.

Until now, all wave fronts have been moving outwards. There are cases, however, where it is better to direct the wave fronts not to the top of the term tree of the induction variable, but towards a universally quantified variable. In the induction hypothesis, universally quantified non-induction variables act as free variables; thus, in the induction conclusion, we can wrap any amount of structure around their eigenvariable[2] counterparts. This type of rippling is called *rippling sideways*. It is often useful in program verification when dealing with accumulators. To ripple sideways, we need to allow a new type of wave rule which moves wave fronts off of one variable and onto another. Such wave rules are called *transverse* wave rules, as opposed to the wave rules we have seen so far, which are called *longitudinal* wave rules.

We must extend our annotation to be able to distinguish the direction in which a wave front is moving, and to be able to distinguish the eigenvariables that correspond to universally quantified variables in the induction hypothesis, which are called *sinks*. The direction of a wave front is indicated by an arrow

---

[2] *Eigenvariables* are new variables introduced by inference rules involving quantifiers. For instance, y is the eigenvariable of the rule

$$\frac{\Gamma \vdash \Delta, A[y/x], \Lambda}{\Gamma \vdash \Delta, \forall x : s.A, \Lambda}$$

The side condition of the rule, i.e., that y may not occur free in the conclusion, is known as the *eigenvariable condition*.

superscript on the box[3]. The arrow indicates the direction the wave front moves in the term tree. Sinks are annotated with $\lfloor\ \rfloor$.

The format of a simple transverse wave rule is

$$F[\boxed{S[\underline{U}]}^{\uparrow}, V] :\Rightarrow F[U, \boxed{T[\underline{V}]}^{\downarrow}] .$$

A simple example of a transverse wave rule is

$$rev(\boxed{H :: \underline{T}}^{\uparrow}, A) :\Rightarrow rev(T, \boxed{H :: \underline{A}}^{\downarrow}) \tag{3.6}$$

where rev is tail-recursive list reversal.

An example that requires rippling sideways is the following theorem on naive and tail-recursive list reversal

$$\forall k, l.\ app(naive_rev(k), l) = rev(k, l) .$$

Besides the transverse wave rule (3.6), we need the longitudinal wave rule for naive list reversal and the transverse wave rule based on the associativity of append

$$naive_rev(\boxed{H :: \underline{T}}^{\uparrow}) \ :\Rightarrow\ \boxed{app(\underline{naive_rev(T)}, H :: nil)}^{\uparrow} \tag{3.7}$$

$$app(\boxed{app(U, \underline{V})}^{\uparrow}, W) \ :\Rightarrow\ app(U, \boxed{app(V, \underline{W})}^{\downarrow}) . \tag{3.8}$$

The step case for induction on k is

$$\begin{array}{l} \forall l.\ app(naive_rev(t), l) = rev(t, l) \\ \vdash \\ app(naive_rev(\boxed{h :: \underline{t}}^{\uparrow}), \lfloor l \rfloor) = rev(\boxed{h :: \underline{t}}^{\uparrow}, \lfloor l \rfloor) . \end{array}$$

We apply wave rules (3.6) and (3.7) to obtain

$$app(\boxed{app(\underline{naive_rev(t)}, h :: nil)}^{\uparrow}, \lfloor l \rfloor) = rev(t, \left\lfloor \boxed{h :: \underline{l}}^{\downarrow} \right\rfloor) ,$$

---

[3]In the following, if there is no possibility of confusion because all wave fronts are moving outwards, the arrows are omitted. Similarly, sinks may be omitted.

and wave rule (3.8) to get

$$\mathsf{app}(\mathsf{naive_rev}(t), \left\lfloor \boxed{\mathsf{app}(h :: \mathsf{nil}, \underline{l})}^{\downarrow}) \right\rfloor = \mathsf{rev}(t, \left\lfloor \boxed{h :: \underline{l}}^{\downarrow} \right\rfloor) .$$

The unblocking method (see below) simplifies the left wave front, $\mathsf{app}(h :: \mathsf{nil}, l)$, to $h :: l$. We can now complete the proof by instantiating the universally quantified variable l in the induction hypothesis to $h :: l$ and appealing to the instantiated hypothesis.

A logical extension to rippling out and rippling sideways is *rippling in*. Rippling in is the reverse of rippling out—an inward-bound wave front is pushed further down the term tree. Rippling in is useful in connection with rippling sideways, when an inward-bound wave front introduced by a transverse wave rule needs to be pushed further down. Rippling in, however, is also useful in connection with weak fertilization (see below).

Rippling as portrayed so far applies to theorems containing universal quantifiers only, and cannot cope with existential quantifiers. To illustrate the problem of existential quantifiers, we use a variant of the associativity of $+$

$$\forall x, y, z. \exists w.\ w + z = x + (y + z) .$$

The induction conclusion for induction on x is

$$\forall y, z. \exists w.\ w + z = \boxed{s(\underline{x})} + (y + z) ,$$

which can be rippled to

$$\forall y, z. \exists w.\ w + z = \boxed{s(\underline{x + (y + z)})}$$

as in the earlier version of the associativity of plus. There, we then applied wave rule (3.4) twice to the left hand-side. This is no longer possible here, since we have an existentially quantified variable w instead of the wave term $\boxed{s(\underline{x})}^{\uparrow} + y$. Clearly, however, we need to somehow apply wave rule (3.4). We can do this by exploiting the fact that the existential variable w could actually stand for a wave term.

To allow wave rule (3.4) to apply here, we introduce existential versions of wave rules. A simple wave rule of the form

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

can be turned into an existential wave rule

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

The existential version of wave rule (3.4) for the example is thus

$$\exists \boxed{w}.\ \boxed{w} + z = \boxed{s(\underline{x})} + (y + z) :\Rightarrow \exists \underset{\text{dashed}}{\boxed{w'}}.\ \boxed{s(\underline{\boxed{w'}})} + z = \boxed{s(\underline{x})} + (y + z)\ .$$

In essence, what existential rippling does is to allow a wave rule to partially instantiate an existential variable to a wave term. To indicate that existential variables can be instantiated to wave terms, they are annotated with dashed boxes, called potential wave fronts.

Applying the existential wave rule to the appropriately annotated step case

$$\forall y, z.\exists \underset{\text{dashed}}{\boxed{w}}.\ \underset{\text{dashed}}{\boxed{w}} + z = \boxed{s(\underline{x + (y + z)})}$$

yields

$$\forall y, z.\exists \underset{\text{dashed}}{\boxed{w'}}.\ \boxed{s(\underline{\boxed{w'}})} + z = \boxed{s(\underline{x + (y + z)})}\ ,$$

and the rippling can now be completed by applying wave rule (3.5).

Finally, rippling sometimes terminates before the induction hypothesis can be exploited. If this happens, we say that the rippling is *blocked*. There are various techniques to *unblock* the rippling. They generally modify the induction conclusion in some way that makes a wave rule become applicable. The most common unblocking step is symbolic evaluation of a wave front.

The ripple method (see Figure 3–7) implements the rippling heuristic as described in the previous sections. It iterates over the methods wave, case_split and unblock.

The wave method covers all the types of rippling discussed above, i.e., rippling out, rippling sideways, rippling in, conditional rippling and existential rippling. The case_split introduces case splits to allow conditional wave rules to apply, and unblock covers unblocking.

```
ripple =
    iterate(
        wave,
        case_split,
        unblock
    )
```

Figure 3–7: The ripple method

```
fertilization =
    fertilization_strong or
    fertilization_weak
```

Figure 3–8: The fertilization method

Fertilization Fertilization (see Figure 3–8) is the method that exploits the induction hypothesis. If, after rippling, the wave front surrounds the entire induction conclusion, or has even disappeared, we use the method called strong fertilization. If the wave fronts do not yet surround the entire induction conclusion, but we can still exploit the induction hypothesis by applying it as a rewrite rule, we use the method called weak fertilization.

Fertilization is called *strong* if the entire induction hypothesis appears as a subterm of the induction conclusion. Often, as in the example (3.1), the induction conclusion is actually identical to the induction hypothesis. In this case, we can appeal directly to the induction hypothesis, and strong fertilization is a terminating method. If the induction hypothesis is a proper subterm of the induction conclusion, we rewrite the subterm to true and prove the residual conclusion.

In some cases, rippling terminates before strong fertilization is possible. Suppose that wave rule (3.5) were not available in the proof of (3.1). Then, rippling would have terminated with the sequent

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

Although we cannot appeal directly to the induction hypothesis, we can nevertheless exploit the induction hypothesis by using it as a rewrite rule. If we use the induction hypothesis as a rewrite rule from left to right to rewrite the wave hole on the left-hand side of the conclusion, we obtain

$$\begin{array}{l}(x+y)+z = x+(y+z)\\ \vdash \\ s(x+(y+z)) = s(x+(y+z))\ ,\end{array}$$

which is proved by the reflexivity of equality. Fertilization by using the induction hypothesis as a rewrite rule is called *weak* fertilization.

In some cases, the subgoal remaining after weak fertilization is not trivial, and requires additional rippling in. This is explained in detail in [Bundy *et al* 93].

The discussion of fertilization completes the overview of the proof planning methods of CL^AM.

3.2.3 Middle-Out Reasoning

Proof planning is a meta-level reasoning technique and thus differentiates between the problem (the object-level) and reasoning about the problem (the meta-level). Middle-out reasoning [Bundy *et al* 90a] extends the meta-level reasoning we have presented so far in that it allows the meta-level representation of object-level entities to contain meta-level variables, i.e., meta-variables can be used to represent unknown terms or formulae. This allows proof planning to proceed even though an object-level entity is not fully specified, thus postponing a

decision about the entity's real identity. [Hesketh 91] shows how middle-out reasoning can be used to synthesize tail-recursive programs from non-tail-recursive specifications and to generalize inductive theorems. [Madden *et al* 93] reports on a generalization of middle-out reasoning for program optimization.

The advantage of middle-out reasoning is that it allows us to get on with the proof planning even though some details are as yet unknown, and fill in the details later. The disadvantage is that the use of meta-variables introduces a larger degree of freedom in the subsequent search. Thus, there is often a need for greater search control when using middle-out reasoning.

In this work, we use middle-out reasoning for two purposes: To synthesize logic programs (Section 4) and to select induction schemas (Section 5). In synthesis, we use a meta-variable to represent an unknown program body in a verification proof; in induction scheme selection, we use meta-variables to represent the induction terms in the induction conclusion.

3.3 Summary

This section provided a brief introduction to automated inductive theorem proving and a more detailed introduction to proof planning for inductive proofs. As we show in the next section, our syntheses occur in the planning of inductive verification proofs. The proof planning methods presented here provide the foundation for our synthesis system and are used and extended in the following sections.

Chapter 4

Middle-Out Synthesis

This chapter illustrates how the technique of middle-out reasoning can be used to turn program verification into program synthesis. First, however, it deals with some preliminaries: It presents the type of program we synthesize, i.e., *pure logic programs*, and the way we verify them (Section 4.1). It then illustrates how logic program verifications are planned (Section 4.2), and how the middle-out reasoning in verification planning can be used for program synthesis. (Section 4.3).

4.1 Pure Logic Programs

The programming logic we use consists of the completions of a restricted subset of normal programs (see Section 2.1.1), which we call *pure logic programs*. Pure logic programs as presented here are similar to the pure logic programs in [Bundy *et al* 90b] and the logic descriptions in [Deville 90]. Formally, we define pure logic programs as follows: They are 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 of type $\overline{t}$, $A(\overline{x})$ 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 the clauses of the program.

An example of a pure logic program is

$$\begin{aligned} \forall x, l.\ \mathrm{member}(x, l) &\leftrightarrow \exists h, t.\ l = h{::}t \wedge (x = h \vee \mathrm{member}(x, t)) \qquad (4.1) \\ \forall i, j.\ \mathrm{subset}(i, j) &\leftrightarrow i = \mathrm{nil} \vee \\ &\qquad \exists h, t.\ i = h{::}t \wedge \mathrm{member}(h, j) \wedge \mathrm{subset}(t, j)\ . \end{aligned}$$

The predicate $\mathrm{member}(x, l)$ is true if x is a member of the list l, the predicate $\mathrm{subset}(i, j)$ 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).
```

We have chosen to synthesize pure logic programs because they are a suitable intermediate representation on the borderline 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.

Pure logic programs are very general: They capture the semantics of the logic programming languages presented in Section 2.1.1. They also capture 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. Thus, pure logic programs represent a kind of common denominator of pure logic programming languages.

We have seen that arbitrary first-order formulae can be regarded as logic programs (see Section 2.1.1). Thus, the question arises why we bother with synthesis at all. We do so mainly for two reasons. First, the Lloyd-Topor transformation, which is used to turn a first-order formula into a normal program, does not change the recursive structure of the formula in any way. Changing the recursion scheme of a program is often essential to achieve a gain in efficiency, and introducing an improved recursive structure is one of the main goals of program synthesis. Furthermore, the Lloyd-Topor transformation sometimes introduces negations, which in turn can produce floundering. For instance, if we transform the subset program

$$\forall i, j.\, \mathrm{subset}(i,j) \leftarrow (\forall x.\, \mathrm{member}(x,i) \rightarrow \mathrm{member}(x,j))$$

using the Lloyd Topor transformation, we obtain the program

$$\begin{aligned} \forall i, j.\, \mathrm{subset}(i,j) &\leftarrow \neg p(i,j) \\ \forall i, j.\, p(i,j) &\leftarrow \mathrm{member}(x,i) \wedge \neg \mathrm{member}(x,j)\,, \end{aligned}$$

which flounders in any but all-ground mode. Thus, if our aim is to synthesize programs that are less likely to flounder, the Lloyd-Topor transformation is not sufficient. Because we do not allow negation in pure logic programs, except in the form of inequality, the risk of floundering is minimized.

By synthesizing what is normally considered to be the semantics of an executable program, we also break down the formidable task of synthesis into two parts. First, we synthesize the basic structure of the algorithm in the form of a pure logic program. Once we have the basic structure, we can translate the pure logic program into a logic programming language of our choice and introduce non-logical primitives as desired.

Synthesizing pure logic programs has another, decisive advantage: The intended meaning of the program coincides with the logical meaning of the program. Thus, we can reason about pure logic programs within the well-understood

framework of predicate logic, and bring knowledge and experience in theorem-proving to bear. Reasoning about logic programs, in particular program correctness, is discussed in the following section.

4.1.1 Program Correctness

Establishing the correctness of a program entails showing that everything the program computes is described by the specification (partial correctness) and that everything the specification describes is computed by the program (completeness). One of the main advantages of logic programming languages is that, because the program statements are logical statements, we can reason directly with the program to establish properties such as partial correctness and completeness.

Unfortunately, however, there is a serious difficulty: When reasoning with impure programs, we must also capture the non-logical aspects of the programs. This means we can no longer reason solely within the framework of logic, and must find another suitable framework. This framework varies depending on the non-logical primitives used and the properties that are to be preserved, and is therefore necessarily language dependent and thus not generally applicable.

But even reasoning with pure programs presents some difficulties. This is because the intended meaning of a logic program does not necessarily coincide with the logical meaning of the program. Take, for instance, the member program

```
member(X, [X|_])
member(X, [_|T]) ← member(X, T)
```

and the goal

```
← member(1, [])
```

In the intended meaning of the program, the goal is false. However, this does not correspond to the logical meaning of the program. This is why the semantics

of logic programs is often taken not to be the logical meaning, but the least Herbrand model, the completion or some other appropriate semantics. These all correspond to some form of the *closed world assumption*. Depending on which semantics we choose for the program, however, there are many notions of program equivalence (see, for instance, [Maher 87, Lever 91]).

If we reason with completions, on the other hand, we can exploit the fact that the intended meaning and the logical meaning coincide, and we can reason purely within the framework of predicate logic. Also, there is only one notion of equivalence, namely logical equivalence.

Using pure logic programs, we can thus establish partial correctness and completeness by proving that program and specification are logically equivalent, i.e.,

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

where $A(\overline{x})$ and H are as in Section 4.1 and S is the specified relation. For instance, to verify the subset program with respect to the specification[1]

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

we prove, in the standard theory of lists,

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

using the definitions of member and subset (4.1).

Such verification proofs form the basis of our synthesis approach. The planning of verification proofs and how these verification proofs are turned into synthesis proofs is discussed in the following two sections.

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

4.2 Proof Planning for Reasoning about Logic Programs

Proof planning as presented in Section 3.2 is implemented in the proof planning system CLAM (Version 2.0) [Bundy *et al* 90c]. To use CLAM to reason about logic programs, some changes need to be made. The main issue is to adapt CLAM to the syntax of sorted first-order predicate logic with equality rather than that of the Martin-Löf type theory for which it was designed. The implementational issues involved in this adaptation are described briefly in Section 7.1.

On the conceptual level, most methods remain virtually unchanged. The one conceptual adaptation that is necessary has to do with logical equivalence, which occurs in first-order predicate logic but not in the type theory. CLAM for type theory therefore generates rewrite rules from equalities and implications. CLAM for first-order predicate logic, however, generates rewrite rules from equalities, implications and equivalences. More importantly, care must be taken when rewriting under equivalence in first-order predicate logic: Under equivalence, only equivalence-preserving rewrites are correct, which eliminates rewrite rules based on implications and also some existential versions of rewrite rules. The problem of existential rippling under equivalence is discussed in Section 6.2.

4.2.1 Planning Logic Program Verification Proofs

As explained above in Section 4.1.1, our verification proofs involve showing that the specification and the program are logically equivalent. The verification of the subset program, for instance, consisted of proving the conjecture

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

using the definitions of member and subset (4.1) and the axioms for lists.

To illustrate what is involved in planning verification proofs, we work through the verification of subset below.

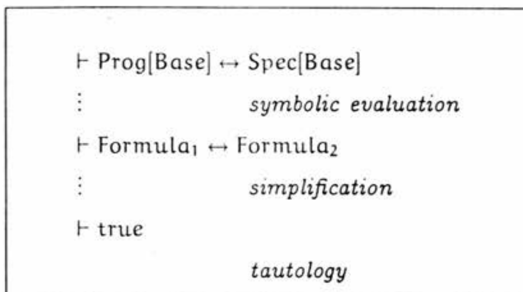

Figure 4–1: Schematic base case in verification

First, recursion analysis (see Section 3.2.2) suggests structural induction on the list i, since it occurs in the recursive position of both subset and member. This yields a base case where i is the empty list and a step case where i is a composite list.

Base case: The base case for induction on i is

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

Symbolic evaluation using the base cases of subset and member yields

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

which simplifies to

$$\vdash \mathrm{true}\ .$$

The base case is thus complete.

This sequence of symbolic evaluation, simplification and tautology checking is typical of the base cases of verification proofs. Figure 4–1 summarizes a schematic base case of a verification proof.

Step case: The annotated step case for induction on i is:

$$\begin{array}{l} \mathrm{subset}(t, j) \leftrightarrow (\forall x.\ \mathrm{member}(x, t) \rightarrow \mathrm{member}(x, j)) \\ \vdash \\ \mathrm{subset}(\boxed{h :: \underline{t}}, j) \leftrightarrow \big(\forall x.\ \mathrm{member}(x, \boxed{h :: \underline{t}}) \rightarrow \mathrm{member}(x, j)\big) \end{array}$$

For the rippling, we need the wave rules

$$\mathrm{subset}(\boxed{\mathrm{H} :: \underline{\mathrm{T}}}, \mathrm{L}) \;:\Rightarrow\; \boxed{\mathrm{member}(\mathrm{H}, \mathrm{L}) \wedge \underline{\mathrm{subset}(\mathrm{T}, \mathrm{L})}} \tag{4.2}$$

$$\mathrm{member}(\mathrm{X}, \boxed{\mathrm{H} :: \underline{\mathrm{T}}}) \;:\Rightarrow\; \boxed{\mathrm{X} = \mathrm{H} \vee \underline{\mathrm{member}(\mathrm{X}, \mathrm{t})}} \tag{4.3}$$

$$\boxed{\mathrm{P} \vee \underline{\mathrm{Q}}} \rightarrow \mathrm{R} \;:\Rightarrow\; \boxed{\mathrm{P} \rightarrow \mathrm{R} \wedge \underline{\mathrm{Q} \rightarrow \mathrm{R}}} \tag{4.4}$$

$$\forall \mathrm{x}.\, \boxed{\mathrm{P} \wedge \underline{\mathrm{Q}}} \;:\Rightarrow\; \boxed{\forall \mathrm{x}.\, \mathrm{P} \wedge \underline{\forall \mathrm{x}.\, \mathrm{Q}}} \tag{4.5}$$

$$\boxed{\mathrm{P} \wedge \underline{\mathrm{Q}}} \leftrightarrow \boxed{\mathrm{P} \wedge \underline{\mathrm{R}}} \;:\Rightarrow\; \mathrm{Q} \leftrightarrow \mathrm{R}\,. \tag{4.6}$$

While the first two wave rules are derived from member and subset, the remaining three are expressed in terms of logical connectives only. Such wave rules, of which there is a large number, are called logical wave rules. The problem of logical wave rules is discussed in Section 6.1.

The rippling of the subset example consists of consecutively applying wave rules (4.2) through (4.5) to the induction conclusion

$$\mathrm{subset}(\boxed{\mathrm{h} :: \underline{\mathrm{t}}}, \mathrm{j}) \leftrightarrow$$
$$\qquad \forall \mathrm{x}.\, \mathrm{member}(\mathrm{x}, \boxed{\mathrm{h} :: \underline{\mathrm{t}}}) \rightarrow \mathrm{member}(\mathrm{x}, \mathrm{j})$$

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

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

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

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

and then simplifying the wave front on the right-hand side (this is done by the unblocking method), so that we can apply wave rule (4.6)

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

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

Note that wave rule (4.6) is not equivalence-preserving, since it is based on an implication[2]. However, since we are no longer rewriting under the equivalence, but are rewriting the entire conclusion, wave rule (4.6) is applicable. It removes all wave fronts, and the induction conclusion is identical to the induction hypothesis. We apply strong fertilization (see Section 3.2.2), and the proof plan[3] is complete:

```
induction([h::t],[i:nat list]) then
   [ sym_eval( [eval_def([2], program(subset(1))),
              eval_def([1,2,1], program(member(1)))]) then
      simplify(true ↔ ∀x:nat. false → member(x,j), true) then
         tautology,
   ripple(wave([2],[program(subset(2)),left],[]) then
         wave([1,2,1],[program(member(2)),left],[]) then
         wave([2,1],[dist_impl_or_r, left],[]) then
         wave([1],[dist_all_and, left],[]) then
         unblock(simplify,
            ∀x:nat. x = h → member(x,j) ∧ ...,
            member(h,j) ∧ ... ) then
         wave([],[cnc_and_l, left],[])) then
      fertilize(strong,[_,[]])
   ] .
```

This sequence of rippling both sides of the equivalence, applying a wave

---

[2]There is potential for confusion here in that the rewrite goes from left to right, but the implication on which the rewrite is based goes from right to left.

[3]For details on the arguments of the methods, see [van Harmelen *et al* 93]. Note that *Periwinkle* splits CLAM's method elementary (see Section 3.2.2) into two separate methods, simplify and tautology.

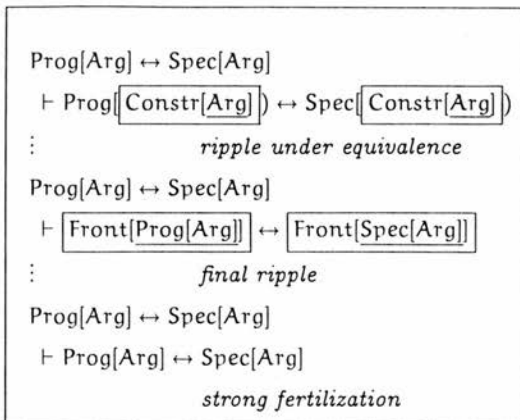

Figure 4–2: Schematic step case in verification

rule that removes the wave fronts and strong fertilizing is typical of verification proofs. Figure (4–2) summarizes a schematic step case of a verification proof.

4.3 From Verification to Synthesis

The concept of middle-out reasoning was presented in Section 3.2.3. In middle-out reasoning, object-level expressions are represented with meta-variables. With this technique, proof planning can progress even though an expression at the object-level is not yet known. In the course of the proof planning, the meta-variable becomes instantiated via unification, thus filling in the details of the initially unknown object-level expression.

In this section, we show how middle-out reasoning can be used to turn verification proof planning into program synthesis. This is achieved by planning the verification of a program while leaving the program unknown. The proof planning starts with a program whose body is represented with a meta-variable. In the course of the planning, the variable becomes instantiated to a program. The planning thus results both in an instantiation of the program body and in

a plan for the verification of that program. If the plan executes successfully, the synthesized program is partially correct and complete.

Representing the body of the program with a meta-variable obviously entails a loss of information, and this loss of information affects the proof planning. In the verification proof[4] in Section 4.2.1, there were a number of steps that crucially depended on the program, but also a number of steps that did not. In particular, the symbolic evaluation of subset in the base case and the ripple with the subset wave rule (4.2) depend on the program.

The main difference between the verification and the synthesis planning is that in verification, the two types of steps tend to be interleaved (see Figure 4–2). In synthesis, on the other hand, the part of the proof that does not depend on the program is planned first, and any step that depends on the program is postponed as long as possible. The reason for this is that, in synthesis, any step that depends on the program partially instantiates, i.e., commits the program. Thus, postponing such steps corresponds to a least commitment strategy.

To illustrate this, we redo the step case of the verification proof in Section 4.2.1, omitting any step that depends on the program. This rules out rippling with the subset wave rule (4.2). The rippling progresses as follows, using wave rules (4.3)–(4.5) and simplification.

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

[4]There is potential for confusion between the terms verification and synthesis on the object and meta-levels. In the following, *verification proof* is used to refer to the verification proof at the object level, *verification* to refer to the planning of a verification proof for a given program and *synthesis* to refer to the planning of a verification proof for a program to be synthesized.

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

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

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

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

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

The lack of wave rule for subset now prevents us from further rippling and strong fertilization (see Figure 4–2). However, we can apply weak fertilization (see Section 3.2.2). In weak fertilization, we use the induction hypothesis

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

as a rewrite rule. This results in

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

Now, there is nothing we can do that does 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. By appealing to the as yet uninstantiated program, we commit the program to correspond to this residual conclusion. Thus, the actual synthesis, i.e., the instantiation of the program body, takes place when the planner appeals to the program to justify the residual conclusion. The details of this are presented in the following section. Figure 4–3 shows schematically how a typical step case of a 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 follows a similar pattern. In verification, symbolic evaluation and simplification of specification and program usually yield a tautology that something is equivalent to itself (see Figure 4–1). In synthesis, only the specification is symbolically evaluated and simplified, and the residual conclusion again corresponds to part of the program (see Figure 4–4). This is illustrated in detail in the following section, which presents the synthesis of subset.

Figure 4–5, finally, contrasts typical verification and synthesis proof plans. Some of the actual methods differ slightly from verification to synthesis, e.g.,

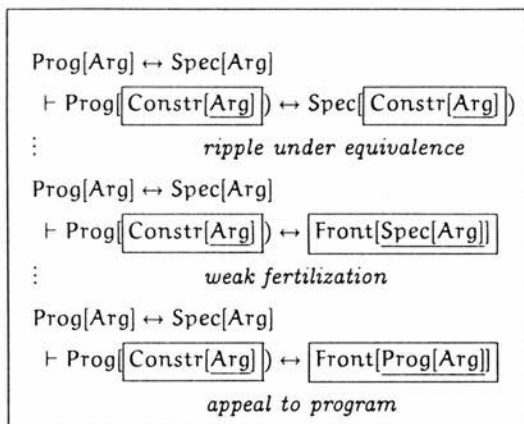

Figure 4–3: Schematic step case in synthesis

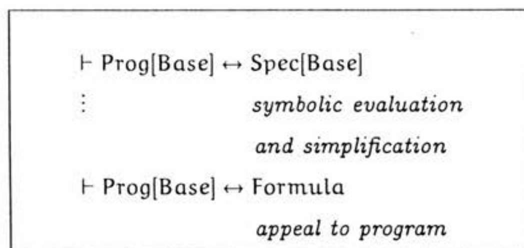

Figure 4–4: Schematic base case in synthesis

Verification	Synthesis
Induction	Induction
Step case: Rippling Strong fertilization	Step case: Rippling Weak fertilization Appeal to program
Base case: Symbolic evaluation Simplification Tautology	Base case: Symbolic evaluation Simplification Appeal to program

Figure 4–5: Overview of verification versus synthesis

rippling. This is mainly due to a need for more search control in synthesis, as is explained in the following sections.

4.4 An Example Synthesis

In this section, we illustrate the synthesis of the program we verified in Section 4.2.1. The by now familiar conjecture is

$$\forall i, j.\, \mathrm{subset}(i,j) \ \leftrightarrow \ (\forall x.\, \mathrm{member}(x,i) \rightarrow \mathrm{member}(x,j)) \ . \tag{4.7}$$

For synthesis, the subset program is represented as

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

where $\mathcal{P}$ is the meta-variable that stands for the program body.

Recursion analysis (now without the definition of subset), still suggests structural induction on the list i. The type of induction immediately determines the recursive structure of the body of the program: There is a base case where i is empty, and a step case where i consists of a head and a tail

$$\forall i, j.\, \mathrm{subset}(i,j) \ \leftrightarrow \ i = \mathrm{nil} \wedge \mathcal{B}(j) \vee \tag{4.8}$$

$$\exists h, t.\ i = h :: t \wedge \mathcal{S}(h, t, j)\ .$$

Each induction scheme has stored with it the corresponding recursive structure of the program, and the program body is unified with this structure. The problem of unifying such terms is discussed in Section 5.3.1.

Base case: The base case for induction on i is

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

Symbolic evaluation using the base case of member yields

$$\vdash \mathrm{subset}(\mathrm{nil}, j) \leftrightarrow (\forall x.\ \mathit{false} \rightarrow \mathrm{member}(x, j))\ ,$$

which simplifies to

$$\vdash \mathrm{subset}(\mathrm{nil}, j) \leftrightarrow \mathrm{true}\ . \tag{4.9}$$

After simplification, we are left with what will become the base case of the program. In the previous section, we explained that this is the part of the base case which would have been proved with the base case of the subset program in the verification. 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 definition (4.8), the synthesis method instantiates it appropriately

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

and simplifies it

$$\mathrm{subset}(\mathrm{nil}, j) \leftrightarrow \mathcal{B}(j)\ . \tag{4.10}$$

Finally, the residual conclusion (4.9) and the simplified program (4.10) are unified (see Section 5.3.1), which yields the instantiation $\lambda u.\ \mathrm{true}$ for $\mathcal{B}$. This step

completes the base case. The (normalized) partially instantiated program so far is thus

$$\begin{aligned}\forall i,j.\ \mathrm{subset}(i,j) \ \leftrightarrow\ & i = \mathrm{nil} \wedge \mathrm{true} \vee \\ & \exists h,t.\ i = h :: t \wedge \mathcal{S}(h,t,j)\ .\end{aligned} \tag{4.11}$$

Step case: The annotated step case for induction on i is:

$$\begin{aligned}&\mathrm{subset}(t,j) \leftrightarrow (\forall x.\ \mathrm{member}(x,t) \rightarrow \mathrm{member}(x,j))\\ &\vdash\\ &\mathrm{subset}(\boxed{h :: \underline{t}}, j) \leftrightarrow \big(\forall x.\ \mathrm{member}(x, \boxed{h :: \underline{t}}) \rightarrow \mathrm{member}(x,j)\big)\end{aligned}$$

In the previous section, we showed that rippling with wave rules (4.3)–(4.5), simplification and weak fertilization yielded the residual conclusion

$$\mathrm{subset}(\boxed{h :: \underline{t}}, j) \leftrightarrow \boxed{\mathrm{member}(h,j) \wedge \underline{\mathrm{subset}(t,j)}}\ . \tag{4.12}$$

In order to complete the planning, we must again establish that this follows from the program definition. The synthesis method instantiates definition (4.11) appropriately

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

and simplifies it

$$\mathrm{subset}(h :: t, j) \leftrightarrow \mathcal{S}(h,t,j)\ . \tag{4.13}$$

Unifying the conclusion (4.12) and the program (4.13) instantiates $\mathcal{S}$ with $\lambda u,v,w.\ \mathrm{member}(u,w) \wedge \mathrm{subset}(v,w)$. We thus obtain the (normalized) fully instantiated program

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

The proof plan is now complete:

```
induction([h::t],[i:nat list]) then
   [ sym_eval( [ eval_def([1,2,1], program(member(1)))]) then
       simplify(... ∀x:nat. false → member(x,i), ... true) then
           synthesis,
   ripple(wave([1,2,1],[program(member(2)),left],[]) then
           wave([2,1],[dist_impl_or_r, left],[]) then
           wave([1],[dist_all_and, left],[]) then
           unblock(simplify,
               ∀x:nat. x = h → member(x,j) ∧ ...,
               member(h,j) ∧ ... ) then
       weak_fertilize(right,↔,[2], _)] then
           synthesis
   ] .
```

Appealing to the program definition is embodied in a new proof planning method called synthesis. This method can be summarized as follows:

Method: Synthesis

1. Retrieve the definition being synthesized from the hypothesis list.
2. Instantiate and simplify the definition.
3. Unify the simplified definition and the conjecture.

The synthesis method is a terminating method, i.e., produces no output sequents. The code is listed in Appendix C.1.3.

4.5 Issues in Middle-Out Synthesis

4.5.1 Representation

Representation of Program Body

Middle-out reasoning uses meta-variables to represent as yet unknown object-level entities. Further proof planning then instantiates these meta variables, via unification, to object-level entities. Thus we must take care to ensure that the meta-variables become instantiated only to legal object-level terms. Ensuring this is non-trivial if meta-variables occur within the scope of quantifiers.

In middle-out synthesis, for instance, we represent the body of the program with a meta-variable. This body occurs within the scope of the universal quantifiers binding the arguments of the program. If we simply represent the body with a first-order metavariable

$$\forall x.\mathrm{prog}(x) \leftrightarrow \mathcal{B} \ ,$$

we will run into difficulties. Assume, for instance, that we have the sequent

$$\begin{array}{l} \forall x.\mathrm{prog}(x) \leftrightarrow \mathcal{B} \\ \vdash \\ \mathrm{prog}(x') \leftrightarrow x' = 0 \end{array}$$

where x' is an eigenvariable. To complete the proof, we need to apply universal elimination on x' to the hypothesis, so that we can apply to the axiom rule. This universal elimination, however, is problematic, since we cannot properly substitute x' for x in the program: The instantiation of $\mathcal{B}$ may actually contain a reference to x, but since $\mathcal{B}$ does not, we have no way of substituting correctly or avoiding capture unless $\mathcal{B}$ is fully instantiated. A straightforward solution is to use not first-order, but higher-order variables ranging over functions. Then

the body is represented as an application $\mathcal{B}(x)$:

$$\forall x.\mathrm{prog}(x) \leftrightarrow \mathcal{B}(x)$$
$$\vdash$$
$$\mathrm{prog}(x') \leftrightarrow x' = 0$$

$\mathcal{B}$ now no longer needs to contain any reference to x. When it becomes instantiated with a function, we can perform a β-reduction. With this representation, we obtain the same result, whether we instantiate $\mathcal{B}$ and then eliminate on x', or eliminate on x' and then instantiate $\mathcal{B}$. Applying universal elimination on x' now yields the correct answer:

$$\mathrm{prog}(x') \leftrightarrow \mathcal{B}(x')$$
$$\vdash$$
$$\mathrm{prog}(x') \leftrightarrow x' = 0 \ .$$

Applying the axiom rule then yields the unifier $\mathcal{B} = \lambda u.\ u = 0$, and $\mathcal{B}(x)$ reduces to $x = 0$, as desired.

An alternative implementation of middle-out reasoning is to formalize terms with holes. Such a calculus has been developed in [Talcott 93]. However, this formalization seems much more unwieldy than the approach above and has there fore not been used.

Representation of the Program Structure

Because of the duality between induction and recursion, the type of induction of the verification proof corresponds to the type of recursion of the program. Thus, in the example of the previous section, selecting one-step induction on the list i meant that the program would be recursive in i, with a base case where i was empty and a recursive case where i is decomposed into its head and its tail. As mentioned earlier, each induction scheme is stored with the dual program structure. Here, we discuss how to represent that recursive structure.

In Section 4.4, we represented the program structure as

$$\ldots \leftrightarrow i = \mathrm{nil} \wedge \mathcal{B}(j) \vee \exists h, t.\ i = h :: t \wedge \mathcal{S}(h, t, j)\ .$$

Originally, in [Kraan *et al* 93a], we suggested an alternative representation

$$\ldots \leftrightarrow \mathrm{i} = \mathrm{nil} \wedge \mathcal{B}(\mathrm{j}) \vee \exists \mathrm{h}, \mathrm{t}.\ \mathrm{i} = \mathrm{h} :: \mathrm{t} \wedge \mathcal{S}(\mathrm{h}, \mathrm{t}, \mathrm{j}, \mathrm{subset}(\mathrm{t}, \mathrm{j}))\ .$$

Here, the recursive case contains an explicit recursive call as an argument. This representation has the advantage of enabling us to derive a partially instantiated wave rule for subset

$$\mathrm{subset}(\boxed{\mathrm{H} :: \underline{\mathrm{T}}}, \mathrm{j}) \Rightarrow \boxed{\mathcal{S}(\mathrm{H}, \mathrm{T}, \mathrm{J}, \underline{\mathrm{subset}(\mathrm{T}, \mathrm{J})})}\ .$$

With this wave rule, the planning for verification and synthesis would be identical. Thus the synthesis of subset, for example, would follow exactly the verification of Section 4.2.1.

However, the disadvantages of this representation outweigh the advantages. The first disadvantage is that it is not a least commitment representation. The recursive call need not be of the given form at all—it could be of the form $\mathrm{subset}(\mathrm{t}, \mathrm{j}')$, where j' would be the result of some further computation on j (or vice versa). This would be the case, for instance, if j were an accumulator. Second, making the recursive call an explicit argument is slightly misleading, in that it in no way ensures that the recursive call will actually appear in the final program. It could simply be eliminated via a projection in unification. Finally, it is of a form that would require full higher-order unification. The simpler version, on the other hand, is the least commitment representation, and falls into a class of terms for which full higher-order unification is not required (see Section 5.3.1).

4.5.2 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 4.1). One way of doing this is to use a language which forces any instantiation of a meta-variable to be of the correct form. This is a difficult task, however, and would complicate our syntheses considerably. We have therefore chosen a more pragmatic solution. We do not enforce the syntactic

restrictions on pure logic programs via the meta-language, but via a mechanism outside the logic.

Once the synthesis has been completed, *Periwinkle* parses the program and marks any subformulae that violate the syntactic restrictions on pure logic programs. Each such subformula is then taken as the specification for an auxiliary synthesis, and the auxiliary syntheses are performed. In the initial program, any subformula for which an auxiliary synthesis was run is substituted with a call to the corresponding auxiliary predicate. Finally, all auxiliary predicates are added to the program. Note that an auxiliary synthesis may again require another auxiliary synthesis, etc. 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 specifies that m is the maximum element of the list l. The initial synthesized program is:

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

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

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

The auxiliary specification states that m is greater than any element of the list l. The final program, with the auxiliary predicate, is

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

4.5.3 Controlling Synthesis

One problem in the planning is deciding when to apply the synthesis method, i.e., the method that appeals to the program definition to justify the conclusion. For example, given the initial sequent

$$\forall \overline{arg}.\ Prog[\overline{arg}] \leftrightarrow \mathcal{P}(\overline{arg})$$
$$\vdash$$
$$\forall \overline{arg}.\ Prog[\overline{arg}] \leftrightarrow Spec[\overline{arg}]$$

the synthesis method could be applied immediately, which would instantiate the body of the program $\mathcal{P}(\overline{arg})$ to the specification $Spec[\overline{arg}]$. There is nothing in the logic itself to prevent this, i.e., the proof planning would succeed and the plan would execute. In general, however, this is not likely to be the kind of synthesis one is interested in. In order to synthesize programs that differ from the specification, therefore, we need to find some mechanism outside the logic to restrict the application of the synthesis method. This can be achieved by controlling the order in which methods are applied[5].

One heuristic to guide the application of the synthesis method would be to instantiate eagerly, i.e., apply the synthesis method as soon as possible. This generally leads to less efficient programs and a larger number of auxiliary syntheses. A typical example of this is the synthesis of delete. The conjecture is

$$\forall x, i, j.\ delete(x, i, j) \leftrightarrow (\exists k, l.\ append(k, l) = i \wedge append(k, x :: l) = j)$$

Thus, i is j with the element x deleted. With an eager application of the synthesis method, the method is applied immediately, which generates two auxiliary

[5]Also, constraints could be defined at the proof level, for instance to force tail recursion, as shown in [Hesketh 91].

syntheses for the two equations. The synthesized program

$$
\begin{array}{ll}
\forall x, i, j. & \mathsf{delete}(x, i, j) \leftrightarrow \\
& \exists k, l.\ \mathsf{deleteaux1}(k, l, i) \wedge \mathsf{deleteaux3}(k, x, l, j) \\
\forall k, l, i. & \mathsf{deleteaux1}(k, l, i) \leftrightarrow \\
& k = \mathsf{nil} \wedge \mathsf{deleteaux2}(l, i) \vee \\
& \exists k', k''.\ k = k' :: k'' \wedge \exists i'.\ k' :: i' = i \wedge \mathsf{deleteaux1}(k'', l, i') \\
\forall l, i. & \mathsf{deleteaux2}(l, i) \leftrightarrow l = i \\
\forall k, x, l, j. & \mathsf{deleteaux3}(k, x, l, j) \leftrightarrow \\
& k = \mathsf{nil} \wedge \mathsf{deleteaux4}(x, l, j) \vee \\
& \exists k', k''.\ k = k' :: k'' \wedge \exists j'.\ k' :: j' = j \wedge \mathsf{deleteaux3}(k'', x, l, j') \\
\forall x, l, j. & \mathsf{deleteaux4}(x, l, j) \leftrightarrow x :: l = j
\end{array}
$$

is quadratic in complexity. The auxiliary predicates $\mathsf{deleteaux1}$ and $\mathsf{deleteaux3}$ both represent a form of the append relation.

An alternative heuristic is to postpone the application of the synthesis method as long as possible, i.e., until no other method applies. This tends to generate simpler, more efficient programs with fewer auxiliary syntheses. For the delete example, this strategy yields the linear program

$$
\begin{array}{ll}
\forall x, i, j. & \mathsf{delete}(x, i, j) \leftrightarrow \\
& i = \mathsf{nil} \wedge j = x :: \mathsf{nil} \vee \\
& \exists i', i''.\ i = i' :: i'' \wedge j = \mathsf{nil} \wedge \mathsf{false} \vee \\
& \exists i', i''.\ i = i' :: i'' \wedge \exists j', j''.\ j = j' :: j'' \wedge \\
& \qquad (x = j' \wedge i' :: i'' = j'' \vee i' = j' \wedge \mathsf{delete}(x, i'', j''))\ .
\end{array}
$$

The main reason why the latter approach tends to produce more efficient programs is that more work is done at synthesis time, leaving less work to be done at runtime. This is a well-known tradeoff. The syntheses in Appendix A were all done using the strategy that delays the application of the synthesis method as long as possible.

4.6 Comparison of Middle-Out Synthesis with Other Approaches

Our approach to logic program synthesis automatically synthesizes truly relational, i.e., mode-insensitive logic programs. In this section, we compare our work to the approaches presented in Section 2.2.2 that synthesize relational programs and are in the course of being automated [Wiggins 92, Wiggins 93] or are semi-automatic [Lau & Prestwich 88a, Lau & Prestwich 88b, Lau & Prestwich 90].

4.6.1 Comparison with *Whelk*

The main emphasis of the *Whelk* project is to develop a logic in which relational programs can be synthesized via proofs-as-programs-style extraction (see Section 2.2.2). 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 following observation demonstrates how the two approaches are related. In our approach, we prove

$$\forall \overline{args}.\ \mathsf{Prog}(\overline{args}) \leftrightarrow \mathsf{Spec}(\overline{args})$$

where the definition of Prog is unknown. This is similar to proving the higher-order specification

$$\exists \mathsf{Prog}.\forall \overline{args}.\ \mathsf{Prog}(\overline{args}) \leftrightarrow \mathsf{Spec}(\overline{args})$$

constructively, since a constructive proof requires showing how a witness for an existentially quantified variable can be constructed. This corresponds closely to the meaning of

$$\vdash \forall x.\ \partial(\mathsf{Spec}(x))\ ,$$

which is defined as

$$\vdash \forall x^{*}.\, Spec(x)^{*} \leftrightarrow Prog(x^{*}) \; ,$$

where Prog is the synthesized program (see Section 2.2.2).

The difference between the two approaches thus lies more in their emphasis. While the *Whelk* project has focussed 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., sorted first-order predicate logic, and have taken a perhaps pragmatic approach by using middle-out reasoning for synthesis and by ensuring executability via 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 CLAM is currently being adapted to plan proofs in the *Whelk* logic. Many of the techniques developed here will be directly applicable, in particular middle-out induction (see Section 5) and extensions to rippling such as generating logical wave rules and unrolling (see Section 6). On the other hand, results in the *Whelk* logic could be used to ensure executability at the meta-level without extra-logical means, and thus improve our handling of auxiliary syntheses. To illustrate how this could work, we take the subset example as it would be expressed in CLAM for *Whelk*. The following sequent is the one which would correspond to the situation prior to weak fertilization

$$\begin{array}{l} \partial(subset(t,j)) \\ \vdash \\ \partial(\boxed{member(h,j) \wedge \underline{subset(t,j)}}) \; . \end{array}$$

The decidability operator ∂ now prevents weak fertilization. Assuming, however, that there is a wave rule for ∂ of the form

$$\partial(\boxed{P \wedge \underline{Q}}) :\Rightarrow \boxed{\partial(P) \wedge \underline{\partial(Q)}} \; ,$$

we could ripple the conclusion to

$$\partial(\mathrm{subset}(t,j))$$
$$\vdash$$
$$\boxed{\partial(\mathrm{member}(h,j)) \wedge \underline{\partial(\mathrm{subset}(t,j))}}\ .$$

Now fertilization is possible, which leaves us with the subgoal

$$\vdash \partial(\mathrm{member}(h,j))\ .$$

Unless $\partial(\mathrm{member}(h,j))$ is a lemma known to the system, this subgoal would force us to prove that member is decidable and thus synthesize a program for it.

4.6.2 Comparison with Lau and Prestwich

The system of Lau and Prestwich [Lau & Prestwich 88a, Lau & Prestwich 88b, Lau & Prestwich 90] is a semi-automatic, unfold/fold-based approach to logic-program synthesis (see Section 2.2.2). It synthesizes partially correct, but not necessarily complete programs. In particular, the system synthesizes programs such that

$$\forall \overline{args}.\ \mathrm{Prog}(\overline{args}) \leftrightarrow \mathrm{Spec}(\overline{args})$$
$$\vdash \forall \overline{args}.\ \mathrm{Prog}(\overline{args}) \leftarrow \mathrm{Body}(\overline{args})\ . \tag{4.14}$$

By contrast, our system synthesizes partially correct and complete programs. In particular, we synthesize programs such that

$$\forall \overline{args}.\ \mathrm{Prog}(\overline{args}) \leftrightarrow \mathrm{Body}(\overline{args})$$
$$\vdash \forall \overline{args}.\ \mathrm{Prog}(\overline{args}) \leftrightarrow \mathrm{Spec}(\overline{args})\ .$$

Using the transitivity of $\leftrightarrow$, this is equivalent to proving

$$\forall \overline{args}.\ \mathrm{Prog}(\overline{args}) \leftrightarrow \mathrm{Spec}(\overline{args})$$
$$\vdash \forall \overline{args}.\ \mathrm{Prog}(\overline{args}) \leftrightarrow \mathrm{Body}(\overline{args})\ ,$$

which is stronger than (4.14) in that it guarantees both partial correctness and completeness.

Lau and Prestwich solve a synthesis problem by decomposing it top-down into subproblems until the subproblems are easily solved. The synthesized program is then composed bottom-up from the solutions of the subproblems. User interaction is required to limit the search space, initially by specifying the desired recursive calls of the program, then by deciding which subproblems to solve if there is a choice.

We have, in fact, borrowed the subset example used throughout this section from [Lau & Prestwich 88a]; we can thus compare the two syntheses.

The input to the system of Lau and Prestwich are the subset specification

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

the definition of member[6] , and a goal specifying the unfold/fold problem

$$\mathrm{FOLD}(subset(h::t,j), \{subset(t,k)\}) \ , \qquad (4.15)$$

where $subset(h :: t,j)$ is the head of the program to be synthesized and $\{subset(t,k)\})$ is the set of desired recursive calls. The representation here differs from that in [Lau & Prestwich 88a] in that it omits their second argument, an initially uninstantiated variable representing the body of the program to be synthesized.

All definitions are brought into a normal form (right-hand sides consist of conjunctions of literals, disjunctions of literals, an existentially or a universally quantified literal). For subset, this yields

$$subset(i,j) \ \leftrightarrow \ \forall x.\, sys(x,i,j) \qquad (4.16)$$

$$sys(x,i,j) \ \leftrightarrow \ \neg member(x,i) \vee member(x,j) \qquad (4.17)$$

6

$$member(x, x::t) \leftarrow$$
$$member(x, h::t) \leftarrow member(x,t)$$
$$\neg member(x, h::t) \leftarrow \neg x = h \wedge \neg member(x,t)$$

The *definition* strategy selects a definition $\mathrm{p} \leftrightarrow \mathrm{q}$. It then uses the `if` part $\mathrm{p} \leftarrow \mathrm{q}$ to unfold, and the `only if` part $\mathrm{q} \leftarrow \mathrm{p}$ to fold. Before folding, unfold/fold subgoals for q need to be solved.

An application of the *definition* strategy to (4.15) using (4.16) yields the unfold formula

$$\mathrm{subset}(\mathrm{h}::\mathrm{t},\mathrm{j}) \leftarrow \forall \mathrm{x}.\ \mathrm{sys}(\mathrm{x},\mathrm{h}::\mathrm{t},\mathrm{j})\ , \tag{4.18}$$

the fold formula

$$\mathrm{sys}(\mathrm{x},\mathrm{t},\mathrm{k}) \leftarrow \mathrm{subset}(\mathrm{t},\mathrm{k}) \tag{4.19}$$

and the subproblem

$$\mathrm{FOLD}(\mathrm{sys}(\mathrm{x},\mathrm{h}::\mathrm{t},\mathrm{j}),\{\mathrm{sys}(\mathrm{x},\mathrm{t},\mathrm{k})\})\ . \tag{4.20}$$

Applying the *definition* strategy again to (4.20) using (4.17), the unfold formula is

$$\mathrm{sys}(\mathrm{x},\mathrm{h}::\mathrm{t},\mathrm{j}) \leftarrow \neg\mathrm{member}(\mathrm{x},\mathrm{h}::\mathrm{t}) \vee \mathrm{member}(\mathrm{x},\mathrm{j}) \tag{4.21}$$

and the fold formula is

$$\neg\mathrm{member}(\mathrm{x},\mathrm{t}) \vee \mathrm{member}(\mathrm{x},\mathrm{k}) \leftarrow \mathrm{sys}(\mathrm{x},\mathrm{t},\mathrm{k})\ . \tag{4.22}$$

Possible subproblems are

$$\mathrm{FOLD}(\neg\mathrm{member}(\mathrm{x},\mathrm{h}::\mathrm{t}),\{\neg\mathrm{member}(\mathrm{x},\mathrm{t})\}) \tag{4.23}$$

and

$$\mathrm{FOLD}(\mathrm{member}(\mathrm{x},\mathrm{j}),\{\mathrm{member}(\mathrm{x},\mathrm{k})\}) \tag{4.24}$$

It is now up to the user to determine which of these subproblems are to be solved. Here, we select both.

We obtain a solution for (4.23) by applying the *implication* strategy. The *implication* strategy exploits known recursive implications. In this case, it uses the implication

$$\neg\mathrm{member}(\mathrm{x},\mathrm{h}::\mathrm{t}) \leftarrow \neg\mathrm{x} = \mathrm{h} \wedge \neg\mathrm{member}(\mathrm{x},\mathrm{t})$$

We obtain a solution for (4.24) by applying the *match* strategy. The *match* strategy solves trivial folding problems where the desired recursive calls unify directly with the head of the folding problem.

Since there are no remaining subproblems, we proceed to compose the solution. First, we unfold the unfold formula (4.21) using the solutions of (4.23) and (4.24)

$$\mathrm{sys}(x, h :: t, j) \leftarrow \neg x = h \wedge \neg \mathrm{member}(x, t) \vee \mathrm{member}(x, j) \tag{4.25}$$

and bring this into conjunctive normal form

$$\mathrm{sys}(x, h :: t, j) \leftarrow (\neg x = h \vee \mathrm{member}(x, j)) \wedge (\neg \mathrm{member}(x, t) \vee \mathrm{member}(x, j)) \,. \tag{4.26}$$

We fold using the fold formula (4.22)

$$\mathrm{sys}(x, h :: t, j) \leftarrow (\neg x = h \vee \mathrm{member}(x, j)) \wedge \mathrm{sys}(x, t, j)$$

to solve problem (4.20).

Next, to solve (4.15), we fold this solution into the unfold formula (4.18)

$$\mathrm{subset}(h :: t, j) \leftarrow \forall x.\, (\neg x = h \vee \mathrm{member}(x, j)) \wedge \mathrm{sys}(t, j)$$

and then fold using the fold formula (4.19)

$$\mathrm{subset}(h :: t, j) \leftarrow \forall x.\, (\neg x = h \vee \mathrm{member}(x, j)) \wedge \mathrm{subset}(t, j) \,.$$

To eliminate the quantifier, we distribute it

$$\mathrm{subset}(h :: t, j) \leftarrow \forall x.\, (\neg x = h \vee \mathrm{member}(x, j)) \wedge \forall x.\, \mathrm{subset}(t, j) \tag{4.27}$$

and (presumably) simplify to

$$\mathrm{subset}(h :: t, j) \leftarrow \mathrm{member}(h, j) \wedge \mathrm{subset}(t, j) \,,$$

thus obtaining a solution for (4.15).

Though at first glance this synthesis seems not to be very similar to ours, there is a close relationship between the two. Setting the initial fold problem

corresponds to selecting the type of induction. The initial unfolding then corresponds to applying induction, and the last folding to fertilization. The remaining decomposition and composition steps all correspond to rippling.

The similarities between the two approaches are somewhat obscured by two things: First, the need for a normal form in the Lau and Prestwich approach, which often requires the introduction of new definitions such as sys, and second, the strict decomposition of a problem into subproblems, even if they can be solved trivially. Thus, in the subset example, of the four folding problems, only (4.15) corresponds to an induction and thus a synthesis in our approach. Problem (4.20) stems from the introduction of the new predicate sys, which is not necessary in our approach. Problems (4.23) and (4.24) are solved by using existing knowledge. In middle-out synthesis, the former is "solved" by the application of the member wave rule, the latter by fertilization.

The approach of Lau and Prestwich requires a substantial amount of rearranging before the fold steps in the composition phase. Examples of this are bringing (4.25) into conjunctive normal form and distributing quantifiers in (4.27). How the current formula needs to be rearranged to enable folding depends on the normal form of the definition which was used in the corresponding unfold step. Thus, for example, (4.25) is brought into conjunctive normal form because the definition of sys was in disjunctive normal form. In middle-out synthesis, this rearrangement of unfolded formulae to allow folding corresponds to the application of logical wave rules to allow fertilization. Thus, we can rely on the rippling annotation to rearrange formulae, and do not need normal forms.

As explained above, one of the major differences between the approaches is that Lau and Prestwich synthesize partially correct, but not necessarily complete programs, whereas we insist on partial correctness and completeness. Not requiring completeness has the advantage that the body of the program being synthesized can always be strengthened. This occurs, for instance, in the problem

$$\mathsf{FOLD}(\mathsf{subset}(i, h::t), \{\mathsf{subset}(k, t)\}) ,$$

which can be solved, given appropriate user guidance, in the system of Lau and Prestwich, but is currently beyond the capabilities of *Periwinkle*. Although strengthening allows greater flexibility in synthesis, it also increases the search space considerably, which, in Lau and Prestwich, translates into a need for user interaction.

In conclusion, we believe that the approach of Lau and Prestwich could be improved by abolishing the need for normal forms and exploiting rippling to guide the folding, as middle-out synthesis does. On the other hand, our approach could well benefit from the strategies of Lau and Prestwich which strengthen formulae to allow folding. This may well be essential when synthesizing larger, more complex programs, or synthesizing programs from partial specifications. We could then use middle-out synthesis in a partial correctness proof, rather than a partial correctness and completeness proof. If desired, we could prove the program complete in a separate, second step.

4.7 Summary

This section presented our approach to logic program synthesis and compared it with similar systems. We have shown how middle-out reasoning can be used to turn verification proofs into synthesis proofs, and how proof planning can automate synthesis. This approach to logic program synthesis is a new application of proof planning, and has proved successful. Proof planning, and in particular heuristics like rippling, have made it possible to achieve full automation of the synthesis process for a number of examples.

Chapter 5

Middle-Out Induction

In the previous chapter, we illustrated how middle-out reasoning can be used to synthesize logic programs. In this chapter, we present a further application. Middle-out reasoning can be used to postpone the first, crucial step in the planning of inductive proofs, namely the selection of an induction scheme. This was first suggested in [Bundy *et al* 90a], but had not been elaborated or implemented.

5.1 Outline

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, Bundy *et al* 89], which was described in Section 3.2.2. However, 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 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.

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 $\times$, $+$ and $<$, recursion analysis cannot find the appropriate induction, which is induction on y, where the successor of y is $y + x$.

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

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

where qr remains undefined. Since the conjecture is universally quantified over all its arguments, we can choose any of them as induction variables. Thus, instead of having to choose among x and y, we can choose among x, y, q and r as induction variables. Hence, recursion analysis stands a somewhat better chance of success. For this conjecture, recursion analysis indeed suggests an appropriate induction, namely 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

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

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

Middle-out reasoning (see Section 3.2.3) can provide such a technique. Using meta-variables, we can set up a schematic step case representing many possible inductions. This is achieved by using the 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 the rippling is complete and fertilization has taken place, the meta-variables are fully instantiated and correspond to a (possibly not unique) type of induction.

Schematically, this works as follows. For a conjecture

$$\forall x, y.\, p(x, y) \,,$$

we would represent the step case as

$$p(x, y) \vdash p(\mathcal{C}(x), \mathcal{D}(y)) \,.$$

The potential induction variables are x and y, and the meta-variables $\mathcal{C}$ and $\mathcal{D}$ represent the constructors applied to them in the induction conclusion. If one of the variables turns out not to be an induction variable, the meta-variable simply becomes instantiated to the identity function. Since the meta-variables stand for the constructors, i.e., the initial wave fronts, we annotate the terms by putting the meta-variables in wave fronts and their arguments in wave holes.

$$p(x, y) \vdash p(\boxed{\mathcal{C}(\underline{x})}, \boxed{\mathcal{D}(\underline{y})}) \,. \tag{5.1}$$

The dashed notation indicates that the wave fronts are only potential—if a variable turns out not to be an induction variable and the meta-variable collapses into the identity function, the wave front is spurious.

To prove (5.1), suppose that we have the wave rule

$$p(\boxed{s(\underline{U})}, V) :\Rightarrow p(U, V) \,.$$

To apply it, we need to unify its left-hand side with the conclusion from 5.1

$$p(\boxed{s(\underline{U})}, V) \stackrel{?}{=} p(\boxed{\mathcal{C}(\underline{x})}, \boxed{\mathcal{D}(\underline{y})}) \,.$$

Unification is discussed below in Section 5.3.1. This unification succeeds and yields the substitutions

$$\{\lambda u.\, s(\mathcal{C}'(u))/\mathcal{C},\ \mathcal{C}'(x)/U,\ \mathcal{D}(x)/V\} \,.$$

The interesting substitution is that of $\mathcal{C}$, since it forms part of the induction scheme. Applying the substitution to the right-hand side of the wave rule and normalizing gives us

$$p(\boxed{\mathcal{C}'(\underline{x})}, \boxed{\mathcal{D}(\underline{y})}) ,$$

and applying the rewrite to the conclusion yields

$$p(x, y) \vdash p(\boxed{\mathcal{C}'(\underline{x})}, \boxed{\mathcal{D}(\underline{y})}) .$$

Strong fertilization is possible when the induction hypothesis and the induction conclusion are unified. In the example, this unification yields the substitution

$$\{\lambda u.u/\mathcal{C}',\ \lambda u.u/\mathcal{D}\} .$$

The meta-variables have now been fully instantiated. Propagating the substitutions to the original step case (5.1) and normalizing shows that we have

$$p(x, y) \vdash p(\boxed{s(\underline{x})}, y) ,$$

which corresponds to structural induction on x.

Once the meta-variables have been fully instantiated in this way, we need to make sure that the induction is a valid one, i.e., that the induction ordering is well-founded. If, for instance, we had unified the hypothesis and the conclusion in (5.1), both $\mathcal{C}$ and $\mathcal{D}$ would have been instantiated to the identity function, which is certainly not a valid induction.

The most general approach would be to prove that the order is in fact well-founded. However, this is a difficult task, and beyond the scope of this thesis. We have therefore restricted ourselves to a simpler task, 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 scheme is in fact valid, we can set up the corresponding base case(s) and complete the proof using the standard proof planning methods (see Section 3.2).

Middle-out induction has two main advantages over recursion analysis: First, it is a more general approach. It can find an appropriate induction 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 issues in middle-out induction: First, it requires some kind of higher-order unification. Second, rippling is no longer terminating. These issues are discussed in Section 5.3. The following section contains an example uniting the two uses of middle-out reasoning, for synthesis and for the selection of induction schemes.

5.2 An Example Synthesis with Middle-Out Induction

The example we present is the synthesis of a predicate that is true whenever a number is even. The conjecture is

$$\forall x.\, \mathrm{even}(x) \leftrightarrow (\exists y.\, \mathrm{double}(y) = x) \ ,$$

where double is defined as

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

The wave rules for double and the replacement axiom of equality for s are

$$\mathrm{double}(\boxed{s(\underline{U})}) \ :\Rightarrow \ \boxed{s(s(\underline{\mathrm{double}(U)}))} \tag{5.2}$$

$$\boxed{s(\underline{U})} = \boxed{s(\underline{V})} \ :\Rightarrow \ U = V \ . \tag{5.3}$$

The schematic step case is

$$\mathrm{even}(x) \leftrightarrow (\exists y.\, \mathrm{double}(y) = x) \tag{5.4}$$

$$\vdash$$

$$\mathrm{even}(\boxed{\mathcal{C}(\underline{x})}) \leftrightarrow \left(\exists y.\, \mathrm{double}(\boxed{\underline{y}}) = \boxed{\mathcal{C}(\underline{x})}\right) \ .$$

where $\mathcal{C}$ is the meta-variable standing for the constructor applied to the potential induction variable. The potential wave front around the existential variable y indicates that it can be involved in the rippling (see Section 3.2.2 and Section 6.2). 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 as we did in Section 4.

We can apply an existential version of wave rule (5.2) to the induction conclusion. This instantiates the existentially quantified variable y with $s(y')$, where y' is a new existentially quantified variable. Applying (5.2) yields the conclusion

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

Applying wave rule (5.3) twice results in

$$\begin{array}{rcl} \mathrm{even}(\boxed{s(\mathcal{C}'(\underline{x}))}) & \leftrightarrow & \left(\exists y'.\ \boxed{s(\underline{\mathrm{double}(y')})} = \mathcal{C}'(\underline{x})\right) \\ \mathrm{even}(\boxed{s(s(\mathcal{C}''(\underline{x})))}) & \leftrightarrow & \left(\exists y'.\ \mathrm{double}(y') = \mathcal{C}''(\underline{x})\right) . \end{array}$$

The applications of wave rule (5.3) partially instantiate $\mathcal{C}$ to $\lambda u.\ s(s(\mathcal{C}''(u)))$.

We can now weak fertilize, i.e., apply the induction hypothesis (5.4) as a rewrite rule (see Section 3.2.2). This leaves us with

$$\mathrm{even}(s(s(x))) \leftrightarrow \mathrm{even}(x) .$$

Weak fertilization has also instantiated $\mathcal{C}''$ to $\lambda u.\ u$ and thus $\mathcal{C}$ to $\lambda u.\ s(s(u))$. Hence, the induction scheme is two-step induction, and the program structure is

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

We can now set up the base cases,

$$\vdash \mathrm{even}(0) \leftrightarrow \exists y.\ \mathrm{double}(y) = 0$$

and

$$\vdash even(s(0)) \leftrightarrow \exists y.\, double(y) = s(0)\ .$$

These and the step case are completed according to the general schema in Section 4, via symbolic evaluation, simplification and appealing to the program.

Middle-out induction is implemented as a strategy, i.e., a method that applies other methods. It can be summarized as follows:

Method: Mor_induction

1. Set up a schematic step case.
2. Ripple and fertilize.
3. Retrieve the induction/recursion scheme.

The output of the mor_induction method are the base and the post-fertilization sequents. The code is listed in Appendix C.1.1.

5.3 Issues in Middle-Out Induction

5.3.1 Unification

Given that we use higher-order meta-variables in our middle-out reasoning, we are confronted with the problem of higher-order unification, which is semi-decidable in general. 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 75], or one restricts oneself to a subset of higher-order terms which is tractable, e.g., *higher-order patterns*. The former approach has been taken, for instance, by [Hesketh 91, Ireland 92, Madden *et al* 93]. The latter approach is taken here.

Higher-order patterns are expressions whose free variables have no arguments other than bound variables. The class of higher-order patterns was first invest-

igated 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 unique most general unifier of unifiable terms. Also, the unification of two higher-order patterns is again a higher-order pattern. The algorithm for higher-order pattern unification of [Nipkow 93] is given in Appendix E. [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 main reason why we have chosen to restrict ourselves to higher-order patterns for the terms in which we use higher-order meta-variables is that 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.

Exploiting Higher-Order Pattern Unification for Middle-Out Induction

In this section, we discuss how higher-order pattern unification is exploited for middle-out induction and synthesis by choosing an appropriate representation.

Suppose we have a conclusion

$$\mathrm{p}(\fbox{$\mathcal{C}(\underline{\mathrm{x}})$})$$

and a wave rule

$$\mathrm{p}(\fbox{$\mathrm{s}(\underline{\mathrm{U}})$}) :\Rightarrow \fbox{$\mathrm{s}(\underline{\mathrm{p}(\mathrm{U})})$}\ . \tag{5.5}$$

To apply the wave rule, we need to unify the conclusion and the left-hand side of the wave rule. To capture the notion that the potential induction variable x is bound in the conclusion, we represent the (unannotated) conclusion

$$\mathrm{p}(\mathcal{C}(\mathrm{x}))$$

as

$$\lambda \mathrm{x}.\ \mathrm{p}(\mathcal{C}(\mathrm{x}))\ .$$

The (unannotated) wave rule

$$\mathrm{p}(\mathrm{s}(\mathrm{U})) :\Rightarrow \mathrm{s}(\mathrm{p}(\mathrm{U}))\ .$$

on the other hand is represented as

$$\lambda \mathrm{x}.\ \mathrm{p}(\mathrm{s}(\mathrm{U}'(\mathrm{x}))) :\Rightarrow \lambda \mathrm{x}.\ \mathrm{s}(\mathrm{p}(\mathrm{U}'(\mathrm{x})))\ , \tag{5.6}$$

i.e., it is lifted by replacing the occurrences of free variables with applications of free variables to the bound variables of the conclusion and abstracting both sides. This ensures that the conclusion and the left-hand side of the wave rule are of the same type. This lifting is very similar to the lifting over universal quantifiers in [Paulson 89]. The remaining question is how to represent annotations. This is discussed in the following section.

Annotations in Higher-Order Pattern Unification

A major problem in representation in CLAM is how to deal with annotations. CLAM uses a first-order representation, with functors to represent wave fronts, wave holes and sinks. Thus, for instance, a wave term

$$\boxed{f(\underline{x})}$$

is represented as

$$wf(f(wh(x))) \ ,$$

and first-order unification is used to unify wave terms.

This representation is inadequate in the context of higher-order terms, since the unification of higher-order wave terms using this representation can lead to ill-annotated terms. For instance, the unification

$$\lambda x. \boxed{s(\underline{U(x)})} \stackrel{?}{=} \lambda x. \boxed{C(\underline{x})} \qquad (5.7)$$

represented as[1]

$$\lambda x.\, wf(s(wh(U(x)))) \stackrel{?}{=} \lambda x.\, wf(C(wh(x)))$$

yields the unification

$$\lambda x.\, wf(s(wh(C'(wh(x))))) \ ,$$

which corresponds to the term

$$\lambda x. \boxed{s(\underline{C'(\underline{x})})} \ ,$$

which is ill-annotated, having two nested wave holes in a wave front.

Finding a representation of annotations that behaves as desired under unification is a topic of ongoing research. An approach that works well for higher-order terms which contain at most one wave hole was first suggested in [Liang 92] and

[1]Note that the right-hand side expression, $\lambda x.\, wf(C(wh(x)))$, is not a higher order pattern.

elaborated in [Gallagher 93]. In this approach, wave terms are represented as a binding construct $\mathit{wave}/2$. A wave term

$$\boxed{\mathsf{f}(\underline{\mathsf{x}})}$$

is represented as

$$\mathit{wave}(\lambda u.\ f(u), x)\ .$$

As shown in [Gallagher 93], for a wave term $\mathit{wave}(F, H)$, the unannotated term is obtained simply via $F(H)$.

In this notation, the unification problem (5.7) above would be represented as[2]

$$\lambda x.\ \mathit{wave}(\lambda u.\ s(u), U(x)) \stackrel{?}{=} \lambda x.\ \mathit{wave}(\lambda u.\ C(u), x)\ ,$$

which yields the most general unification

$$\lambda x.\ \mathit{wave}(\lambda u.\ s(u), x)\ ,$$

corresponding to the well-annotated

$$\lambda x.\ \boxed{s(\underline{x})}\ .$$

Using this representation, we need to modify the representation of wave rules slightly. Suppose we have a term of the form

$$\mathrm{plus}(\boxed{s(s(\underline{x}))}, y)\ ,$$

which is represented as

$$\lambda x, y.\ \mathrm{plus}(\mathit{wave}(\lambda u.\ s(s(u)), x), y)\ ,$$

and the wave rule for plus,

$$\mathrm{plus}(\boxed{s(\underline{U})}, V) :\Rightarrow \boxed{s(\underline{\mathrm{plus}(U, V)})}\ ,$$

[2]This time, the right-hand side expression, $\lambda x.\ \mathit{wave}(\lambda u.\ C(u), x)$, is a higher-order pattern.

which in the higher-order notation so far would be represented as

$$\lambda x, y.\ \mathrm{plus}(wave(\lambda u.\ s(u), U(x,y))), V(x,y)) :\Rightarrow$$
$$\lambda x, y.\ wave(\lambda u.\ s(u), \mathrm{plus}(U(x,y), V(x,y)))\ .$$

Clearly, one would like the wave rule to apply to the term in such a situation. However, the unification of the term and the left-hand side of the wave rule fails, because the wave fronts $\lambda u.\ s(s(u))$ and $\lambda u.\ s(u)$ do not unify. This problem also occurs in the first-order representation of CLAM Version 2.0. There, the problem is solved by splitting the wave front. Thus, the wave front of

$$\mathrm{plus}(\boxed{s(s(\underline{x}))}, y)$$

would be split into two

$$\mathrm{plus}(\boxed{s(\underline{\boxed{s(\underline{x})}})}, y)\ ,$$

to which the plus wave rule would apply.

[Gallagher 93] shows that this effect can be achieved directly in the higher-order notation, by introducing in the wave front of the left-hand side of the wave rule a new variable, which then "absorbs" any "extra" wave front of the conclusion into the wave hole of the wave rule. Thus, the left-hand side of the plus wave rule becomes

$$\lambda x, y.\ \mathrm{plus}(wave(\lambda u.\ s(F(u)), U(x,y)), V(x,y))\ .$$

where F is the new variable. Since F may become instantiated with a wave front, all occurrences of the wave hole on the right-hand side must be annotated appropriately, i.e., replaced with $wave(F, U(x,y))$. The entire wave rule then becomes

$$\lambda x, y.\ \mathrm{plus}(wave(\lambda u.\ s(F(u)), U(x,y))), V(x,y)) :\Rightarrow$$
$$\lambda x, y.\ wave(\lambda u.\ s(u), \mathrm{plus}(wave(F, U(x,y)), V(x,y)))\ .$$

In the more intelligible box-and-underlining notation, this corresponds to

$$\lambda x, y.\ \mathrm{plus}(\boxed{s(F(\underline{U(x,y)}))}, V(x,y)) :\Rightarrow \lambda x, y.\ \boxed{s(\underline{\mathrm{plus}(\boxed{F(\underline{U(x,y)})}, V(x,y))})}\ .$$

This representation has a further advantage in the context of quantifiers. Using the first-order notation, a wave rule such as

$$\forall x. \boxed{P \wedge \underline{Q(x)}} :\Rightarrow \boxed{P \wedge \underline{\forall x.\ Q(x)}}$$

requires checking the side condition that x does not occur free in P. In the higher-order notation, however, this condition can be encoded directly in the wave rule

$$\forall(\lambda x.\ wave(\lambda u.\ P \wedge F(u), Q(x))) :\Rightarrow wave(\lambda u.\ P \wedge u, \forall(\lambda x.\ wave(F, Q(x))))$$

In conclusion, Gallagher's higher-order representation allows us to ripple correctly with annotated higher-order terms in an elegant way, without leaving the realm of higher-order patterns. The representation is sufficient as long as we have no wave fronts with multiple holes. These can occur when rippling towards more than one induction hypothesis, as would be the case, for instance, when inducing over trees. [Gallagher 93] contains some suggestions on how to extend the representation to multiple holes.

5.3.2 A More General Representation of the Step Case

The representation of the schematic step case presented in Section 5.1 does not cover more sophisticated induction schemes where the successor of an induction variable is a combination of itself and another variable. This is the case, for instance, in the quotient remainder example

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

where the successor of y is $y + x$. In the schematic step case in the previous section, we represented the successor of a variable with a meta-variable applied to the variable, e.g., the successor of y was represented as $\mathcal{D}(y)$. In Section 4.5.1, we explained the rationale of using such applications. One reason was to ensure that the instantiation of the meta-variable was a closed term, i.e., to avoid spurious instantiations. This means, however, that the instantiation of the meta-variable

cannot refer to any of the other potential induction variables. In terms of the quotient remainder example, it means that the instantiation of $\mathcal{D}$ cannot refer to x, q or r. To allow this, we must generalize the representation of the schematic step case by representing the successor of a potential induction variable as an application of the meta-variable to *all* potential induction variables. Thus, in the example, the successor of y is represented as $\mathcal{D}(x, y, q, r)$, which, properly annotated, becomes $\boxed{\mathcal{D}(x, \underline{y}, q, r)}$. The entire schematic step case is then

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

While dealing with this representation is not a problem for *Periwinkle*, it is not particularly fit for human consumption. In the current implementation, therefore, both representations are available as separate methods (mor_induction and fancy_mor_induction, resp.), and users can select the one they want. A future version should use the more general representation, but suppress unwanted detail in the user interface.

5.3.3 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 in [Bundy *et al* 93] makes some restrictions, i.e., existential rippling and meta-variables are excluded, precisely because they can lead to nontermination. Since middle-out induction inherently depends on meta-variables, and we do not want to rule out existential quantification in our specifications, 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 the 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:

1. Non-termination in branches that contain no solutions (failure branches)

2. Non-termination in branches that contain at least one solution (success branches)

The subset example illustrates both sources of non-termination. The schematic induction conclusion is

$$\mathsf{subset}(\boxed{\mathcal{C}(\underline{x})}, \boxed{\mathcal{D}(\underline{y})}) \leftrightarrow \forall z.\ \mathsf{member}(z, \boxed{\mathcal{C}(\underline{x})}) \rightarrow \mathsf{member}(z, \boxed{\mathcal{D}(\underline{y})})\ .$$

Initially, the conclusion contains only potential wave fronts. One wave rule, that for member, is applicable

$$\mathsf{member}(X, \boxed{H :: \underline{T}}) :\Rightarrow \boxed{X = H \vee \underline{\mathsf{member}(X, T)}}\ . \tag{5.8}$$

However, it can be applied in two ways—to either occurrence of member. Suppose we apply it to the second occurrence, i.e., to $\mathsf{member}(z, \boxed{\mathcal{D}(\underline{y})})$. We get

$$\mathsf{subset}(\boxed{\mathcal{C}(\underline{x})}, \boxed{\mathcal{D}' :: \mathcal{D}''(\underline{y})}) \leftrightarrow$$
$$\forall z.\ \mathsf{member}(z, \boxed{\mathcal{C}(\underline{x})}) \rightarrow \boxed{z = \mathcal{D}' \vee \underline{\mathsf{member}(z, \boxed{\mathcal{D}''(\underline{y})})}}\ .$$

We could now easily apply wave rule (5.8) again to either occurrence of member, and again ad infinitum. The reason for such infinite sequences is that every application of a wave rule to a potential wave front generates a new copy of the potential wave front nested in the new wave hole. Thus, if we have a potential wave front

$$f(\boxed{\mathcal{C}(\underline{x})})\ ,$$

a wave rule application will lead to an instantiation

$$\boxed{F[\underline{f(\boxed{\mathcal{C}'(\underline{x})})}]}\ .$$

Thus, if we can apply a wave rule to a potential wave front at all, we can apply it infinitely many times. This can occur both in success and failure branches. A

simple strategy that helps avoid going down these infinite branches is to prefer wave rules that ripple at least one definite wave front. In the following, we call the application of a wave rule to potential wave fronts only a *speculative* ripple, and an application of a wave rule to at least one definite wave front a *definite* ripple. Using this strategy in the subset example means that preference is given to a wave rule that ripples either

$$\mathtt{subset}(\boxed{\mathcal{C}(\underline{x})}, \boxed{\mathcal{D}' :: \mathcal{D}''(\underline{y})})$$

or

$$\mathtt{member}(z, \boxed{\mathcal{C}(\underline{x})}) \to \boxed{z = \mathcal{D}' \vee \underline{\mathtt{member}(z, \boxed{\mathcal{D}''(\underline{y})})}}\ .$$

In synthesis, we have no wave rule for subset and therefore cannot ripple the former. There is, however, a wave rule that ripples the latter, namely the propositional one (see Section 6.1)

$$P \to \boxed{Q \vee \underline{R}} :\Rightarrow \boxed{Q \vee \underline{P \to R}}\ .$$

Applying it yields

$$\begin{aligned}&\mathtt{subset}(\boxed{\mathcal{C}(\underline{x})}, \boxed{\mathcal{D}' :: \mathcal{D}''(\underline{y})}) \leftrightarrow \\ &\quad \forall z. \boxed{z = \mathcal{D}' \vee \underline{\mathtt{member}(z, \boxed{\mathcal{C}(\underline{x})}) \to \mathtt{member}(z, \boxed{\mathcal{D}''(\underline{y})})}}\ .\end{aligned}$$

Now, however, there is no wave rule that applies to a definite wave front. In fact, although there is a way for the proof to continue, it involves a case split on whether $\mathcal{D}'$ is a member of $\mathcal{C}(x)$ or not. Finding this case split is currently beyond the scope of the system. Thus, since we are on a failure branch, we should not apply another speculative ripple. What we need to do is to go back to the first application of a wave rule and try the alternative, namely applying wave rule (5.8) to the other occurrence of member, i.e., $\mathtt{member}(z, \boxed{\mathcal{C}(\underline{x})})$. Ignoring for the moment how to detect this and achieve the backtracking, we go back to the initial schematic conclusion and apply the member wave rule to the first occurrence of member

$$\begin{aligned}&\mathtt{subset}(\boxed{\mathcal{C}' :: \mathcal{C}''(\underline{y})}, \boxed{\mathcal{D}(\underline{y})}) \leftrightarrow \\ &\quad \forall z. \boxed{z = \mathcal{C}' \vee \underline{\mathtt{member}(z, \boxed{\mathcal{C}''(\underline{y})})}} \to \mathtt{member}(z, \boxed{\mathcal{D}(\underline{y})})\ .\end{aligned}$$

The propositional wave rule

$$\boxed{P \vee \underline{Q}} \rightarrow R :\Rightarrow \boxed{(P \rightarrow R) \wedge \underline{Q \rightarrow R}}$$

applies to the definite wave front on the right-hand side, and we get

$$\begin{aligned} &\text{subset}(\boxed{\mathcal{C}' :: \mathcal{C}''(\underline{y})}, \mathcal{D}(\underline{y})) \leftrightarrow \\ &\quad \forall z. \boxed{z = \mathcal{C}' \rightarrow \text{member}(z, \mathcal{D}(\underline{y})) \wedge \\ &\quad\quad \underline{\text{member}(z, \mathcal{C}''(\underline{y})) \rightarrow \text{member}(z, \mathcal{D}(\underline{y}))}} . \end{aligned}$$

Now the logical wave rule

$$\forall x. \boxed{P \wedge \underline{Q}} :\Rightarrow \boxed{\forall x. P \wedge \underline{\forall x. Q}}$$

applies to the new definite wave front on the right-hand side, and we obtain

$$\begin{aligned} &\text{subset}(\boxed{\mathcal{C}' :: \mathcal{C}''(\underline{y})}, \mathcal{D}(\underline{y})) \leftrightarrow \\ &\quad \boxed{\forall z. z = \mathcal{C}' \rightarrow \text{member}(z, \mathcal{D}(\underline{y})) \wedge \\ &\quad\quad \underline{\forall z. \text{member}(z, \mathcal{C}''(\underline{y})) \rightarrow \text{member}(z, \mathcal{D}(\underline{y}))}} . \end{aligned}$$

Again, there is no wave rule that applies to a definite wave front. Now, however, we have a choice between a speculative ripple and weak fertilization. Similarly to the situation in the failure branch, another speculative ripple would trigger another cycle of wave rule applications. Weak fertilization, however, completes the rippling, which constitutes success. It should therefore be preferred.

To summarize, some infinite branches of the rippling search tree can be avoided by the simple heuristic of preferring definite rippling and fertilization to speculative rippling. However, this heuristic does not avoid non-termination in failure branches. If we knew that there is always at least one success branch in the rippling search tree, breadth-first search would solve the termination problem. Unfortunately, however, this is not the case. A simple example of a rippling search tree with nothing but failure branches is a variant of the associativity of

plus[3]

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

What is needed to deal with this problem is a global control over speculative steps such as speculative rippling, generalization and lemma conjecturing. Such global control can be provided by the planning critics proposed in [Ireland 92]. A planning critic is a program that, given a failed proof planning attempt, analyzes it and suggests ways of correcting it. In the case of middle-out induction, the method and its critic would work together in the following way: The method allows an initial speculative ripple to trigger the 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 which would give rise to a wave rule, or a generalization. Implementing such a critic is left as future work (see Section 8).

Rippling for middle-out induction is thus controlled as follows: It allows only one speculative step (see Appendix C.2.1 for the code), which can be a speculative ripple, an unrolling step (see Section 6.3) or a case split. It then ripples (see Appendix C.1.2 for the code) while trying to fertilize as soon as possible. Until a middle-out induction critic is developed, however, this means that *Periwinkle* cannot find a proof for a theorem which depends on more than one speculative ripple. This is in fact rare, so that it is not a severe limitation.

---

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

5.4 Comparison of Middle-Out Induction with Other Approaches

The most widely used technique to select induction schemes is recursion analysis, which works well for formulae containing only universal quantifiers, but not necessarily for formulae containing existential quantifiers. There has not been much work on selecting induction schemes for formulae containing existential quantifiers, except within the framework of the Inka theorem-prover [Biundo *et al* 86, Biundo 89, Hutter 94], which is a resolution-based theorem prover with destructor-style induction.

The program synthesis system of [Biundo 89] (see Section 2.2.1), which is part of the Inka system, uses what is called the *most nested function* heuristic. A most nested function is one that occurs at an innermost position in the specification, i.e., has arguments that are variables, constants, or the skolem term only. 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 as well. If there are several such most-nested functions, the function which has the largest number of recursive arguments is chosen. The constraint that the recursive arguments of the most nested function must correspond to arguments of the skolem function serves to determine the recursion of the program. Thus, the type of recursion of the program will always corresponds to the type of recursion of the selected most nested function. While this heuristic is sufficient in simple examples, it performs as poorly as recursion analysis in more complex examples. For example, for the (skolemized) quotient remainder specification

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

the most nested function heuristic selects structural induction on y. While the system does find a proof using this induction, the resulting program is unintuitive and highly inefficient.

[Hutter 94], on the other hand, suggests a more sophisticated technique to select induction schemes for $\forall\exists$ formulae. Hutter recognizes the close relationship between the induction variables, the instantiation of the existential variables and the type of induction. Instead of selecting the induction variable and type of induction and then trying to find the instantiation of the existential variable, Hutter picks an induction variable and an instantiation of an existential variable, leaving the type of induction to be determined in the course of the proof. In doing so, Hutter is no longer limited only to recursions appearing in the specification.

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 context-moving rules (wave rules), are abstracted in that the only information retained is the dominating functor 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-ordering of the induction order remains to be established.

This approach and our middle-out approach are, in fact, closely related. Both rely in a similar fashion 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 in fact 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 approach 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.

5.5 Summary

In this section, we have shown how middle-out reasoning can be applied to solve another problem in program synthesis—the selection of induction schemes. Middle-out induction is a good alternative to recursion analysis and similar techniques, which are inadequate in synthesis proofs. While the idea of using middle-out reasoning to select induction schemes is not new [Bundy *et al* 90a], the issue of search control had not been addressed before. We have presented some simple heuristics which reduce the search space and avoid non-termination of rippling.

Chapter 6

Extensions to Rippling

The synthesis of logic programs is a new application of proof planning. As such, it poses new problems to the proof planner. While many of the problems are specific to program synthesis, a number of them can occur in proof planning in general. The methods developed to solve these more general problems are presented together in this section. They are applicable in the step cases of inductive proofs, when the rippling has become *blocked*, i.e., when no wave rule applies, but fertilization is not yet possible. In such situations, methods are needed which make further rippling or fertilization possible. These methods are generally called *unblocking* methods. Rippling can become blocked for several reasons. Often, however, the reasons are missing wave rules or nested quantifiers. Sections 6.1 and 6.2 address the problem of generating missing wave rules, while Section 6.3 addresses the issue of nested quantifiers. Section 6.4 presents a method which, in certain cases, allows fertilization even though the conclusion is not fully rippled.

6.1 Generating Logical Wave Rules

Initially, *Periwinkle* used a library of around sixty wave rules based on schematic lemmas about logical connectives, including associative, commutative and distributive laws for connectives or combinations thereof. The wave rules (4.4) and (4.5) used in Section 4.4, are two examples of such wave 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., ***propositional*** wave rules, which are wave rules expressed in terms of propositional connectives only. Wave rules (6.1)–(6.4) are examples of propositional wave rules. Wave rule (6.5) is an example of a logical wave rule that is not propositional, since it involves quantifiers.

$$\boxed{\mathsf{P} \wedge \underline{\mathsf{Q}}} \wedge \mathsf{R} \;:\Rightarrow\; \boxed{\mathsf{P} \wedge \underline{\mathsf{Q} \wedge \mathsf{R}}} \tag{6.1}$$

$$\boxed{\underline{\mathsf{P}} \wedge \mathsf{Q}} \wedge \mathsf{R} \;:\Rightarrow\; \boxed{\underline{\mathsf{P} \wedge \mathsf{R}} \wedge \mathsf{Q}} \tag{6.2}$$

$$\boxed{\underline{\mathsf{Q}} \wedge \mathsf{P}} \leftrightarrow \boxed{\underline{\mathsf{R}} \wedge \mathsf{P}} \;:\Rightarrow\; \mathsf{Q} \leftrightarrow \mathsf{R} \tag{6.3}$$

$$\neg\boxed{\underline{\mathsf{P}} \wedge \mathsf{Q}} \;:\Rightarrow\; \boxed{\underline{\neg \mathsf{P}} \vee \neg \mathsf{Q}} \tag{6.4}$$

$$\neg\boxed{\forall \mathsf{X}.\ \underline{\mathsf{P}}} \;:\Rightarrow\; \boxed{\exists \mathsf{X}.\ \underline{\neg \mathsf{P}}} \tag{6.5}$$

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 synthesis of the not_member program (see Appendix A), for instance, the rippling is blocked in

$$\mathrm{not_member}(\mathsf{x}, \boxed{\mathsf{h} :: \underline{\mathsf{t}}}) \leftrightarrow \neg\boxed{\mathsf{x} = \mathsf{h} \vee \underline{\mathsf{member}(\mathsf{x}, \mathsf{t})}}\,.$$

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 missing part of the lemma. The truth tables for the known expressions are

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

We now need to find an expression $\mathcal{F}$ that contains $\neg Q$ as a subterm and has the same truth values as $\neg(P \vee Q)$.

First, we try the simple cases: $\neg Q$ itself and its negation $\neg\neg Q$. This fails, and we therefore 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. In the column for E, T indicates that E must be true for the given values of the variables P and Q, F indicates that E must be false, and T/F indicates that E may be either true or false. A notation of X indicates a conflict; no possible value is logically consistent. 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

Any formula with a conflict, here $\neg Q \vee E$, is discarded. We then try to complete the surviving candidates, here $\neg Q \wedge E$, by finding a variable or a negation of a variable that satisfies the derived truth table. In this example, the truth table is satisfied by $\neg P$.

P	Q	$\neg P$	$\neg Q$	E
T	T	F	F	T/F
T	F	F	T	F
F	T	T	F	T/F
F	F	T	T	T

If no variable or 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 the original database of propositional wave rules. Although the approach uses exhaustive search, the search rarely goes beyond a depth of three, and is sufficiently efficient at that level.

Although this algorithm was able to generate all propositional wave rules in our original database, it is not clear whether it is actually complete, i.e. whether it will always find a wave rule if a wave rule exists. The problem lies in the syntactic restrictions that were made to cut down the search space, i.e., the requirement that the wave hole is at depth one and that the first argument of a binary connective is a variable or the negation of a variable. The question

is thus whether there are wave rules that cannot be expressed in this syntactic form.

The algorithm, while being entirely sufficient for the work presented here, is not as general as we would like. Because it is based on truth tables, it cannot generate logical wave rules involving quantifiers, and it is applicable only in classical logic, not in constructive logics. We are therefore currently investigating an alternative approach that can deal with quantifiers and is applicable to constructive logics (see Section 8.4.3).

The method that generates and applies propositional wave rules can be summarized as follows (see Appendix C.2.2 for the code):

Method: Prop_wave

1. Find a subexpression in the conclusion such that it contains at least one wave front, the connective dominating the wave fronts is a propositional connective, and the connectives in the wave fronts are propositional.
2. Generate a lemma that gives rise to an appropriate wave rule.
3. Apply the wave rule.

The output of the prop_wave method is the rewritten sequent. The corresponding tactic proves the lemma and applies the rewrite.

6.2 Equivalence-Preserving Existential Rippling

The variation of rippling known as existential rippling was presented in Section 3.2.2. This section discusses some difficulties when applying existential rippling in logic program synthesis. As pointed out in Section 4.2.1, wave rules

used to rewrite a subexpression of an equivalence must be based on equivalence or equality. Thus, for instance, to apply the wave rule

$$even(\boxed{s(s(\underline{U}))}) :\Rightarrow even(U)$$

under an equivalence, the lemma underlying the wave rule must be

$$\forall u.\ even(s(s(u))) \leftrightarrow even(u)\ .$$

Existential rippling allows one or more of the wave terms on the left-hand side of the wave rule to match with existential variables annotated as potential wave fronts. Given, for instance, the conclusion

$$\forall y.\ \exists x.\ even(x) \wedge double(x, y)\ ,$$

existential rippling allows x and $\boxed{s(s(\underline{U}))}$ to match, and the term is rewritten to

$$\forall y.\ \exists x.\ even(x) \wedge double(\boxed{s(s(\underline{x}))}, y)\ .$$

The main problem with existential rippling for logic program synthesis proofs is that the existential version of a wave rule is not necessarily equivalence-preserving, even if the original version is. An example where the existential wave rule is not equivalence-preserving is the synthesis conjecture specifying that a list k occurs at the back of a list l

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

Applying structural induction on l yields the step case

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

Now, we would like to apply the wave rule

$$app(\boxed{H_1 :: \underline{T_1}}, L, \boxed{H_2 :: \underline{T_2}}) :\Rightarrow \boxed{H_1 = H_2 \wedge \underline{app(T_1, L, T_2)}} \tag{6.6}$$

in its existential version

$$\exists \boxed{X}.\ app(\boxed{X}, K, \boxed{H :: \underline{T}}) :\Rightarrow \boxed{\exists x_1.\ \exists x_2.\ x_1 = H \wedge \underline{app(x_2, K, T)}}\ . \tag{6.7}$$

While the lemma underlying (6.6)

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

holds, the equivalence that would justify (6.7)

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

is a non-theorem (The left-hand side is true, but the right-hand 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, therefore, we have developed a less expensive and more generally applicable alternative, which is the unrolling method discussed in the following section.

6.3 Unrolling for Unblocking

The specification language we use is full first-order predicate logic. Therefore, specifications may well contain quantifiers. This means that the synthesis conjectures can contain quantifiers that are nested in the right-hand side of the equivalence. This is the case, for instance, in the *back* specification of the previous section. The theorems that have been proved by CLAM are normally quantified at the front, i.e., are in prenex normal form[1]. Moving away from prenex normal form creates some problems for proof planning in that the quantifiers may block the rippling. The method presented in this section, *unrolling*, can, in many

---

[1]The reason for this is probably that many of the theorems were taken from the Boyer-Moore corpus [Boyer & Moore 79], and the prover NQTHM uses a quantifier-free logic where all variables are implicitly universally quantified.

cases, unblock the rippling by introducing a case split on the quantified variable causing the blockage.

We illustrate this using the back example of the previous section. The induction conclusion is

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

The rippling is blocked from the outset by the existentially quantified variable x. As we showed in the previous section, the existential version (6.7) of the app wave rule (6.6) cannot be applied because it is not equivalence-preserving. However, applying wave rule (6.6) is clearly what is called for. Thus, we need to find some other way to do so. For wave rule (6.6) to be applicable, x needs to be brought into the form $\boxed{x_1 :: \underline{x_2}}$. One way to do this is to introduce 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 make sure that the skeleton of the conclusion is preserved and that any additional structure introduced by the case split is put into a wave front. This is achieved by annotating the case split accordingly. Schematically, a case split on x in $\forall x{:}nat.\ P(x)$ is annotated as

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

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

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

to which wave rule (6.6) applies, yielding

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

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

In the current implementation, only case splits on the structure of data types are considered. Thus, for numbers, unrolling for existential quantifiers substitutes

$$\exists x{:}\mathit{nat}.\ P[x]$$

with

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

and unrolling for universal quantifiers substitutes

$$\forall x{:}\mathit{nat}.\ P[x]$$

with

$$\boxed{P[0] \wedge \underline{\forall x{:}\mathit{nat}.\ P[\boxed{s(\underline{x})}]}}\ .$$

while for lists, unrolling for existential quantifiers substitutes

$$\exists x{:}\mathit{nat\ list}.\ P[x]$$

with

$$\boxed{P[\mathit{nil}] \vee \exists h{:}\mathit{nat}.\ \underline{\exists t{:}\mathit{nat\ list}.\ P[\boxed{h::\underline{t}}]}}\ ,$$

and unrolling for universal quantifiers substitutes

$$\forall x{:}\mathit{nat\ list}.\ P[x]$$

with

$$\boxed{P[\mathit{nil}] \wedge \forall h{:}\mathit{nat}.\ \underline{\forall t{:}\mathit{nat\ list}.\ P[\boxed{h::\underline{t}}]}}\ .$$

The method could be extended to cover more sophisticated case splits as well.

It is worth noting the relationship between equivalence-preserving existential rippling and unrolling. Applying an equivalence-preserving existential wave rule to a conclusion

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

yields a conclusion

$$\exists y'{:}\mathit{nat}.\ \boxed{F(\underline{P(x, y')})}\ . \tag{6.8}$$

Unrolling initially yields

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

Now, however, because the existential rewrite was equivalence-preserving, the case $\mathsf{P}(x,0)$ must be false. Thus, after we simplify the wave front and apply the original version of the wave rule, we obtain the same conclusion as with the existential rippling, i.e., (6.8).

Unrolling can cause looping. In fact, the looping problems are identical to the ones that occur when rippling in middle-out induction. Therefore, unrolling is controlled in the same way as speculative waves (see Section 5.3.3): Rippling allows only one speculative step. This prevents looping, but also means that a proof that depends on two or even more unrollings cannot be found.

The unrolling method can be summarized as follows (see Appendix C.2.3 for the code):

Method: Unroll

1. Find a quantified subexpression in the conclusion that contains a subexpression with the quantified variable as an argument, to which a wave rule applies if a case split is introduced on that variable.
2. Apply the appropriate case split.

The output of the unroll method is the sequent with the case split.

6.4 Very Weak Fertilization

The unblocking 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, and 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, i.e., where one side of the equality is surrounded by a wave front.

A simple example where this occurs is the synthesis of a reverse relation from a reverse function,

$$\forall k, l.\ \mathsf{rrev}(k, l) \leftrightarrow \mathsf{frev}(k) = l\ ,$$

where frev is defined as

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

From the frev definition, we get the wave rule

$$\mathsf{frev}(\boxed{H :: \underline{T}}) :\Rightarrow \boxed{\mathsf{app}(\underline{\mathsf{frev}(T)}, H :: \mathsf{nil})}\ . \tag{6.9}$$

By structural induction on k, we obtain the step case

$$\begin{aligned} &\forall l.\ \mathsf{rrev}(t, l) \leftrightarrow \mathsf{frev}(t) = l \\ &\vdash \\ &\forall l.\ \mathsf{rrev}(\boxed{h :: \underline{t}}, l) \leftrightarrow \mathsf{frev}(\boxed{h :: \underline{t}}) = l\ . \end{aligned}$$

Applying wave rule (6.9) yields

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

The rippling is now blocked. However, the wave hole on the right-hand side of the equivalence, i.e., $\mathsf{frev}(t)$, 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

$$\boxed{\phi(\underline{\psi(x)})} = y \Rightarrow \boxed{\exists y'.\, \phi(y') = y \land \underline{\psi(x) = y'}} \tag{6.10}$$

which is almost, but not quite a wave rule. It does not quite preserve the skeleton. The skeleton of the left-hand side is $\psi(x) = y$, that of the right-hand side is $\psi(x) = y'$. The notion of rippling is currently being extended to allow such rules as wave rules as well, provided the induction hypothesis counterpart of y and y' is a sink, i.e., a universally quantified variable.

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

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

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

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

Very weak fertilization deals with an instance of the problems that arise from dealing with 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 letting the corresponding relations share existential variables. Thus, in order to make progress in the synthesis of a relation, it is necessary to

unpack the nesting of functions by introducing existential variables. For a further discussion of the problems arising from the use of relations, see Section 8.3.

The very weak fertilization method can be summarized as follows (see Appendix C.2.4 for the code):

Method: Very_weak_fertilize

1. There is a subexpression in the conclusion such that it is an equality, one side of the equality is a sink and the other side of the equality is fully rippled.
2. Apply the schematic rewrite (6.10) and weak fertilize.

The output of the very_weak_fertilize method is the post-fertilization sequent.

6.5 Summary

In this section, we have presented several methods which proved necessary to synthesize all but the most trivial programs. They are generally applicable when rippling becomes blocked before fertilization is possible. The methods that generate missing logical lemmas and that introduce case splits on quantified variables are particularly promising in that they are useful in proof planning in general. It would be useful to extend both methods, the propositional wave rule method to quantifiers and constructive logics, and the unrolling method to cover more sophisticated case splits.

Chapter 7

Implementation and Results

This chapter gives an overview of the implementation of our system, *Periwinkle*, as an extension to CLAM, and reports on some of the practical results achieved. In particular, we discuss the respects in which *Periwinkle* outperforms other systems, and we classify the programs it can synthesize in terms of the problems their syntheses presented.

7.1 Implementation

Our system, *Periwinkle*, is an adaptation and extension of the CLAM-Oyster system [Bundy *et al* 90c]. Oyster is an interactive proof checker for a variant of Martin-Löf type theory, based on NUPRL [Constable *et al* 86]. CLAM is a proof planner that generates proof plans which can be executed in Oyster. Initially, CLAM was meant to be a proof planner to plan inductive proofs in type theory. Recently, however, there has been an increased interest in using CLAM for other types of proofs, logics or proof-checkers. Therefore, there is an ongoing effort into making CLAM less logic-dependent.

At the object level, Oyster is being replaced by an interactive proof checking shell called Mollusc [Richards 93], which, given a specification of a logic, becomes a proof checker for that logic. Mollusc provides an interface to CLAM.

Using Mollusc, we implemented a proof checker for many-sorted first-order predicate logic with equality, following Gallier's presentation of the Gentzen System $G_=$ [Gallier 86] (see Appendix D).

At the planning level, CL^AM needed to be adapted to first-order predicate logic with equality. CL^AM was designed as a proof planner for Martin-Löf type theory, and some of its code contained concrete syntax. This code was identified and reimplemented. Concrete syntax also appeared in some of the methods, which were therefore also rewritten. Code that was rewritten generally uses constructor/destructor predicates rather than concrete syntax. This was particularly emphasized in the new methods developed in this thesis (see Section 6). Thus, for instance, a predicate `strip/3` that strips leading universal quantifiers is written as

```
strip( [], T1, T1 ) :-
    not( universal_quantifier( T1, _ ) ).

strip( [ V | Vs ], T1, T3 ) :-
    universal_quantifier( T3, _ ),
    bound_variable( T3, V ),
    bound_formula( T3, T2 ),
    strip( Vs, T1, T2).
```

rather than

```
strip( [], T1, T1 ) :-
    not( T1 = forall( _,_ ) ).

strip( [ X:Type | Vs ], T1, forall( X:Type, T2 ) ) :-
    strip( Vs, T1, T2).
```

The predicates `universal_quantifier`, `bound_variable` and `bound_formula` are the constructor/destructor predicates. The parser generator of Mollusc can

be used to generate such predicates automatically from the syntax description of the logic. The use of constructors/destructors rather than concrete syntax has already proved useful, as a number of the methods presented here are being used directly in other logics.

Beyond these syntax-related adaptations, CIAM 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. Higher-order pattern unification was implemented following the algorithm of [Nipkow 93] (see Appendix E). Second, we implemented code to automatically detect the need for and run auxiliary syntheses. Finally, we implemented the new methods (see Appendix C) and auxiliary code related to these.

7.2 Synthesized Programs

One of the aims of this thesis 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 & Prestwich 88a], delete from [Bundy *et al* 90b] and qr from [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. We discuss these examples in more detail below.

The complete set of examples synthesized by *Periwinkle* is listed in Appendix A, which shows the specification, the program synthesized, the definitions and lemmas used, and the planning and tactic execution times for each example. Additionally, Appendix B contains traces of the planning for the examples subset, even and rrev.

Our methodology used three classes of examples. We used a number of simple examples as the basis for developing our methods. We then refined these methods using the examples taken from the literature. Finally, we tested the

system's robustness using a number of additional examples. These three classes of examples are discussed in the following subsections.

7.2.1 Examples for Method Development

We used several simple examples as the basis for developing our synthesis approach and the methods presented in the previous sections. These examples were generally also the ones used as running examples. In particular, simple program synthesis (i.e., requiring no auxiliary syntheses and none of the methods of Section 6) was developed using $\mathrm{between}$; auxiliary syntheses were developed using max; middle-out induction using even; propositional rippling using $\mathrm{not_member}$; unrolling using back; and very weak fertilization using rrev.

7.2.2 Examples from the Literature

The examples from the literature were chosen to present *Periwinkle* with a number of challenge problems. Using these examples, the methods were refined to address problems which arose out of these examples. The final version of *Periwinkle* is able to automatically synthesize good programs for all of the specifications and can thus be considered to have achieved its aims.

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

$$\forall x, y.\ \mathrm{subset}(x, y) \leftrightarrow (\forall z. \mathrm{member}(z, x) \rightarrow \mathrm{member}(z, y)) \ ,$$

posed the problems of rippling using logical wave rules (see Section 6.1) and controlling the rippling to avoid non-termination (see Section 5.3.3). While the synthesis requires three instances of user interaction in [Lau & Prestwich 88a], it requires none in *Periwinkle*.

The delete example from [Bundy *et al* 90b],

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

contains nested quantifiers which block the rippling from the outset and thus require unrolling (see Section 6.3). 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 $fapp$,

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

whose proof is beyond the current planning capabilities of *Periwinkle*. This is because the proof is non-inductive, and *Periwinkle* is geared towards inductive proofs.

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

$$\forall w, x, y, z.\ qr(w, x, y, z) \leftrightarrow plus(times(w, x), y) = z\ \wedge\ y < x$$

requires both middle-out induction (see Section 5) and auxiliary syntheses (see Section 4.5.2). In [Biundo 89], structural induction on z is chosen, which leads to an inefficient and unintuitive program. *Periwinkle* chooses a more appropriate induction, namely structural induction on w and $plus$ induction on z, which leads to a simpler, more efficient program.

The synthesis of qr requires three lemmas, whose proofs can be automatically planned by *Periwinkle*. The lemmas are the associativity of $plus$

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

a variant of the cancellation of $plus$

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

and a lemma on $<$ and $plus$

$$\forall x, y, z.\ (z < x \rightarrow plus(x, y) = z) \leftrightarrow false\ .$$

7.2.3 Test Examples

A number of additional examples were run to test the robustness of *Periwinkle*. These are also included in Appendix A. The following table lists the test examples, as well as the development and literature examples, categorized by the

problems they posed. The test examples are marked with an asterisk. The category "General synthesis" contains examples that do not fall into any other category, while the category "Middle-out induction" contains only the examples which would have failed or used an unsuitable induction with recursion analysis instead of middle-out induction. All syntheses were run using middle-out induction.

General synthesis	between, rapp*, rplus*
Middle-out induction	even, qr
Auxiliary syntheses	add3*, app_length*, max, qr, rrev
Logical wave rules	back, before*, insert*, max, not_member, subset
Unrolling	back, before*, delete, even, insert*
Very weak fertilization	rrev, rtimes*

Appendix A contains one example, rcount, that has not been mentioned so far. The synthesis of rcount requires a case split (see Section 3.2.2). While the case split method of CL^AM has been adapted and extended to suffice for this example, it needs to be refined to interact properly with middle-out induction.

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 8.4.2), the latter because *Periwinkle* is not able to do the required case split. The case split is similar to the one mentioned in the subset example in Section 5.3.3. Another example that fails is the even example in Section 5.1

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

This example is interesting in that *Periwinkle* finds the appropriate induction, i.e., two-step induction on x, but fails on the auxiliary synthesis in the second base case

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

since it is not able to determine that $\exists y.\ y \times s(s(0)) = s(0)$ is false.

7.3 Summary

This section discussed the implementation of the synthesis approach presented in the previous chapters as well as its application to a number of examples. We have tested the system *Periwinkle* on a representative selection of examples, including difficult problems taken from the literature. The test results show that *Periwinkle* has improved on the abilities of competitive systems for the examples taken from the literature, in particular on the ability to do automatic syntheses and the ability to select appropriate induction schemes.

Chapter 8

Future Research

We have investigated how proof planning and middle-out reasoning can be exploited for logic program synthesis. While the results we have presented are encouraging, there is always room for improvement. A number of improvements and extensions are discussed in the following sections. Immediate improvements to the implementation are discussed in Section 8.1, issues related to middle-out synthesis are discussed in Section 8.2, issues related to middle-out induction are discussed in Section 8.3 and issues related to rippling are discussed in Section 8.4.

8.1 Immediate Improvements

Our approach is implemented in a system called *Periwinkle*, which should be considered an experimental prototype. Many of its shortcomings are due to the fact that it was built as an extension to the proof planner CLAM, and a number of design decisions made in the implementation of CLAM are not suitable for the type of middle-out reasoning we used.

First, CLAM does not have an explicit representation of the plan. This poses a serious problem in middle-out reasoning, since the plan may well contain multiple occurences of a meta-variable. When a meta-variable becomes instantiated, this instantiation needs to be propagated throughout the entire plan. Since CLAM has no explicit representation of the plan, it is difficult to achieve

this propagation by accumulating and applying a list of substitutions. Therefore, we decided to use Prolog variables to represent meta-variables, despite the danger of spurious instantiations. The next release of CLAM will have an explicit representation of the plan, which will make a ground representation of meta-variables possible. Moving to a higher-order language such as λ-Prolog [Miller & Nadathur 88] at the same time would considerably ease the manipulation of the binding constructs inherent in middle-out reasoning.

Also, CLAM's first-order representation of annotations has proved flawed (see Section 5.3.1), and much time was spent searching for a suitable representation. It was not until the work in [Gallagher 93] became available that a satisfactory solution was found. While the new representation is used in the unification, there was not sufficient time to reimplement the existing annotation-dependent code. The implementation is thus awkward in that CLAM's first-order notation of annotations is used in all places except unification. Prior to unification, therefore, we translate to the higher-order notation, and after unification, we translate back to the first-order notation. This should be avoided by reimplementing the annotation-dependent code in CLAM.

8.2 Synthesis

The work in this thesis has concentrated on the fully automatic synthesis of recursive logic programs that are partially correct and complete. There is no user interaction with the system beyond the user's entering of the specification. 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 the desired type of the program, for instance by giving the complexity or type of recursion of the program, or by requiring that the program be tail-recursive. These requirements map either into a partial instantiation of the program body or into constraints on the proof [Hesketh 91]. 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 then determined from the outset. The induction, however, may well be incompatible with the wave rules available. If so, the rippling will become blocked. To find missing wave rules, one could ask the user for assistance or one could use critics and middle-out reasoning, as shown in [Ireland 92], to speculate lemmas giving rise to the missing wave rules.

8.3 Middle-Out Induction

8.3.1 Multiple Step Cases

In middle-out induction, we set up a schematic step case and then allow rippling to instantiate it. In Section 5, we implicitly assume that the proof has only one step case. Some types of induction, however, require more than one step case. If a conjecture requires such an induction, a choicepoint will occur at some stage in the rippling, i.e., more than one wave rule will apply and the subsequent rippling for each of the wave rules will lead to different instantiations of the induction meta-variables.

To deal with such inductions, the approach so far could be extended in the following way: Rippling would pick one of the wave rules and pursue that branch to fertilization. Then, this step case could be matched against the database of induction schemes. Thus, after completing one schematic step case, we would retrieve the induction scheme, and complete the proof by proving the remaining step and base cases.

8.3.2 Unknown Induction Orderings

Once the schematic step case has been completed, the instantiation of the induction meta-variables is used as an index into the database of induction schemes.

Currently, the middle-out induction method 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 the 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.

An example is the between proof

$$\forall x, y, z.\ between(x, y, z) \leftrightarrow x \leq y \wedge y \leq z\ .$$

Periwinkle proves this conjecture via a simultaneous induction on x, y and z. If this type of induction is not available, the induction could be patched by doing a simultaneous induction on x and y, and a case split on $z = 0 \vee \exists z'.\ z = s(z')$. In the second case, z would be used as a sink rather than an 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 try to prove (or disprove) the well foundedness of the ordering. Work in this direction has been done for the type theory version of CLAM using what is known as the Acc Type [Phillips 91]. It would be worth investigating to what extent the underlying ideas would carry over to first-order logic.

8.4 Rippling

8.4.1 Control

The control of rippling in middle out induction poses a considerable challenge. Most importantly, rippling needs to be controlled in such a way that the search space is kept within reasonable bounds and non-termination is avoided. In Section 5.3.3, this was achieved by a simple mechanism, i.e., allowing only one speculative step. Even so, the search space for rippling in middle-out induction

was large enough to require the rippling to be carefully controlled. The control is currently achieved by carefully ordering the submethods of rippling, in particular by fertilizing as soon as possible.

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 of a proof that requires two speculative steps is

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

Using only one speculative step, a blockage will occur that will prevent fertilisation. For instance, if the speculative step applied to the schematic step case

$$plus(C(\underline{w}), D(\underline{x})) = plus(E(\underline{y}), D(\underline{x})) \leftrightarrow$$
$$plus(C(\underline{w}), F(\underline{z})) = plus(E(\underline{y}), F(\underline{z}))$$

is the application of the plus wave rule

$$plus(\boxed{s(\underline{U})}, V) :\Rightarrow \boxed{s(\underline{plus(U, V)})}$$

to $plus(C(\underline{w}), D(\underline{x}))$, we obtain

$$\boxed{s(\underline{plus(C'(\underline{w}), D(\underline{x}))})} = plus(E(\underline{y}), D(\underline{x})) \leftrightarrow$$
$$plus(\boxed{s(C'(\underline{w}))}, F(\underline{z})) = plus(E(\underline{y}), F(\underline{z}))$$

Once we ripple the right-hand side of the equivalence with the plus wave rule

$$\boxed{s(\underline{plus(C'(\underline{w}), D(\underline{x}))})} = plus(E(\underline{y}), D(\underline{x})) \leftrightarrow$$
$$\boxed{s(\underline{plus(\boxed{C'(\underline{w})}, F(\underline{z}))})} = plus(E(\underline{y}), F(\underline{z})),$$

the rippling is blocked unless we allow a further speculative ripple on the term $plus(E(\underline{y}), D(\underline{x}))$ or the term $plus(E(\underline{y}), F(\underline{z}))$. Such analyses would again best be carried out in the framework of critics [Ireland 92].

8.4.2 Relational Rippling

The class of programs we can synthesize so far is mainly limited because rippling as presented in [Bundy *et al* 93] is based on nested functions. The idea of a wave rule is precisely that it moves terms 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 shared as arguments by more than one literal. Take, for instance, the step cases of the standard functional and relational definitions of list reversal

$$\forall h, t.\ \mathsf{frev}(h :: t) \;=\; \mathsf{append}(\mathsf{frev}(t), h :: \mathsf{nil}) \tag{8.1}$$

$$\forall h, t.\ \mathsf{rrev}(h :: t, l) \;\leftrightarrow\; \exists l'.\ \mathsf{rrev}(t, l') \wedge \mathsf{append}(l', h :: \mathsf{nil}, l)\ . \tag{8.2}$$

While (8.1) gives rise to a wave rule

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

(8.2) currently does not. The rule

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

is not a wave rule in the traditional sense because the skeleton of the left-hand side, $\mathsf{rrev}(T, L)$, and the skeleton of the right-hand side, $\mathsf{rrev}(T, l')$, are not identical. This "almost" wave rule is very similar to the schematic rewrite (6.10) in Section 6 in that they both almost preserve the skeleton. Such rules are clearly within the spirit of rippling, and the notion of rippling is being extended to cope with them.

Preliminary results in relational rippling are reported in [Åhs & Wiggins 94]. Relational rippling would enable us to synthesize a large number of programs that are currently beyond the capabilities of the system.

8.4.3 Logical Wave Rules

In Section 6.1, we presented a mechanism for generating wave rules expressed in terms of propositional connectives. Since the method is based on truth tables,

it cannot be applied to generate wave rules for constructive logics or wave rules for quantifiers. We are currently investigating an approach, originally suggested by Alan Smaill, to use inference rules rather than truth tables to generate wave rules.

The basic idea is to take the partially specified lemma, and prove it as far as possible without committing the unspecified part. Then, the remaining unproven nodes of the proof tree are analyzed to generate a set of constraints for the unspecified part of the lemma. The example of Section 6.1 was to generate a wave rule

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

based on the lemma

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

where $\mathcal{F}$ represents the missing part of the lemma. After applying all possible inference rules without committing $\mathcal{F}(\neg \mathsf{Q})$, we obtain the three unproven nodes

$$\begin{aligned} &\vdash \ \mathcal{F}(\neg \mathsf{Q}), \mathsf{P}, \mathsf{Q} \\ \mathcal{F}(\neg \mathsf{Q}), \mathsf{P} \ &\vdash \ \bot \\ \mathcal{F}(\neg \mathsf{Q}), \mathsf{Q} \ &\vdash \ \bot \ . \end{aligned}$$

The first node can be completed if one of the conclusions were either $\neg \mathsf{Q}$ or $\neg \mathsf{P}$, the second if one of the hypotheses were $\neg \mathsf{P}$ and the third if one of the hypotheses were $\neg \mathsf{Q}$. This knowledge, together with the restriction that $\mathcal{F}(\neg \mathsf{Q})$ must contain $\neg \mathsf{Q}$, indicates that $\mathcal{F}(\neg \mathsf{Q})$ should be $\neg \mathsf{P} \wedge \neg \mathsf{Q}$. A first implementation shows that the approach is feasible, although as yet significantly slower than the approach in Section 6.1. Whether it is possible to correct this through the use of better heuristics is currently being investigated.

8.5 Summary

We have presented a number of topics where further research would increase the power of middle-out synthesis and middle-out induction. The most rewarding extension with a view to synthesizing more complex programs will be relational rippling. With relational rippling, sorting algorithms, for instance, should be within reach of the system. The most rewarding area for further research in middle-out induction lies in improving the control of rippling. The synthesis and verification proofs that we have used middle-out induction for are, from the point of view of the induction, fairly simple, with perhaps the exception of the qr example. To use middle-out induction in more complex theorems, better control of the rippling will be necessary.

Chapter 9

Conclusions

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

- It has led to new applications and a better understanding of middle-out reasoning. It has shown how one source of choice-points in the search for proofs, i.e., higher-order unification, can be avoided by restricting the higher-order terms involved in middle-out reasoning to higher-order patterns. While higher-order patterns are not expressive enough for all applications of middle-out reasoning, the restriction to higher-order patterns is a natural and useful one in this context.

- It has shown that middle-out reasoning provides a mechanism through which proof planning can be used to synthesize logic programs fully automatically.

- It has shown that middle-out reasoning provides an elegant 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.

- It has shown a number of ways in which rippling can be unblocked when the blockage is caused by a missing wave rule on propositional connectives or a nested quantifier.

In the following sections, we discuss these contributions in more detail.

9.1 Improving the Prospects of Middle-Out Reasoning

The use of middle-out reasoning in proof planning is central to this thesis. While middle-out reasoning was suggested early on [Bundy *et al* 90a], it has taken some time for successful applications to appear. The first major investigation and implementation of middle-out reasoning was [Hesketh 91], which uses it for tail-recursion optimization and generalization. Ongoing research includes [Ireland 92, Ireland & Bundy 92, Madden *et al* 93]. The reasons for the delay in applications to arrive have mostly been problems in search control.

One source of choice-points in middle-out reasoning is higher-order unification, which generates a set of unifiers rather than a unique most general unifier. While [Hesketh 91, Ireland 92] have concentrated on heuristics to select a suitable unifier, we have eliminated this search by restricting ourselves to higher-order patterns. Higher-order patterns in proof planning have also been exploited in [Gallagher 93], and we believe that other applications of middle-out reasoning could exploit higher-order patterns as well.

Another source of search is caused by middle-out reasoning itself, which is inherently speculative. In middle-out reasoning, stronger guidance is needed than in non-speculative proof planning to avoid going down unpromising branches of the search tree. This is particularly important when the search tree contains infinite failure branches. In this thesis, we have devised simple heuristics to avoid non-terminating failure branches in rippling, at the cost of overly restricting the search space. Better heuristics may well be achieved using critics [Ireland 92].

One of the dangers of middle-out reasoning is too much speculation. Though we use middle-out reasoning simultaneously for synthesis and induction, this does not cause an explosion of the search space. This is because the two uses occur in isolated parts of the proof. It remains to be seen whether combinations of middle-out reasoning that do affect each other are feasible. This may be necessary, for instance, if the rippling in middle-out induction becomes blocked, and we need to use middle-out reasoning to speculate a missing lemma. Again, critics [Ireland 92] may well provide the answer.

9.2 Automating Logic Program Synthesis

The goal of this thesis was to synthesize logic programs fully automatically. Most of the work in automated reasoning about logic programs has been within the framework of program transformation and analysis techniques such as partial evaluation [Lloyd & Shepherdson 87], abstract interpretation [Cousot & Cousot 77] and unfold/fold transformation [Tamaki & Sato 84], rather than synthesis proper.

Our work is most closely related to work in the automation of unfold/fold transformation and synthesis, in particular to [Lau & Prestwich 88b, Lau & Prestwich 88a, Lau & Prestwich 90] presented in Section 2.2.2. While Lau and Prestwich are able to synthesize more complex programs than we, e.g., sorting algorithms, their system is semi-automatic, relying on the user to provide the recursive structure of the program and that of any auxiliary program. Automating the selection of an appropriate recursive structure has been one of the main emphases of this research. For instance, while we are able to synthesize a subset program (see Section 4.4) fully automatically, the system of Lau and Prestwich requires three instances of user interaction [Lau & Prestwich 88b] for the same synthesis.

The LOPS system [Bibel 80, Bibel & Hörnig 84] is often cited as synthesizing logic programs, though, strictly speaking, it synthesizes functional programs (see

Section 2.2.2). Nevertheless, it is worth noting that LOPS is able to cope with an example on which *Periwinkle* fails, i.e., partitioning lists. *Periwinkle* fails because it cannot yet find the necessary case split, whereas LOPS finds the case split via its GUESS and DOMAIN strategies and their underlying knowledge base.

The system of [Biundo 88] also synthesizes functional programs. One of the main weaknesses of the system is the the most-nested-function heuristic it employs to select induction schemes. In [Biundo 89], for instance, a quotient remainder function is synthesized. The resulting program, however, is unintuitive and extremely inefficient, due to a bad choice of induction. Middle-out induction enables us to find a more appropriate induction scheme, which makes the program easier to understand and more efficient.

Appendix A gives a representative sample of programs that can be synthesized with the approach presented here, as implemented in *Periwinkle*. The type of programs we synthesize fall into three main categories:

- Synthesizing relations from functions. Such relational programs have the advantage that they can be run in various modes. Thus, for instance, a plus relation can also be used for subtraction. Examples programs are $rapp$, $rcount$, $rtimes$ and $rrev$.

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

- Synthesizing more efficient programs from executable but less efficient programs. An example program is $between$.

9.3 Induction Beyond Recursion Analysis

To date, most automated induction theorem provers use recursion analysis or similar techniques to select appropriate induction schemes. As explained in Section 5.1, however, recursion analysis often fails in the context of synthesis conjectures. We have presented an alternative to recursion analysis, i.e., middle-out induction. First results have been encouraging, and the class of theorems which can be proved automatically via induction has been extended. Examples that would fail with recursion analysis are *even* and *qr* (see Appendix A).

9.4 Unblocking

A major issue in rippling is unblocking when the rippling becomes blocked before fertilization is possible. We have presented three unblocking techniques here: Generating logical wave rules, unrolling and very weak fertilization (see Section 6). Without these techniques, the majority of the examples in Appendix A would fail.

9.5 Summary

We have investigated the application of proof planning and middle-out reasoning to the automatic synthesis of logic programs from formal specifications. Proof planning has emerged as a viable vehicle for program synthesis.

In particular, we have established that middle-out reasoning can be used to synthesize programs and that proof planning can be used to automate that synthesis. Furthermore, we have established that middle-out reasoning can also be used to select induction schemes in synthesis proofs.

Although we are still far away from automatically synthesizing complex programs from formal specifications, this thesis constitutes definite progress towards that ultimate goal. We have succeeded in automating some of 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, in program synthesis and elsewhere, is far from exhausted.

Appendix A

Examples

The following examples are the programs listed in Section 7, synthesized by *Periwinkle*. For each example, we give the specification and the program and list the definitions, programs and lemmas used. The axioms for natural numbers and lists are used in all examples and are therefore not listed separately. Planning time is listed for all syntheses, plan execution time for main syntheses only. Time is measured as CPU time on a Sparc station 10 using Quintus Prolog Release 3.1.4 (average of three runs). Appendix B shows the traces (including plans) for three of the examples, namely subset, even and rrev. All examples were planned with a depth-first planner.

Example A.1 (add3) *Addition relation for three numbers*

Specification	Program
$\forall w, x, y, z.\ add3(w, x, y, z) \leftrightarrow$	$\forall w, x, y, z.\ add3(w, x, y, z) \leftrightarrow$
$plus(plus(w, x), y) = z$	$w = 0 \wedge aux(x, y, z) \vee$
	$\exists w'.\ w = s(w') \wedge \exists z'.\ z = s(z') \wedge add3(w', x, y, z')$

Auxiliary Specification	Program
$\forall w, x, y, z.\ aux(x, y, z) \leftrightarrow$	$\forall w, x, y, z.\ aux(x, y, z) \leftrightarrow$
$plus(x, y) = z$	$x = 0 \wedge y = z \vee$
	$\exists x'.\ x = s(x') \wedge z = 0 \wedge false \vee$
	$\exists x'.\ x = s(x') \wedge \exists z'.\ z = s(z') \wedge aux(x', y, z')$

Needs: Definition of plus *(plus function)*

Planning time: 9.5 sec (main) and 7.3 sec (auxiliary)

Plan execution time: 5.8 sec

Example A.2 (app_length) *Combined length of two lists*

Specification	**Program**
$\forall x, y, z.\ \mathsf{app_length}(x, y, z) \leftrightarrow$	$\forall x, y, z.\ \mathsf{app_length}(x, y, z) \leftrightarrow$
$\quad \mathsf{length}(\mathsf{fapp}(x, y)) = z$	$\quad x = \mathsf{nil} \wedge \mathsf{aux}(l, x) \vee$
	$\quad \exists x', x''.\ x = x' :: x'' \wedge \exists z'.\ z = s(z') \wedge$
	$\quad\quad \mathsf{app_length}(x'', y, z')$

Auxiliary Specification	**Program**
$\forall x, y.\ \mathsf{aux}(x, y) \leftrightarrow$	$\forall x, y.\ \mathsf{aux}(x, y) \leftrightarrow$
$\quad \mathsf{length}(x) = y$	$\quad x = \mathsf{nil} \wedge y = 0 \vee$
	$\quad \exists x', x''.\ x = x' :: x'' \wedge \exists y'. y = s(y') \wedge \mathsf{aux}(x'', y')$

Needs: Definition of fapp *and* length *(append and length of list functions)*

Planning time: 10.9 sec (main) and 8.7 sec (auxiliary)

Plan execution time: 5.8 sec (main)

Example A.3 (back) *Back of list*

Specification	**Program**
$\forall x, y.\ \mathsf{back}(x, y) \leftrightarrow$	$\forall x, y.\ \mathsf{back}(x, y) \leftrightarrow$
$\quad \exists z. \mathsf{rapp}(z, x, y)$	$\quad y = \mathsf{nil} \wedge \exists z.\ \mathsf{rapp}(z, x, \mathsf{nil}) \vee$
	$\quad \exists y', y''.\ y = y' :: y'' \wedge (x = y' :: y'' \vee \mathsf{back}(x, y''))$

Needs: Definition of rapp *(append relation)*

Planning time: 14.0 sec

Plan execution time: 5.6 sec

Example A.4 (before) *One element immediately precedes another in a list*

Specification	**Program**
$\forall x, y, z.\ \mathsf{before}(x, y, z) \leftrightarrow$	$\forall x, y, z.\ \mathsf{before}(x, y, z) \leftrightarrow$
$\quad \exists k, l.\ \mathsf{fapp}(k, x :: y :: l) = z$	$\quad z = \mathsf{nil} \wedge \mathsf{false} \vee$
	$\quad \exists z', z''.\ z = z' :: z'' \wedge$
	$\quad\quad (\exists l.\ x = z' \wedge y :: l = z'' \vee \mathsf{before}(x, y, z''))$

Needs: Definition of fapp *(append function) and lemma*

$$\forall k, l.\ \mathsf{fapp}(k, l) = \mathsf{nil} \leftrightarrow k = \mathsf{nil} \wedge l = \mathsf{nil}$$

Planning time: 39.1 sec

Plan execution time: 19.2 sec

Example A.5 (between) *A number lies between two other numbers*

Specification	Program
$\forall x, y, z.\ \mathsf{between}(x, y, z) \leftrightarrow$	$\forall x, y, z.\ \mathsf{between}(x, y, z) \leftrightarrow$
$x \leq y \leq z$	$x = 0 \wedge y \leq z \vee$
	$\exists x'.\ x = s(x') \wedge y = 0 \wedge \mathsf{false} \vee$
	$\exists x'.\ x = s(x') \wedge \exists y'.\ y = s(y') \wedge z = 0 \wedge \mathsf{false} \vee$
	$\exists x'.\ x = s(x') \wedge \exists y'.\ y = s(y') \wedge \exists z'.\ z = s(z') \wedge$
	$\mathsf{between}(x, y, z)$

Needs: Definition of leq *(lesser-or-equal-to relation)*

Planning time: 34.2 sec

Plan execution time: 4.9 sec

Example A.6 (delete) *Delete an element from a list*

Specification	Program
$\forall x, y, z.\ \mathsf{delete}(x, y, z) \leftrightarrow$	$\forall x, y, z.\ \mathsf{delete}(x, y, z) \leftrightarrow$
$\exists k, l.\ \mathsf{fapp}(k, l) = y \wedge$	$y = \mathsf{nil} \wedge z = x :: \mathsf{nil} \vee$
$\mathsf{fapp}(k, x :: l) = z$	$\exists y', y''.\ y = y' :: y'' \wedge z = \mathsf{nil} \wedge \mathsf{false} \vee$
	$\exists y', y''.\ y = y' :: y'' \wedge \exists z', z''.\ z = z' :: z'' \wedge$
	$(x = z' \wedge y' :: y'' = z'' \vee$
	$y' = z' \wedge \mathsf{delete}(x, y'', z''))$

Needs: Definition of fapp *(append function) and lemma*

$$\forall k, l.\ \mathsf{fapp}(k, l) = \mathsf{nil} \leftrightarrow k = \mathsf{nil} \wedge l = \mathsf{nil}$$

Planning time: 95.0 sec

Plan execution time: 31.3 sec

Example A.7 (even) *Even number*

Specification	Program
$\forall x.\ \mathrm{even}(x) \leftrightarrow$	$\forall x.\ \mathrm{even}(x) \leftrightarrow$
$\quad \exists y.\ \mathrm{double}(x, y)$	$\quad x = 0 \wedge \mathrm{true} \vee$
	$\quad x = s(0) \wedge \mathrm{false} \vee$
	$\quad \exists x'.\ x = s(s(x')) \wedge \mathrm{even}(x')$

Needs: Definition of rdouble *(double relation)*

Planning time: 7.2 sec

Plan execution time: 4.7 sec

Example A.8 (insert) *Insert an element before another in a list*

Specification	Program
$\forall w, x, y, z.\ \mathrm{insert}(w, x, y, z) \leftrightarrow$	$\forall w, x, y, z.\ \mathrm{insert}(w, x, y, z) \leftrightarrow$
$\quad \exists k, l.\ \mathrm{fapp}(k, x :: l) = y \wedge$	$\quad y = \mathrm{nil} \wedge \mathrm{false} \vee$
$\quad \mathrm{fapp}(k, w :: x :: l) = z$	$\quad \exists y', y''.\ y = y' :: y'' \wedge z = \mathrm{nil} \wedge \mathrm{false} \vee$
	$\quad \exists y', y''.\ y = y' :: y'' \wedge \exists z', z''.\ z = z' :: z'' \wedge$
	$\quad (w = y' \wedge w = z' \wedge x :: y' = z'' \vee$
	$\quad y' = z' \wedge \mathrm{insert}(w, x, y'', z'')$

Needs: Definition of fapp *(append function) and lemma*

$$\forall k, l.\ \mathrm{fapp}(k, l) = \mathrm{nil} \leftrightarrow k = \mathrm{nil} \wedge l = \mathrm{nil}$$

Planning time: 124.8 sec

Plan execution time: 37.8 sec

Example A.9 (max) *Maximum of a list*

Specification	Program
$\forall x, y.\ \max(x, y) \leftrightarrow$	$\forall x, y.\ \max(x, y) \leftrightarrow$
$\quad \mathrm{member}(x, y) \wedge$	$\quad y = \mathrm{nil} \wedge \mathrm{false}$
$\quad \forall n.\ \mathrm{member}(n, y) \rightarrow x \geq n$	$\quad \exists y', y''.\ y = y' :: y'' \wedge$
	$\quad (x \geq y' \wedge x = y' \wedge \mathrm{aux}(y', x) \vee$
	$\quad x \geq y' \wedge \max(x, y''))$

Auxiliary Specification	**Program**
$\forall x, y.\ aux(x,y) \leftrightarrow$	$\forall x, y.\ aux(x,y) \leftrightarrow$
$\quad \forall n.\ member(n,x) \rightarrow y \geq n$	$\quad x = nil \wedge true \vee$
	$\quad \exists x', x''.\ x = x' :: x'' \wedge y \geq x' \wedge aux(x'', y)$

Needs: Definition of member *and* $\geq$ *(membership and greater-than-or-equal-to relations)*

Planning time: 30.4 sec (main) and 20.1 sec (auxiliary)

Plan execution time: 10.4 sec (main)

Example A.10 (not_member) *Non-membership in a list*

Specification	**Program**
$\forall x, l.\ not_member(x,l) \leftrightarrow$	$\forall x, l.\ not_member(x,l) \leftrightarrow$
$\quad \neg member(x,l)$	$\quad l = nil \wedge true \vee$
	$\quad \exists h, t.\ h::t = l \wedge x \neq h \wedge notmember(x,t)$

Needs: Definition of member *(membership relation)*

Planning time: 9.1 sec

Plan execution times: 3.0 sec

Example A.11 (qr) *Quotient remainder relation*

Specification	**Program**
$\forall w, x, y, z.\ qr(w,x,y,z) \leftrightarrow$	$\forall w, x, y, z.\ qr(w,x,y,z) \leftrightarrow$
$\quad plus(times(w,x),y) = z \wedge less(y,x)$	$\quad w = 0 \wedge y = z \wedge less(y,x) \vee$
	$\quad \exists w'.\ w = s(w') \wedge less(z,x) \wedge false \vee$
	$\quad \exists w'.\ w = s(w') \wedge \exists z'.\ aux1(x,z',z) \wedge$
	$\quad\quad qr(w',x,y,z')$

Auxiliary Specification	**Program**
$\forall w, x, y, z.\ aux(x,y,z) \leftrightarrow$	$\forall w, x, y, z.\ aux(x,y,z) \leftrightarrow$
$\quad plus(x,y) = z$	$\quad x = 0 \wedge y = z \vee$
	$\quad \exists x'.\ x = s(x') \wedge z = 0 \wedge false \vee$
	$\quad \exists x'.\ x = s(x') \wedge \exists z'.\ z = s(z') \wedge aux(x',y,z')$

Needs: Definitions of plus *and* times *(plus and times functions) and* less *(lesser-than relation) and lemmas*

$$\forall x, y, z.\ \mathrm{plus}(x, \mathrm{plus}(y, z)) = \mathrm{plus}(\mathrm{plus}(x, y), z)$$
$$\forall x, y, z.\ \mathrm{plus}(x, y) = \mathrm{plus}(x, z) \leftrightarrow y = z$$
$$\forall x, y, z.\ (\mathrm{less}(z, x) \rightarrow \mathrm{plus}(x, y) = z) \leftrightarrow \mathrm{false}$$

Planning time: 42.1 sec (main) and 29.2 sec (auxiliary) Plan execution time: 12.9 sec (main)

Example A.12 (rapp) *Append relation*

Specification	Program
$\forall x, y, z.\ \mathrm{rapp}(x, y, z) \leftrightarrow$	$\forall x, y, z.\ \mathrm{rapp}(x, y, z) \leftrightarrow$
$\mathrm{fapp}(x, y) = z$	$x = \mathrm{nil} \wedge y = z \vee$
	$\exists x', x''.\ x = x' :: x'' \wedge \exists z'.\ x' :: z' = z \wedge \mathrm{rapp}(x'', y, z'')$

Needs: Definition of fapp *(append function)*

Planning time: 9.4 sec

Plan execution time: 3.7 sec

Example A.13 (rcount) *Counting occurences of an element in a list*

Specification	Program
$\forall x, y, z.\ \mathrm{rcount}(x, y, z) \leftrightarrow$	$\forall x, y, z.\ \mathrm{rcount}(x, y, z) \leftrightarrow$
$\mathrm{fcount}(x, y) = z$	$y = \mathrm{nil} \wedge z = 0 \vee$
	$\exists y', y''.\ y = y' :: y'' \wedge (x \neq y' \wedge \mathrm{rcount}(x, y, z) \vee$
	$x = y' \wedge \exists z'.\ z = s(z') \wedge \mathrm{rcount}(x, y'', z'))$

Needs: Definition fcount *(count function)*

Planning time: 22.2 sec

Example A.14 (rplus) *Addition relation for two numbers*

Specification	Program
$\forall x, y, z.\ \mathrm{rplus}(x, y, z) \leftrightarrow$	$\forall x, y, z.\ \mathrm{rplus}(x, y, z) \leftrightarrow$
$\mathrm{plus}(x, y) = z$	$x = 0 \wedge y = z \vee$
	$\exists x'.\ x = s(x') \wedge \exists z'.\ z = s(z') \wedge \mathrm{rplus}(x', y, z')$

Needs: Definition of plus *(plus function)*

Planning time: 6.0 sec

Plan execution time: 3.6 sec

Example A.15 (rrev) *List reversal*

Specification	Program
$\forall x, y.\ \mathrm{rrev}(x, y) \leftrightarrow$	$\forall x, y.\ \mathrm{rrev}(x, y) \leftrightarrow$
$\mathrm{frev}(x) = y$	$x = \mathrm{nil} \wedge y = \mathrm{nil} \wedge \mathrm{true} \vee$
	$\exists x', x''.\ x = x' :: x'' \wedge \exists z.\ \mathrm{aux}(z, x', y) \wedge \mathrm{rrev}(x'', z)$

Auxiliary Specification	Program
$\forall x, y, z.\ \mathrm{aux}(x, y, z) \leftrightarrow$	$\forall x, y, z.\ \mathrm{aux}(x, y, z) \leftrightarrow$
$\mathrm{fapp}(x, y :: \mathrm{nil}) = z$	$x = \mathrm{nil} \wedge y :: \mathrm{nil} = z \vee$
	$\exists x', x''.\ x = x' :: x'' \wedge z = \mathrm{nil} \wedge \mathrm{false} \vee$
	$\exists x', x''.\ x = x' :: x'' \wedge \exists z', z''.\ z = z' :: z'' \wedge$
	$x' = z' \wedge \mathrm{aux}(x'', y :: \mathrm{nil}, z'')$

Needs: Definition of frev *and* fapp *(list reversal and append functions)*

Planning time: 8.4 sec (main) and 10.6 sec (auxiliary)

Plan execution time: 4.6 sec

Example A.16 (rtimes) *Multiplication relation*

Specification	Program
$\forall x, y, z.\ \mathrm{rtimes}(x, y, z) \leftrightarrow$	$\forall x, y, z.\ \mathrm{rtimes}(x, y, z) \leftrightarrow$
$\mathrm{times}(x, y) = z$	$x = 0 \wedge z = 0 \vee$
	$\exists x'.\ x = s(x') \wedge \exists z'.\ \mathrm{aux}(y, z', z) \wedge \mathrm{rtimes}(x', y, z')$

Auxiliary Specification	Program
$\forall w, x, y, z.\ \mathrm{aux}(x, y, z) \leftrightarrow$	$\forall w, x, y, z.\ \mathrm{aux}(x, y, z) \leftrightarrow$
$\mathrm{plus}(x, y) = z$	$x = 0 \wedge y = z \vee$
	$\exists x'.\ x = s(x') \wedge \exists z'.\ z = s(z') \wedge \mathrm{aux}(x', y, z')$

Needs: Definition of times *(times function)*

Planning time: 6.8 sec (main) and 7.6 sec (auxiliary)

Plan execution time: 5.3 sec

Example A.17 (subset) *Subset relation*

Specification	Program
$\forall x, y.\ \mathrm{subset}(x, y) \leftrightarrow$	$\forall x, y.\ \mathrm{subset}(x, y) \leftrightarrow$
$\quad \forall z.\mathrm{member}(z, x) \rightarrow \mathrm{member}(z, y)$	$\quad x = \mathrm{nil} \wedge \mathrm{true} \vee$
	$\quad \exists x', x''.\ x = x' :: x'' \wedge \mathrm{member}(x', y) \wedge$
	$\quad\quad \mathrm{subset}(x'', y)$

Needs: Definition of member *(membership relation)*

Planning time: 21.2 sec

Plan execution time: 5.0 sec

Appendix B

Traces

This appendix includes the planning traces as produced by *Periwinkle* for the examples *subset*, *even*, and *rrev*. The notational conventions in *Periwinkle* to represent logical symbols in ASCII are summarized in the following table.

ASCII representation	Description
`/\`	Conjunction
`\/`	Disjunction
`=>`	Implication
`<=>`	Equivalence
`not`	Negation
`exists(Var:Type,Formula)`	Existential quantification
`forall(Var:Type,Formula)`	Universal quantification
`Term1=Term2 in Type`	Equality
`==>`	Sequent symbol

The notational convention for annotations in ASCII in *Periwinkle* is that wave fronts are enclosed in quotes, wave holes in curly brackets. Thus, for example, the wave front $\boxed{s(\underline{x})}^{\uparrow}$ would be represented as `''s({x})''`.

Meta-variables generally appear as uppercase letters, occasionally they may appear as unbound Prolog variables, e.g., _123445. Applications of metavariables are written as `((C, x), y)`. The outermost parentheses are sometimes omitted.

The traces are commented to ease reading, and comments are indented and preceded by a % symbol.

B.1 Subset

```
DEPTH: 0
forall(x:nat list,forall(y:nat list,subset(x,y) <=>
     (S,x),y
==>forall(x:nat list,forall(y:nat list,subset(x,y)<=>forall(z:nat,member(z,x)=>member(z,y))))

                                  % Middle-out induction sets up the schematic step case
                                  % and ripples
The middle-out induction conclusion is:
 subset(``A,{x}''<out>,``B,{y}''<out>)<=>
    forall(z:nat,member(z,``A,{x}''<out>)=>member(z,``B,{y}''<out>))
                                  % The member wave rule is applied to the first
                                  % occurrence of member
Applying wave rule program(member(2)) speculatively ... New goal is:
 subset(``A,{x}''<out>,``C:: (D,{y})''<out>)<=>
    forall(z:nat,member(z,``A,{x}''<out>)=>``z=C in nat\/{member(z,``D,{y}''<out>)}''<out>)
                                  % A propositional wave rule is synthesized and
                                  % applied
Applying synthesized propositional wave rule P=>Q\/R <=> Q\/ (P=>R)... New goal is:
 subset(``A,{x}''<out>,``C:: (D,{y})''<out>)<=>
    forall(z:nat,``z=C in nat\/{member(z,``A,{x}''<out>)=>member(z,``D,{y}''<out>)}''<out>)
                                  % Rippling fails. On backtracking, the member
                                  % wave rule is applied to the second occurrence of
                                  % member
Applying wave rule program(member(2)) speculatively ... New goal is:
 subset(``C:: (D,{x})''<out>,``B,{y}''<out>)<=>
    forall(z:nat,``z=C in nat\/{member(z,``D,{x}''<out>)}''<out> =>member(z,``B,{y}''<out>))
                                  % A number of logical wave rules are applied
Applying synthesized propositional wave rule P\/Q=>R <=> ((notP)\/R)/\ (Q=>R)... New goal is:
 subset(``C:: (D,{x})''<out>,``B,{y}''<out>)<=>
    forall(z:nat,``((not z=C in nat)\/member(z,``B,{y}''<out>))/\
       {member(z,``D,{x}''<out>)=>member(z,``B,{y}''<out>)}''<out>)
Applying wave rule dist_all_and ... New goal is:
 subset(``C:: (D,{x})''<out>,``E,{y}''<out>)<=>
    ``forall(z:nat,(not z=C in nat)\/member(z,``E,{y}''<out>))/\
       {forall(z:nat,member(z,``D,{x}''<out>)=>member(z,``E,{y}''<out>))}''<out>
                                  % Weak fertilization succeeds. Middle-out
                                  % induction checks the induction scheme and sets up the
                                  % base and post-fertilization sequents
Applying weak fertilization ... New goal is:
 subset(C::x,y)<=>``forall(z:nat,(not z=C in nat)\/member(z,y))/\{subset(x,y)}''<in>
```

```
SELECTED METHOD at depth 0: mor_induction([v0::x],[x:nat list],
   speculative_step(spec_wave([1,2,2],[program(member(2)),left]))then[ripple(...)])

                                    % The base case. Symbolic evaluation and
                                    % simplification apply

|DEPTH: 1
|v0:nat
|forall(x:nat list,forall(y:nat list,subset(x,y) <=>
     x=nil in nat list/\ (_514576,y) \/
     exists(v1:nat,exists(v2:nat list,x=v1::v2 in nat list/\ (((_514668,v1),v2),y)))
|==>forall(y:nat list,subset(nil,y)<=>forall(z:nat,member(z,nil)=>member(z,y)))
|SELECTED METHOD at depth 1: sym_eval(...)

||DEPTH: 2
||v0:nat
||forall(x:nat list,forall(y:nat list,subset(x,y) <=>
     x=nil in nat list/\ (_514576,y) \/
     exists(v1:nat,exists(v2:nat list,x=v1::v2 in nat list/\ (((_514668,v1),v2),y)))
||==>forall(y:nat list,subset(nil,y)<=>forall(z:nat,false=>member(z,y)))
||SELECTED METHOD at depth 2: simplify([y:nat list],
   subset(nil,y)<=>forall(z:nat,false=>member(z,y)),
   subset(nil,y)<=>true)

|||DEPTH: 3
|||v0:nat
|||forall(x:nat list,forall(y:nat list,subset(x,y) <=>
     x=nil in nat list/\ (_514576,y) \/
     exists(v1:nat,exists(v2:nat list,x=v1::v2 in nat list/\ (((_514668,v1),v2),y)))
|||==>forall(y:nat list,subset(nil,y)<=>true)
|||TERMINATING METHOD at depth 3: synthesis

                                    % The synthesis method completed the base case
                                    % and partially instantiated the program. Now, the
                                    % post-fertilization case. Simplifaction applies

|DEPTH: 1
|v0:nat
|ih:[USED,v0:subset(x,y)<=>forall(z:nat,member(z,x)=>member(z,y))]
|
|x:nat list
|y:nat list
|forall(x:nat list,forall(y:nat list,subset(x,y) <=>
     x=nil in nat list/\true \/
     exists(v1:nat,exists(v2:nat list,x=v1::v2 in nat list/\ (((_514668,v1),v2),y)))
```

```
|==>subset(v0::x,y)<=>forall(z:nat,(not z=v0 in nat)\/member(z,y))/\subset(x,y)
|SELECTED METHOD at depth 1: simplify([v0:nat,x:nat list,y:nat list],
   subset(v0::x,y)<=>forall(z:nat,(not z=v0 in nat)\/member(z,y))/\subset(x,y),
   subset(v0::x,y)<=>member(v0,y)/\subset(x,y))

||DEPTH: 2
||v0:nat
||ih:[USED,v0:subset(x,y)<=>forall(z:nat,member(z,x)=>member(z,y))]
||
||x:nat list
||y:nat list
||forall(x:nat list,forall(y:nat list,subset(x,y) <=>
     x=nil in nat list/\true \/
     exists(v1:nat,exists(v2:nat list,x=v1::v2 in nat list/\ (((_514668,v1),v2),y)))
||==>subset(v0::x,y)<=>member(v0,y)/\subset(x,y)
||TERMINATING METHOD at depth 2: synthesis

                          % The synthesis method completed the post-fertilization case,
                          % and thus the proof and program

The initial program is:
forall(x:nat list,forall(y:nat list,subset(x,y) <=>
     x=nil in nat list/\true \/
     exists(v1:nat,exists(v2:nat list,x=v1::v2 in nat list/\member(v1,y)/\subset(v2,y)))


No auxiliary syntheses required.

The final program for subsetsynth is:

forall(x:nat list,forall(y:nat list,subset(x,y) <=>
     x=nil in nat list/\true \/
     exists(v1:nat,exists(v2:nat list,x=v1::v2 in nat list/\member(v1,y)/\subset(v2,y)))

Planning took 14.317 sec.
Tactic subsetsynth/0 (source file *unsaved*)
subsetsynth :-
    mor_induction([v0::x],[x:nat list],
                 speculative_step(spec_wave([1,2,2],[program(member(2)),left]))
                    then [ripple(prop_wave([2,2],(Q\/R=>S)<=> ((not Q)\/S)/\ (R=>S))
                       then[wave([2],[dist_all_and,left])
                          then[fertilize(weak,fertilize_then_ripple(fertilize_left_or_right(
                             right,[weak_fertilize(right,<=>,[2],A)]))))]])])
        then  [ sym_eval([eval_def([1,2,2],program(member(1)))])
                   then  [ simplify([y:nat list],
```

```
                 subset(nil,y)<=>forall(z:nat,false=>member(z,y)),
                 subset(nil,y)<=>true)
                then  [ synthesis ] ],
    simplify([v0:nat,x:nat list,y:nat list],
       subset(v0::x,y)<=>forall(z:nat,(not z=v0 in nat)\/member(z,y))/\subset(x,y),
       subset(v0::x,y)<=>member(v0,y)/\subset(x,y))
        then  [ synthesis ]
  ].

.     Plan stored as tactic subsetsynth; to execute, enter "apply(subsetsynth)"
```

B.2 Even

```
DEPTH: 0
forall(x:nat,even(x) <=>
     S,x
==>forall(x:nat,even(x)<=>exists(y:nat,rdouble(x,y)))

                              % Middle-out induction sets up the schematic step case
                              % and ripples
The middle-out induction conclusion is:
 even(''A,{x}''<out>)<=>exists(y:nat,rdouble(''A,{x}''<out>,y))
                              % Unrolling is applied to y, enabling the
                              % wave rule for rdouble to apply
Unrolling for wave rule program(rdouble(4)) ... New goal is:
 even(''A,{x}''<out>)<=>
    ''rdouble(''A,{x}''<out>,0)\/{exists(y:nat,rdouble(''A,{x}''<out>,''s({y})''<out>))}''<out>
Applying wave rule program(rdouble(4)) ... New goal is:
 even(''s(s((B,{x})))''<out>)<=>
    ''rdouble(''s(s((B,{x})))''<out>,0)\/{exists(y:nat,rdouble(''B,{x}''<out>,y))}''<out>
Applying weak fertilization ... New goal is:
 even(s(s(x)))<=>''rdouble(s(s(x)),0)\/{even(x)}''<in>
                              % Weak fertilization succeeds. Middle-out
                              % induction checks the induction scheme and sets up the
                              % base and post-fertilization sequents

SELECTED METHOD at depth 0: mor_induction([s(s(x))],[x:nat],
   speculative_step(unroll([2],wave([2,2,2],[program(rdouble(4)),left])))then[ripple(...)])

                              % The first base case. Symbolic evaluation and
                              % simplification apply

|DEPTH: 1
|forall(x:nat,even(x) <=>
     x=0 in nat/\_493520 \/
     x=s(0)in nat/\_493657 \/
     exists(v0:nat,x=s(s(v0))in nat/\ (_493707,v0))
|==>even(0)<=>exists(y:nat,rdouble(0,y))
|SELECTED METHOD at depth 1: sym_eval(...)

||DEPTH: 2
||forall(x:nat,even(x) <=>
     x=0 in nat/\_493520 \/
     x=s(0)in nat/\_493657 \/
```

```
     exists(v0:nat,x=s(s(v0))in nat/\ (_493707,v0))
||==>even(0)<=>exists(y:nat,y=0 in nat)
||SELECTED METHOD at depth 2: simplify([],even(0)<=>exists(y:nat,y=0 in nat),even(0)<=>true)

|||DEPTH: 3
|||forall(x:nat,even(x) <=>
     x=0 in nat/\_493520 \/
     x=s(0)in nat/\_493657 \/
     exists(v0:nat,x=s(s(v0))in nat/\ (_493707,v0))
|||==>even(0)<=>true
|||TERMINATING METHOD at depth 3: synthesis

                              % The synthesis method completed the first base case
                              % and partially instantiated the program. Now, the
                              % second base case. Symbolic evaluation and simplifaction apply

|DEPTH: 1
|forall(x:nat,even(x) <=>
     x=0 in nat/\true \/
     x=s(0)in nat/\_493657 \/
     exists(v0:nat,x=s(s(v0))in nat/\ (_493707,v0))
|==>even(s(0))<=>exists(y:nat,rdouble(s(0),y))
|SELECTED METHOD at depth 1: sym_eval(...)

||DEPTH: 2
||forall(x:nat,even(x) <=>
     x=0 in nat/\true \/
     x=s(0)in nat/\_493657 \/
     exists(v0:nat,x=s(s(v0))in nat/\ (_493707,v0))
||==>even(s(0))<=>exists(y:nat,false)
||SELECTED METHOD at depth 2: simplify([],even(s(0))<=>exists(y:nat,false),even(s(0))<=>false)

|||DEPTH: 3
|||forall(x:nat,even(x) <=>
     x=0 in nat/\true \/
     x=s(0)in nat/\_493657 \/
     exists(v0:nat,x=s(s(v0))in nat/\ (_493707,v0))
|||==>even(s(0))<=>false
|||TERMINATING METHOD at depth 3: synthesis

                              % The synthesis method completed the second base case
                              % and partially instantiated the program. Now, the
                              % post-fertilization case. Symbolic evaluation and simplifaction
                              % apply
```

```
|DEPTH: 1
|ih:[USED,v0:even(x)<=>exists(y:nat,rdouble(x,y))]
|
|x:nat
|forall(x:nat,even(x) <=>
     x=0 in nat/\true \/
     x=s(0)in nat/\false \/
     exists(v0:nat,x=s(s(v0))in nat/\ (_493707,v0))
|==>even(s(s(x)))<=>rdouble(s(s(x)),0)\/even(x)
|SELECTED METHOD at depth 1: sym_eval(...)

||DEPTH: 2
||ih:[USED,v0:even(x)<=>exists(y:nat,rdouble(x,y))]
||
||x:nat
||forall(x:nat,even(x) <=>
     x=0 in nat/\true \/
     x=s(0)in nat/\false \/
     exists(v0:nat,x=s(s(v0))in nat/\ (_493707,v0))
||==>even(s(s(x)))<=>false\/even(x)
||SELECTED METHOD at depth 2: simplify([x:nat],
   even(s(s(x)))<=>false\/even(x),
   even(s(s(x)))<=>even(x))

|||DEPTH: 3
|||ih:[USED,v0:even(x)<=>exists(y:nat,rdouble(x,y))]
|||
|||x:nat
|||forall(x:nat,even(x) <=>
     x=0 in nat/\true \/
     x=s(0)in nat/\false \/
     exists(v0:nat,x=s(s(v0))in nat/\ (_493707,v0))
|||==>even(s(s(x)))<=>even(x)
|||TERMINATING METHOD at depth 3: synthesis

                         % The synthesis method completed the post-fertilization case,
                         % and thus the proof and program

The initial program is:
forall(x:nat,even(x) <=>
     x=0 in nat/\true \/
     x=s(0)in nat/\false \/
     exists(v0:nat,x=s(s(v0))in nat/\even(v0))
```

```
No auxiliary syntheses required.


The final program for evensynth is:


forall(x:nat,even(x) <=>
     x=0 in nat/\true \/
     x=s(0)in nat/\false \/
     exists(v0:nat,x=s(s(v0))in nat/\even(v0))


Planning took 6.933 sec.
Tactic evensynth/0 (source file *unsaved*)
evensynth :-
    mor_induction([s(s(x))],[x:nat],
                 speculative_step(unroll([2],wave([2,2,2],[program(rdouble(4)),left])))
                    then[ripple(fertilize(weak,fertilize_then_ripple(fertilize_left_or_right(
                       right,[weak_fertilize(right,<=>,[2],A)])))))])
         then  [ sym_eval([eval_def([2,2],program(rdouble(1)))])
                    then  [ simplify([],even(0)<=>exists(y:nat,y=0 in nat),even(0)<=>true)
                              then  [ synthesis ] ],
                sym_eval([eval_def([2,2],program(rdouble(2)))])
                    then  [ simplify([],even(s(0))<=>exists(y:nat,false),even(s(0))<=>false)
                              then  [ synthesis ] ],
                sym_eval([eval_def([1,2],program(rdouble(3)))])
                    then  [ simplify([x:nat],
                               even(s(s(x)))<=>false\/even(x),
                               even(s(s(x)))<=>even(x))
                              then  [ synthesis ] ]
              ].

Plan stored as tactic evensynth; to execute, enter "apply(evensynth)"
```

B.3 Reverse

```
DEPTH: 0
forall(k:nat list,forall(l:nat list,rrev(k,l) <=>
     (S,k),l
==>forall(k:nat list,forall(l:nat list,rrev(k,l)<=>frev(k)=l in nat list))

                                 % Middle-out induction sets up the schematic step case
                                 % and ripples
The middle-out induction conclusion is:
 rrev(''A,{k}''<out>,''B,{l}''<out>)<=>frev(''A,{k}''<out>)=''B,{l}''<out>in nat list
                                  % The wave rule for frev is applicable
Applying wave rule def(frev(2)) speculatively ... New goal is:
 rrev(''D:: (C,{k})''<out>,''B,{l}''<out>)<=>
    ''fapp({frev(''C,{k}''<out>)},D::nil)''<out> =''B,{l}''<out>in nat list
Applying very weak fertilization ... New goal is:
 rrev(''C::{k}''<out>,l)<=>exists(e0:nat list,fapp(e0,C::nil)=l in nat list/\rrev(k,e0))
                                 % Very weak fertilization succeeds. Middle-out
                                 % induction checks the induction scheme and sets up the
                                 % base and post-fertilization sequents

SELECTED METHOD at depth 0: mor_induction([v0::k],[k:nat list],
   speculative_step(spec_wave([1,1,2],[def(frev(2)),left]))then[ripple(...)])

                                 % The base case. Symbolic evaluation applies

|DEPTH: 1
|v0:nat
|forall(k:nat list,forall(l:nat list,rrev(k,l) <=>
     k=nil in nat list/\ (_423640,l) \/
     exists(v1:nat,exists(v2:nat list,k=v1::v2 in nat list/\ (((_423732,v1),v2),l)))
|==>forall(l:nat list,rrev(nil,l)<=>frev(nil)=l in nat list)
|SELECTED METHOD at depth 1: sym_eval(...)

||DEPTH: 2
||v0:nat
||forall(k:nat list,forall(l:nat list,rrev(k,l) <=>
     k=nil in nat list/\ (_423640,l) \/
     exists(v1:nat,exists(v2:nat list,k=v1::v2 in nat list/\ (((_423732,v1),v2),l)))
||==>forall(l:nat list,rrev(nil,l)<=>nil=l in nat list)
||TERMINATING METHOD at depth 2: synthesis

                                 % The synthesis method completed the base case
```

```
                              % and partially instantiated the program. Now, the
                              % post-fertilization case

|DEPTH: 1
|v0:nat
|
|ih:[RAW,v0:rrev(k,l)<=>frev(k)=l in nat list]
|k:nat list
|l:nat list
|forall(k:nat list,forall(l:nat list,rrev(k,l) <=>
     k=nil in nat list/\nil=l in nat list \/
     exists(v1:nat,exists(v2:nat list,k=v1::v2 in nat list/\ (((_423732,v1),v2),l)))
|==>rrev(''v0::{k}''<out>,l)<=>exists(e0:nat list,fapp(e0,v0::nil)=l in nat list/\rrev(k,e0))
|TERMINATING METHOD at depth 1: synthesis

                              % The synthesis method completed the post-fertilization case,
                              % and thus the proof and the initial program.
                              % However, an auxiliary synthesis is required for
                              % fapp.


The initial program is:
forall(k:nat list,forall(l:nat list,rrev(k,l) <=>
     k=nil in nat list/\nil=l in nat list \/
     exists(v1:nat,exists(v2:nat list,k=v1::v2 in nat list/\
          exists(e0:nat list,fapp(e0,v1::nil)=l in nat list/\rrev(v2,e0))))

Auxiliary syntheses required for:

fapp(e0,v1::nil)=l in nat list

DEPTH: 0
forall(e0:nat list,forall(v1:nat,forall(l:nat list,auxlrrevsynth(e0,v1,l) <=>
     ((S,e0),v1),l
==>forall(e0:nat list,forall(v1:nat,forall(l:nat list,auxlrrevsynth(e0,v1,l)<=>
     fapp(e0,v1::nil)=l in nat list)))

                              % Middle-out induction sets up the schematic step case
                              % and ripples
The middle-out induction conclusion is:
 auxlrrevsynth(''A,{e0}''<out>,''B,{v1}''<out>,''C,{l}''<out>)<=>
    fapp(''A,{e0}''<out>,''B,{v1}''<out> ::nil)=''C,{l}''<out>in nat list
                              % The wave rule for fapp applies
Applying wave rule def(fapp(3)) speculatively ... New goal is:
 auxlrrevsynth(''D:: (E,{e0})''<out>,''B,{v1}''<out>,''C,{l}''<out>)<=>
```

```
    ``D::{fapp(``E,{e0}''<out>,``B,{v1}''<out> ::nil)}''<out> =``C,{l}''<out>in nat list
Applying very weak fertilization ... New goal is:
 auxlrrevsynth(``D::{e0}''<out>,v1,l)<=>
    exists(e1:nat list,D::e1=l in nat list/\auxlrrevsynth(e0,v1,e1))
                              % Weak fertilization succeeds. Middle-out
                              % induction checks the induction scheme and sets up the
                              % base and post-fertilization sequents

SELECTED METHOD at depth 0: mor_induction([v0::e0],[e0:nat list],
   speculative_step(spec_wave([1,1,2],[def(fapp(3)),left]))then[ripple(...)])

                              % The base case. Symbolic evaluation applies

|DEPTH: 1
|v0:nat
|forall(e0:nat list,forall(v1:nat,forall(l:nat list,auxlrrevsynth(e0,v1,l) <=>
     e0=nil in nat list/\ ((_462369,v1),l) \/
     exists(v2:nat,exists(v3:nat list,e0=v2::v3 in nat list/\ ((((_462478,v2),v3),v1),l)))
|==>forall(v1:nat,forall(l:nat list,auxlrrevsynth(nil,v1,l)<=>fapp(nil,v1::nil)=l in nat list))
|SELECTED METHOD at depth 1: sym_eval(...)

||DEPTH: 2
||v0:nat
||forall(e0:nat list,forall(v1:nat,forall(l:nat list,auxlrrevsynth(e0,v1,l) <=>
     e0=nil in nat list/\ ((_462369,v1),l) \/
     exists(v2:nat,exists(v3:nat list,e0=v2::v3 in nat list/\ ((((_462478,v2),v3),v1),l)))
||==>forall(v1:nat,forall(l:nat list,auxlrrevsynth(nil,v1,l)<=>v1::nil=l in nat list))
||TERMINATING METHOD at depth 2: synthesis

                              % The synthesis method completed the base case
                              % and partially instantiated the program. Now, the
                              % post-fertilization case

|DEPTH: 1
|v0:nat
|
|ih:[RAW,v0:auxlrrevsynth(e0,v1,l)<=>fapp(e0,v1::nil)=l in nat list]
|e0:nat list
|v1:nat
|l:nat list
|forall(e0:nat list,forall(v1:nat,forall(l:nat list,auxlrrevsynth(e0,v1,l) <=>
     e0=nil in nat list/\v1::nil=l in nat list \/
     exists(v2:nat,exists(v3:nat list,e0=v2::v3 in nat list/\ ((((_462478,v2),v3),v1),l)))
|==>auxlrrevsynth(``v0::{e0}''<out>,v1,l)<=>
      exists(e1:nat list,v0::e1=l in nat list/\auxlrrevsynth(e0,v1,e1))
```

```
|TERMINATING METHOD at depth 1: synthesis

                              % The synthesis method completed the post-fertilization case,
                              % and thus the proof and the initial auxiliary program.
                              % No more auxiliary syntheses are required, and
                              % the final program is assembled.

The initial program is:
forall(e0:nat list,forall(v1:nat,forall(l:nat list,aux1rrevsynth(e0,v1,l) <=>
     e0=nil in nat list/\v1::nil=l in nat list \/
     exists(v2:nat,exists(v3:nat list,e0=v2::v3 in nat list/\
          exists(e1:nat list,v2::e1=l in nat list/\aux1rrevsynth(v3,v1,e1))))


No auxiliary syntheses required.

The final program for aux1rrevsynth is:

forall(e0:nat list,forall(v1:nat,forall(l:nat list,aux1rrevsynth(e0,v1,l) <=>
     e0=nil in nat list/\v1::nil=l in nat list \/
     exists(v2:nat,exists(v3:nat list,e0=v2::v3 in nat list/\
          exists(e1:nat list,v2::e1=l in nat list/\aux1rrevsynth(v3,v1,e1))))

Planning took 8.467 sec.

The final program for rrevsynth is:

forall(k:nat list,forall(l:nat list,rrev(k,l) <=>
     k=nil in nat list/\nil=l in nat list \/
     exists(v1:nat,exists(v2:nat list,k=v1::v2 in nat list/\
          exists(e0:nat list,aux1rrevsynth(e0,v1,l)/\rrev(v2,e0))))

Planning took 7.067 sec.

Tactic rrevsynth/0 (source file *unsaved*)
rrevsynth :-
    mor_induction([v0::k],[k:nat list],
                 speculative_step(spec_wave([1,1,2],[def(frev(2)),left]))
                    then[ripple(fertilize(very_weak,[right,left,[],[1]]))])
        then  [ sym_eval([reduction([1,1,2],def(frev(1)))])
                   then  [ synthesis ],
                synthesis
              ].

Plan stored as tactic rrevsynth; to execute, enter "apply(rrevsynth)"
```

Appendix C

Methods and Submethods

C.1 Methods

C.1.1 Mor_induction

```
method( mor_induction( Scheme, Vars, speculative_step( S ) then
                                                AllRippleTacs ) ,
% sequent in
        Hyps==>Goal,

% preconditions
        [
         induction,
                         % strip universal quantifiers
         matrix( UVars, Matrix, Goal ),

                         % construct schematic step case and do some
                         % pretty printing housekeeping
         mor_induction_conclusion( UVars, Matrix, MORGoal, MORTerms,
                        PrintInfo ),
         hfree( [IndHypVar], UVars ),
         append( [ ih:[ihmarker(_,[]), IndHypVar:Matrix],
              meta_vars:PrintInfo | UVars ], Hyps, NewHyps ),
         format( 'The middle-out induction conclusion is:~n', [] ),
         print_mor_conclusion( MORGoal, PrintInfo ),

                         % apply one speculative step
```

```
        applicable_submethod( NewHyps ==> MORGoal,
             speculative_step( S ), _, SpecSeqs ),

                                 % ripple all resulting sequents
        repeat(
          SpecSeqs,
          Goal1 :=> SubGoals1,
          ripple( RippleTac ),
          applicable( Goal1, ripple( RippleTac ), _, SubGoals1 ),
          AllRippleTacs,
          PostFertSeqs
              ),
                                 % retrieve induction schema
        induction_terms( UVars, MORTerms, Scheme, Vars ),
        mor_scheme( Scheme, Vars, Hyps==>Goal, BaseSequents,
               BodyScheme, NewVars )
       ],

% postconditions
       [
        append( BaseSequents, PostFertSeqs, AllSequents ),
% unify program body and program structure
 instantiate_body( Hyps, BodyScheme ),
% Housekeeping: normalize
        cleanup_sequents( AllSequents, CleanSequents ),
        add_hyps_to_sequents( NewVars, CleanSequents, NewSequents )
       ],

% sequents out
       NewSequents,

% tactic
       mor_induction( Scheme, Vars, speculative_step( S ) then
                                                  AllRippleTacs ).
```

C.1.2 Ripple

```
method( ripple( SubPlan ),

% sequent in
        H==>G,

% preconditions
        [
          wave_fronts( _, WFs, G ),
          \+ WFs = [],
          repeat( [H==>G],
                  Goal :=> SubGoals,
                  Method,
                  ( member( Method,
                            [  fertilize( _, _ ),
                               wave(_,_),
                               prop_wave(_,_),
                               unblock(_,_,_,_)
                            ] ),
                    applicable_submethod( Goal, Method, _, SubGoals )
                  ),
                  [SubPlan],
                  SubGoals
                ),!,
          SubPlan ~ idtac
        ],

% postconditions
        [
        ],

% sequents out
        SubGoals,

% tactic
        ripple( SubPlan )
      ).
```

C.1.3 Synthesis

```
method( synthesis,

% sequent in
        Hyps==>WGoal,

% preconditions
        [
                                % Housekeeping. Check that we are doing a
                                % synthesis, find the program and normalize it
         synthesis,
         hyp( synthesizing:(_,UglyHyp), Hyps ),
         cleanup_formula( UglyHyp, Hyp ),
                                % Strip annotation from goal
         wave_fronts( Goal, _, WGoal ),
                                % Find appropriate instantiation of
                                % universal variables of program
         matrix( GVars, GoalMatrix, Goal ),
         universal_variable_names( Hyps==>Goal, UVars ),
         universal_variables( Hyps==>Goal, UVarTs ),
         clamaccess:left_formula( GoalMatrix, GoalHalf ),
         matrix( HVars, HypMatrix, Hyp ),
         clamaccess:left_formula( HypMatrix, HypHalf ),
         matrix( HVars, HypHalf, QHypHalf ),
         instantiate( QHypHalf, GoalHalf, Vals ),
         findall( Name, member( Name:_, HVars ), Names ),
         s( HypMatrix, Vals, Names, TempProgMatrix ),
         matrix( HVars, TempProgMatrix, TempProg ),
                                % Simplify the program as far as possible
         ( ( applicable( Hyps ==> TempProg, sym_eval( S ), _, Concl ),
             Concl = [_ ==> TempProg2] )
          v
             TempProg = TempProg2
         ),
         matrix( _, TempProg2Matrix, TempProg2 ),
         simplify1( Hyps, TempProg2Matrix, TempProg3Matrix ),
         simplify( TempProg3Matrix, Prog2, UVarTs ),
                                % and finally unify
         match2( Prog2, GoalMatrix, UVars )
        ],

% postconditions
        [
                                % Housekeeping. Normalize and save program for
```

```
                        % posterity
     cleanup_formula( Hyp, Hyp1 ),
     store_program( Hyp1 )
    ],

% sequents out
    [],

% tactic
    synthesis
   ).
```

C.2 Submethods

C.2.1 Speculative_step

```
submethod( speculative_step( Type ),

% sequent in
       Seq,

% preconditions
       [
         ( ( Type = spec_wave( _, _ ) ) v
           ( Type = unroll( _, _ ) ) v
           ( Type = casesplit( _ ) )
         ),
         applicable_submethod( Seq, Type , _, NewG )
       ],

% postconditions
       [
       ],

% sequents out
       NewG,

% tactic
       speculative_step( Type )
      ).
```

C.2.2 Prop_wave

```
submethod( prop_wave( MatrixPos, LHS<=>RHS ),

% sequent in
        Sequent,

% preconditions
        [
                                % Housekeeping. Normalize sequent
         cleanup_sequent( Sequent, Hyps ==> Goal ),
                                % Find a wave term with a hard wave front
         matrix( QVars, Matrix, Goal ),
         universal_variable_names( Hyps==>Goal, UVars ),
         wave_terms_at( Matrix, MatrixPos, WTerm ),
         wave_fronts( Term, WFs, WTerm ),
         findall( [A]-B/[C, D], member( [A]-B/[C, D], WFs ), WF1s ),
         \+ WF1s = [],
                                % and find a propositional wave rule
         sort_it_out( Term, WF1s, LHS, Hole_RHS, RemovedWFs ),
         find_prop_lemma( LHS, Hole_RHS, RHS )
        ],

% postconditions
        [
                                % find the right annotation and instantiation
         wave_fronts( LHS, RemovedWFs, WLHS ),
         varnumbers( ( LHS, WLHS, Hole_RHS, RHS ),
                     ( LHS1, WLHS1, Hole_RHS1, RHS1 ) ),
         WLHS1 = WTerm,
         exp_at( RHS1, HolePos, Hole_RHS1 ),
         wave_fronts( RHS1, [[]-[HolePos]/[hard, out]], NWRight ),
                                % and apply the rewrite
         replace( MatrixPos, NWRight, Matrix, TempMatrix ),
                                % Housekeeping. Normalize and pretty print
         cleanup_formula( TempMatrix, CleanMatrix ),
         matrix( QVars, CleanMatrix, NewGoal ),
         format(
     'Applying synthesized propositional wave rule ~p <=> ~p... New goal is:~n',
                [LHS, RHS] ),
         print_new_conclusion( Hyps, NewGoal, NewHyps )
        ],

% sequents out
        [
```

```
        NewHyps ==> NewGoal
       ],

% tactic
       prop_wave( MatrixPos, LHS<=>RHS )
      ).
```

C.2.3 Unroll

```
submethod( unroll( Pos, wave( PosW, [RuleName,Dir] ) ),

% sequent in

            Hyps==>Goal,

% preconditions
        [
                % Pick a quantified subexpression of the goal
        exp_at( Goal, Pos, Exp ),
        nonvar( Exp ),
        clamaccess:binding_operator( Exp, Op ),
                % Pick a subexpression of the body of the quantified formula
        clamaccess:bound_formula( Exp, Formula ),
        clamaccess:bound_variable( Exp, Var ),
        clamaccess:variable_name( Var, Name ),
        clamaccess:variable_type( Var, Type ),
        exp_at( Formula, _, SubExp ),
        nonvar( SubExp ),
                % Pick an argument of the subexpression that corresponds
                % to the quantified variable
        clamaccess:arguments( SubExp, Args ),
        nth1( N, Args, Name ),
        clamaccess:functor( SubExp, Functor ),
                % Construct the pattern of the compound case
        length( Args, M ),
        length( NewArgs, M ),
        unroll_util:step( Type, Name, Step, NewVars ),
        nth1( N, NewArgs, Step ),
        clamaccess:functor( Schema, Functor ),
        clamaccess:arguments( Schema, NewArgs ),
                % And make sure there is a wave rule that is potentially
                % applicable
        wave_occ( Goal, Name, WavePos, P1, P2, P3, RuleName:_ ),
        wave_rule( RuleName, genw( _, [_-[]-_-_] ), [] => Schema :=> _ )
        ],

% postconditions
        [
                % Construct case split--similar to base and step cases
                % First argument--replace Name with corresponding base
        unroll_util:base( Type, Base ),
        replace_all( Name, Base, Formula, Left ),
```

```
                % Second argument--replace Arg with corresponding compound
                % term, prepend any new existential variables and annotate
%       unroll_util:step( Type, Name, Step, NewVars ), % done already above
        replace_all( Name, Step, Formula, RightMatrix ),
        append( NewVars, [Var], AllNQVars ),
        ( ( clamaccess:existential_quantifier( _, Op ),
             pw2clam:exmatrix( AllNQVars, RightMatrix, Right ) ) v
          ( clamaccess:universal_quantifier( _, Op ),
             pw2clam:univmatrix( AllNQVars, RightMatrix, Right ) )
        ),
                % Warning: This works only as long as there are only
                % numbers and lists
        (  NewVars = [_|_]
        -> wave_fronts( Right, [[]-[[2]]/[hard,out]], WRight )
        ;  WRight = Right
        ),
        ( ( clamaccess:existential_quantifier( _, Op ),
             clamaccess:disjunction( New, _ ) ) v
          ( clamaccess:universal_quantifier( _, Op ),
             clamaccess:conjunction( New, _ ) )
        ),
        clamaccess:right_formula( New, WRight ),
        clamaccess:left_formula( New, Left ),
        wave_fronts( New, [[]-[[2]]/[hard,out]], NewExp ),
                % Do the rewrite
        replace( Pos, NewExp, Goal, NewGoal ),
        matrix( _, Matrix, NewGoal ),
% Housekeeping. Pretty printing
        format( 'Unrolling for wave rule ~p ... New goal is:~n',
             RuleName ),
        print_new_conclusion( Hyps, Matrix, NewHyps ),
        % Finally, apply the wave rule you unrolled for
        applicable_submethod( NewHyps==>NewGoal, wave( PosW, [RuleName,Dir] ),
                 _, Seqs )
        ],

% sequents out
        Seqs,

% tactic
        unroll( Pos, wave( PosW, [RuleName,Dir] ) )
      ).
```

C.2.4 Very_weak_fertilize

```
submethod( very_weak_fertilize( [Dir1, Dir2, EqPos, HolePos] ),

%sequent in
        Hyps==>Goal,

% preconditions
        [
         matrix( QVars, GoalMatrix, Goal ),
                % goal is an equivalence
         universal_variables( Hyps==>Goal, Vars ),
         clamaccess:equivalence( GoalMatrix, _ ),
         clamaccess:left_formula( GoalMatrix, LeftGoalMatrix ),
         clamaccess:right_formula( GoalMatrix, RightGoalMatrix ),
                % allow for blockage on right or left side of equivalence
         ( ( Dir1 = right,
             GoalFertilizationSide = RightGoalMatrix,
             GoalOtherSide = LeftGoalMatrix ) ;
           ( Dir1 = left,
     GoalFertilizationSide = LeftGoalMatrix,
             GoalOtherSide = RightGoalMatrix ) ),
                % fertilization side is surrounded
                % subterm of fertilization side is equality
         exp_at( GoalFertilizationSide, EqPos, GoalEquality ),
         nonvar( GoalEquality ),
         clamaccess:equality( GoalEquality, _ ),
         clamaccess:left_term( GoalEquality, LeftGoalTerm ),
         clamaccess:right_term( GoalEquality, RightGoalTerm ),
                % allow for blockage on left or right side of equality
         ( ( Dir2 = left,
     LeftGoalTerm = WGoalTerm,
             RightGoalTerm = WGoalVariable ) ;
           ( Dir2 = right,
             LeftGoalTerm = WGoalVariable,
             RightGoalTerm = WGoalTerm ) ),
                % and a wave front is blocking one side of the equality
         wave_fronts( GoalTerm, WFs, WGoalTerm ),
         hard_wave_fronts( WGoalTerm, HWFs ),
         member( []-[HolePos]/[hard,out], HWFs )
        ],
% postconditions
        [
                % the other side of the equality must be a sink and
                % not a potential wave front--instantiate wave front
```

```
        % to identity function
wave_fronts( GoalVariable, _, WGoalVariable ),
GoalVariable = ( lambda( X, X ) , Sink ),

        % find an induction hypothesis
hyp(_:Hyp, Hyps),
matrix(_, HypMatrix, Hyp ),
clamaccess:equivalence( HypMatrix, _ ),
clamaccess:left_formula( HypMatrix, LeftHypMatrix ),
clamaccess:right_formula( HypMatrix, RightHypMatrix ),
        % allow for fertilization left-to-right or right-to-left
( ( HypEquality = RightHypMatrix,
    HypOtherSide = LeftHypMatrix ) ;
  ( HypEquality = LeftHypMatrix,
    HypOtherSide = RightHypMatrix ) ),
nonvar( HypEquality ),
        % that is suitable because one side is an equality
clamaccess:equality( HypEquality, _ ),
clamaccess:left_term( HypEquality, LeftHypTerm ),
clamaccess:right_term( HypEquality, RightHypTerm ),
        % and one side of the equality is the sink
( ( LeftHypTerm = WHypTerm,
    RightHypTerm == Sink ) ; % test
  ( LeftHypTerm == Sink,
    RightHypTerm = WHypTerm ) ),
         % set up the unification problem
         % fish out the wave hole
exp_at( GoalTerm, HolePos, Hole ),
        % find the position of the sink and the blocked term
exp_at( GoalEquality, VarPos, WGoalVariable ),
exp_at( GoalEquality, TermPos, WGoalTerm ),
        % generate an equality between a new variable and the hole
clamaccess:equality( Equality, _ ),
once( utility:new_variable_name( e, Vars, NewName ) ),
exp_at( Equality, VarPos, NewName ),
exp_at( Equality, TermPos, Hole ),
clamaccess:type_expression( GoalEquality, Type ),
clamaccess:type_expression( Equality, Type ),
        % update the context for new variable
universal_variable_names( Hyps==>Goal, Names ),
replace_all( Sink, NewName, Names, NewNames ),
        % generate an appropriate application
applications( _, NewNames, Application ),
        % assemble equivalence in same way as induction hypothesis
exp_at( HypMatrix, EqualityPos, HypEquality ),
```

```
    exp_at( HypMatrix, HypOtherPos, HypOtherSide ),
    clamaccess:equivalence( Equivalence, _ ),
    exp_at( Equivalence, EqualityPos, Equality ),
    exp_at( Equivalence, HypOtherPos, Application ),
            % update induction hypothesis for new variable
    replace_all( Sink, NewName, HypMatrix, RHypMatrix ),
            % and finally unify...
    match2( Equivalence, RHypMatrix, NewNames ),
            % now assemble the existentially quantified conjunction
    clamaccess:existential_quantifier( NewSide, _ ),
    clamaccess:bound_variable( NewSide, BVar ),
    clamaccess:bound_formula( NewSide, NewSideBound ),
    clamaccess:variable_name( BVar, NewName ),
    clamaccess:variable_type( BVar, Type ),
    clamaccess:conjunction( NewSideBound, _ ),
            % right side is instantiated application
    clamaccess:right_formula( NewSideBound, Application ),
            % update goal equality
            % find position of wave hole relative to equality
    exp_at( WGoalTerm, HolePos, WHole ),
    exp_at( GoalEquality, HPos, Exp ),
    Exp == WHole,
            % replace old wave hole with new variable
    replace( HPos, NewName, GoalEquality, WNewTerm1 ),
    cleanup_formula( WNewTerm1, WNewTerm ),
            % and strip old wave fronts
    wave_fronts( NewTerm, _, WNewTerm ),
    clamaccess:left_formula( NewSideBound, NewTerm ),
            % replace the old equality
    replace( Pos, NewSide, GoalFertilizationSide, NewFertilizationSide ),
            % reassemble the matrix
    clamaccess:equivalence( TempGoalMatrix, _ ),
    exp_at( GoalMatrix, GoalOtherPos, GoalOtherSide ),
    exp_at( GoalMatrix, FertPos, GoalFertilizationSide ),
    exp_at( TempGoalMatrix, GoalOtherPos, GoalOtherSide ),
    exp_at( TempGoalMatrix, FertPos, NewFertilizationSide ),
            % Housekeeping: Normalize and pretty print
    cleanup_formula( TempGoalMatrix, CleanGoalMatrix ),
    matrix( QVars, CleanGoalMatrix, NewGoal ),
    format( 'Applying very weak fertilization ... New goal is:~n',
        [] ),
    print_new_conclusion( Hyps, NewGoal, NewHyps )
   ],

% sequents out
```

```
        [
        NewHyps ==> NewGoal
        ],

% tactic
        very_weak_fertilize( [Dir1, Dir2, EqPos, HolePos] )
).
```

C.2.5 Wave

```
% First for standard notation

submethod( wave( MatrixPos, [ Name, Dir ] ),

% sequent in
        Sequent,

% preconditions
        [
                % do the housekeeping
         cleanup_sequent( Sequent, Hyps ==> Goal ),
         universal_variable_names( Hyps==>Goal, UVars ),
                % find a subterm of the conclusion containing wave fronts
         matrix( QVars, Matrix, Goal ),
         wave_terms_at( Matrix, MatrixPos, WTerm ),
                % make sure at least one of them is a hard wave front
         hard_wave_fronts( WTerm, [HWFT|HWFTs] ),
                % find a wave rule
         wave_rule( Name, genw( Dir, [_-[]-_-Pos] ), Cond => L :=> R ),
                % make sure that at least one of the hard wave fronts of the
                % conclusion corresponds to a wave front of the wave rule
         wave_fronts( _, WFRs, L ),
         member( F-_/[_,_], [HWFT|HWFTs] ),
         member( F-_/[_,_], WFRs ),
                % undo AndrewS's nifty encoding omitting wave fronts
                % from the right hand side--simplified version
         wave_fronts( R, [[]-[Pos]/[hard, out]], TempR ),
                % match subexpression and wave rule
         match( WTerm, Cond, L :=> TempR, UVars, NewR, NewCond ),
                % check that precondition holds
         cleanup_formula( Matrix, TMatrix ),
         exp_at( TMatrix, MatrixPos, _, WSuper ),
         ( elementary( Hyps==>NewCond, _ )
         v
           ( ( wave_front_proper( WSuper, Super ),
                clamaccess:conjunction( Super, _ ),
                clamaccess:left_formula( Super, NewCond ) )
           v
             ( clamaccess:equality( WSuper in _, _ ),
                MatrixPos = [_,_,_|P],
                NewCond = [NC],
                exp_at( TMatrix, [1|P], NC ) )
           )
```

```
	)
	],

% postconditions
	[
		% execute the rewrite
	 replace( MatrixPos, NewR, Matrix, TempMatrix ),
		% do some more housekeeping
	 cleanup_formula( TempMatrix, CleanMatrix ),
	 matrix( QVars, CleanMatrix, NewGoal ),
	 format( 'Applying wave rule ~p ... New goal is:~n',
	     Name ),
	 print_new_conclusion( Hyps, NewGoal, NewHyps )
	],

% sequents out
	[
	 NewHyps ==> NewGoal
	],

% tactic
	wave( MatrixPos, [Name, Dir] )
       ).


% Now for Jason's notation

submethod( wave( MatrixPos, [ Name, Dir ] ),

% sequent in
	Sequent,

% preconditions
	[
		% do the housekeeping
	 cleanup_sequent( Sequent, Hyps ==> Goal ),
	 universal_variable_names( Hyps==>Goal, UVars ),
		% find a subterm of the conclusion containing wave fronts
	 matrix( QVars, Matrix, Goal ),
	 wave_terms_at( Matrix, MatrixPos, WTerm ),
		% make sure at least one of them is a hard wave front
	 hard_wave_fronts( WTerm, [HWFT|HWFTs] ),
		% find a wave rule in jason's notation
	 wavej( Name, L :=> R ),
	 matchj( WTerm, L :=> R, UVars, NewR )
```

```
        ],

% postconditions
        [
                % execute the rewrite
         replace( MatrixPos, NewR, Matrix, TempMatrix ),
                % do some more housekeeping
         cleanup_formula( TempMatrix, CleanMatrix ),
         matrix( QVars, CleanMatrix, NewGoal ),
         format( 'Applying wave rule ~p ... New goal is:~n',
             Name ),
         print_new_conclusion( Hyps, NewGoal, NewHyps )
        ],

% sequents out
        [
         NewHyps ==> NewGoal
        ],

% tactic
        wave( MatrixPos, [Name, Dir] )
       ).


% If wave-front is nested inside identical terms we can simply moved the
% wave annotation outwards. Borrowed from original Clam, fixed such
% that meta-variables do not become instantiated spuriously.

submethod(wave(Path,ident),

% sequent in
        H==>G,

% preconditions
        [
         matrix(Vars,Matrix,G),
         wave_fronts( BareMatrix, Waves, Matrix ),
         select( Pos-WaveVars/TypDir, Waves, RWaves ),
         exp_at(BareMatrix, Pos, WaveTerm ),
         replace_multiple(WaveVars,_,WaveTerm, WavePatt ),
         % For efficiency: first check that terms are right ignoring waves
         % then verify no waves get in the way.
         \+ \+ surrounding_term( Pos, BareMatrix, WavePatt, Path ),
         wave_fronts( BareMatrix, RWaves, LessOneMatrix ),
         surrounding_term( Pos, LessOneMatrix, WavePatt1, Path ),
```

```
         WavePatt1 == WavePatt
        ],

% postconditions
        [
         wave_fronts( LessOneMatrix, [Path-WaveVars/TypDir], OneOutMatrix ),
         matrix( Vars, OneOutMatrix, NewG ),
         format( 'Moving wave front up... New goal is:~n' ),
         print_new_conclusion( Hyps, NewG, NewHyps )
        ],

% sequents out
        [
         NewHyps==>NewG
        ],

% tactic
        wave(Path,ident)
       ).
```

Appendix D

Gentzen System $G_=$ for Languages with Equality

The object-level logic of *Periwinkle* is a many-sorted first-order logic with equality. The rules are taken from [Gallier 86]. Γ, Λ and Δ denote sets of formulae. A and B formulae.

$$\frac{\Gamma, A, B, \Delta \vdash \Lambda}{\Gamma, A \wedge B, \Delta \vdash \Lambda}(\wedge : \text{left}) \qquad \frac{\Gamma \vdash \Delta, A, \Lambda \quad \Gamma \vdash \Delta, B, \Lambda}{\Gamma \vdash \Delta, A \wedge B, \Lambda}(\wedge : \text{right})$$

$$\frac{\Gamma, A, \Delta \vdash \Lambda \quad \Gamma, B, \Delta \vdash \Lambda}{\Gamma, A \vee B, \Delta \vdash \Lambda}(\vee : \text{left}) \qquad \frac{\Gamma \vdash \Delta, A, B, \Lambda}{\Gamma \vdash \Delta, A \vee B, \Lambda}(\vee : \text{right})$$

$$\frac{\Gamma, \Delta \vdash A, \Lambda \quad B, \Gamma, \Delta \vdash \Lambda}{\Gamma, A \rightarrow B, \Delta \vdash \Lambda}(\rightarrow : \text{left}) \qquad \frac{A, \Gamma \vdash B, \Delta, \Lambda}{\Gamma \vdash \Delta, A \rightarrow B, \Lambda}(\rightarrow : \text{right})$$

$$\frac{\Gamma, \Delta \vdash A, \Lambda}{\Gamma, \neg A, \Delta \vdash \Lambda}(\neg : \text{left}) \qquad \frac{A, \Gamma \vdash \Delta, \Lambda}{\Gamma \vdash \Delta, \neg A, \Lambda}(\neg : \text{right})$$

In the following quantifier rules, x is any variable of sort s and y is any variable of sort s free for x in A and not free in A, unless $y = x$. In the $(\forall : \text{right})$ and the $(\exists : \text{left})$ rules, the eigenvariable y does not occur free in the lower sequent.

$$\frac{\Gamma, A[t/x], \forall x{:}s.A, \Delta \vdash \Lambda}{\Gamma, \forall x{:}s.A, \Delta \vdash \Lambda}(\forall : \text{left}) \qquad \frac{\Gamma \vdash \Delta, A[y/x], \Lambda}{\Gamma \vdash \Delta, \forall x{:}s.A, \Lambda}(\forall : \text{right})$$

$$\frac{\Gamma, A[y/x].\Delta \vdash \Lambda}{\Gamma, \exists x{:}s.A.\Delta \vdash \Lambda}(\exists : \text{left}) \qquad \frac{\Gamma \vdash \Delta, A[t/x], \exists x{:}s.A, \Lambda}{\Gamma \vdash \Delta, \exists x{:}s.A, \Lambda}(\exists : \text{right})$$

The following rules hold for every sort s, term t, function f and predicate P.

$$\frac{\Gamma, t =_s t \vdash \Delta}{\Gamma \vdash \Delta}$$

$$\frac{\Gamma, s_1 =_{u_1} t_1 \wedge \ldots \wedge s_n =_{u_n} t_n \rightarrow f(s_1, \ldots, s_n) =_s f(t_1, \ldots, t_n) \vdash \Delta}{\Gamma \vdash \Delta}$$

$$\frac{\Gamma, s_1 =_{u_1} t_1 \wedge \ldots \wedge s_n =_{u_n} t_n \wedge P(s_1, \ldots, s_n) \rightarrow P(t_1, \ldots, t_n) \vdash \Delta}{\Gamma \vdash \Delta}$$

Appendix E

Higher-Order Pattern Unification Algorithm

The algorithm for higher-order pattern unification of [Nipkow 93] is presented as a set of transformation rules. The conventions are that s and t denote terms, F, G and H free variables, x, y and z bound variables, a and b atoms (bound variables or constants), and c constants. The rules are of the form

$$e \to \langle E', \theta' \rangle \,,$$

which are read as: The unification problem e is reduced to the list of unification problems E' under the new substitution θ'.

$$\begin{array}{rcl}
\lambda x.s \stackrel{?}{=} \lambda x.t & \to & \langle [s \stackrel{?}{=} t], \{\} \rangle \\
a(\overline{s_n}) \stackrel{?}{=} a(\overline{t_n}) & \to & \langle [s_1 \stackrel{?}{=} t_1, \ldots, s_n \stackrel{?}{=} t_n], \{\} \rangle \\
F(\overline{x_m}) \stackrel{?}{=} a(\overline{s_n}) & \to & \langle [H_1(\overline{x_m}) \stackrel{?}{=} s_1, \ldots, H_n(\overline{x_m}) \stackrel{?}{=} s_n], \{F \mapsto \lambda \overline{x_m}.a(\overline{H_n(\overline{x_m})})\} \rangle \\
 & & \text{where } F \notin \mathcal{FV}(\overline{s_n}) \text{ and } a \text{ is a constant or } a \in \{\overline{x_m}\} \\
F(\overline{x_n}) \stackrel{?}{=} F(\overline{y_n}) & \to & \langle [], \{F \mapsto \lambda \overline{x_n}.H(\overline{z_p})\} \rangle \\
 & & \text{where } \{\overline{z_p}\} = \{x_i \mid x_i = y_i\} \\
F(\overline{x_m}) \stackrel{?}{=} G(\overline{y_n}) & \to & \langle [], \{F \mapsto \lambda \overline{x_m}.H(\overline{z_p}), G \mapsto \lambda \overline{y_n}.H(\overline{z_p})\} \rangle \\
 & & \text{where } F \neq G \text{ and } \{\overline{z_p}\} = \{\overline{x_m}\} \cap \{\overline{y_n}\}
\end{array}$$

Bibliography

[Åhs & Wiggins 94] T. Åhs and G. A. Wiggins. Relational rippling for logic program synthesis and transformation, 1994. Submitted to Twelfth International Conference on Automated Deduction.

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

[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.

[Bouverot 91] A. Bouverot. Extracting and transforming logic programs. Technical report, LIENS-91-4, Ecole Normale Supérieure de Liens, March 1991.

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

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

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

[Bundy *et al* 89] A. Bundy, F. van Harmelen, J. Hesketh, A. Smaill, and A. Stevens. A rational reconstruction and extension of recursion analysis. In N.S. Sridharan, editor, *Proceedings of the Eleventh Inter-*

national 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.

[Burstall & Darlington 77] R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal*

of the Association for Computing Machinery, 24(1):44–67, 1977.

[Clark & Darlington 80] K.L. Clark and J. Darlington. Algorithm classification through synthesis. *The Computer Journal*, 23(1):61–65, 1980.

[Clark 78] K.L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.

[Clark 81] K.L. Clark. The synthesis and verification of logic programs. Research Report DOC 81/36, Department of Computing, Imperial College, London, September 1981.

[Constable *et al* 86] R.L. Constable, S.F. Allen, H.M. Bromley, *et al.* *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall, 1986.

[Coquand & Huet 88] Th. Coquand and G. Huet. The calculus of constructions. *Information and Computation*, 76:95–120, 1988.

[Cousot & Cousot 77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[Curry & Feys 58] H.B. Curry and R. Feys. *Combinatory Logic*, volume 1. North-Holland, 1958.

[Deville & Lau 93] Y. Deville and K.K. Lau. Logic program synthesis, 1993. Submitted to the Journal of Logic Programming.

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

[Dowek *et al* 91] G. Dowek, A. Felty, H. Herbelin, G. Huet, C. Paulin, and B. Werner. The Coq proof assistant user's guide, version 5.6. Technical Report 134, INRIA, 1991.

[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.

[Gallagher 93] J. K. Gallagher. *The Use of Proof Plans in Tactic Synthesis.* Unpublished PhD thesis, University of Edinburgh, 1993.

[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.

[Harper *et al* 86] R. Harper, D. MacQueen, and R. Milner. Standard ml. Report ECS-LFCS-86-2, Department of Computer Science, University of Edinburgh, March 1986.

[Harper *et al* 87] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. In *Proc. of the Second Symposium on Logic in Computer Science*, 1987.

[Hayes & Jones 89] I.J. Hayes and C.B. Jones. Specifications are not (necessarily) executable. *Software Engineering Journal*, pages 330–338, November 1989.

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

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

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

[Hogger 90] C.J. Hogger. *Essentials of Logic Programming.* Oxford University Press, 1990.

[Horn 88] C. Horn. The Nurprl proof development system. Working paper 214, Dept. of Artificial Intelligence, Edinburgh, 1988. The Edinburgh version of Nurprl has been renamed Oyster.

[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 94] D. Hutter. Synthesis of induction orderings for existence proofs. In *Proceedings of CADE*, 1994. Forthcoming.

[Ireland & Bundy 92] A. Ireland and A. Bundy. Using failure to guide inductive proof. Technical report, Dept. of Artificial Intelligence, Edinburgh, 1992. Available from Edinburgh as DAI Research Paper 613.

[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.

[Kanamori & Horiuchi 87] T. Kanamori and K. Horiuchi. Construction of logic programs based on generalized unfold/fold rules. In *Proceedings of the 4th International Conference on Logic Programming*, pages 744–768. MIT Press, 1987.

[Kowalski 74] R. Kowalski. Logic for problem solving. DCL TechReport 75, Dept. of Artificial Intelligence, Edinburgh, 1974.

[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 P. Szeredi, 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 88a] K. K. Lau and S. D. Prestwich. Synthesis of logic programs for recursive sorting algorithms. Technical Report UMCS-88-10-1, Dept. of Computer Science, University of Manchester, October 1988.

[Lau & Prestwich 88b] 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.

[Lever 91] J. M. Lever. Program equivalence, program development and integrity checking. In K-K. Lau and T. Clement, editors, *Proceedings of LOPSTR-91*.

Springer Verlag, 1991. Workshops in Computing Series.

[Liang 92] Ch. Liang. λ-prolog implementation of ripple-rewriting: Abstract. In *Proceedings of the 1992 λ-Prolog Workshop*, 1992.

[Lloyd & Shepherdson 87] J.W. Lloyd and J.C. Shepherdson. Partial evaluation in logic programming. Technical Report CS-87-09, Department of Computer Science, University of Bristol, 1987. Also in Journal of Logic Programming.

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

[Luo & Pollack 92] Z. Luo and R. Pollack. Lego proof development system: User's manual. Report ECS-LFCS-92-211, Department of Computer Science, University of Edinburgh, May 1992.

[Madden *et al* 93] P. Madden, J. Hesketh, I. Green, and A. Bundy. A general technique for automatically optimizing programs through the use of proof plans. Research Paper 608, Dept. of Artificial Intelligence, Edinburgh, 1993. Extended abstract to appear in Proceedings of LOPSTR-93.

[Maher 87] M.J. Maher. Equivalences of logic programs. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming.* Morgan Kaufmann, 1987.

[Manna & Waldinger 91] Z. Manna and R. Waldinger. Fundamentals of deductive program synthesis. In F.L. Bauer, editor, *Logic, Algebra, and Computation*, pages 41–107. Springer-Verlag, 1991. NATO ASI Series F: Computer and Systems Sciences Vol. 79.

[Manna & Waldinger 92] Z. Manna and R. Waldinger. Constructive logic considered obstructive. 1992. Forthcoming.

[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 & Nadathur 88] D. Miller and G. Nadathur. An overview of λProlog. In R. Bowen, K. & Kowalski, editor, *Proceedings of the Fifth International Logic Programming Conference/ Fifth Symposium on Logic Programming.* MIT Press, 1988.

[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. To appear in *Extensions of Logic Programming*, edited by P. Schröder-Heister, Lecture Notes in Artificial Intelligence, Springer-Verlag.

[Neugebauer 93] G. Neugebauer. The LOPS approach: A transformation point of view. In K-K. Lau and T. Clement, editors, *Proceedings of LOPSTR-92*. Springer Verlag, 1993. Workshops in Computing Series.

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

[Nipkow 93] T. Nipkow. Functional unification of higher-order patterns. In *Proceedings of the 8th IEEE Symposium* Logic in Computer Science, 1993.

[Partsch & Steinbrüggen 83] H. Partsch and R. Steinbrüggen. Program transformation systems. *ACM Computing Surveys*, 15(3):199–236, September 1983.

[Paulson 89] L. Paulson. The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5:363–397, 1989.

[Phillips 91] C. Phillips. Well-founded induction and program synthesis using proof plans. Research Paper 559, Dept. of Artificial Intelligence, Edinburgh, November 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.

[Richards 93] B. L. Richards. Mollusc user's guide version 1.1. DAI Technical paper 23, University of Edinburgh, September 1993.

[Robinson 65] J.A. Robinson. A machine oriented logic based on the resolution principle. *J Assoc. Comput. Mach.*, 12:23–41, 1965.

[Smith 90] D. R. Smith. KIDS: A semiautomatic program development system. *IEEE Transactions on Software Engineering*, 16(9):1024–1043, September 1990.

[Sterling & Shapiro 86] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, Cambridge, Ma., 1986.

[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.

[Talcott 93] C. Talcott. A theory of binding structures, and applications to rewriting. *Theoretical Computer Science*, 112:330–338, 1993.

[Tamaki & Sato 84] H. Tamaki and T. Sato. A transformation system for logic programs that preserves equivalence. Technical Report TR-018, ICOT, 1984.

[van Harmelen *et al* 93] F. van Harmelen, A. Ireland, A. Stevens, and S. Negrete. The CLAM proof planner, user manual and programmer manual *(version 2.1).* to appear as a dai technical paper, DAI, 1993.

[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.

[Wiggins 93] G. A. Wiggins. The *Whelk* proof development and logic program synthesis system, 1993. Compulog Deliverable; to be submitted to the Journal of Logic Programming.

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