The Dynamic Creation of Induction Rules Using Proof Planning

Jeremy Gow

Doctor of Philosophy

Centre for Intelligent Systems and their Applications

School of Informatics

University of Edinburgh

2004

Abstract

A key problem in automating proof by mathematical induction is choosing an induction rule suitable for a given conjecture. Since Boyer & Moore's NQTHM system the standard approach has been based on *recursion analysis*, which uses a combination of induction rules based on the relevant recursive function definitions. However, there are practical examples on which such techniques are known to fail.

Recent research has tried to improve automation by delaying the choice of inductive rule until later in the proof, but these techniques suffer from two serious problems. Firstly, a lack of search control: specifically, in controlling the application of 'speculative' proof steps that partially commit to a choice of induction rule. Secondly, a lack of generality: they place significant restrictions on the form of induction rule that can be chosen.

In this thesis we describe a new delayed commitment strategy for inductive proof that addresses these problems. The strategy dynamically creates an appropriate induction rule by proving schematic proof goals, where unknown rule structure is represented by meta-variables which become instantiated during the proof. This is accompanied by a proof that the generated rule is valid. The strategy achieves improved control over speculative proof steps via a novel *speculation critic*. It also generates a wider range of useful induction rules than other delayed commitment techniques, partly because it removes unnecessary restrictions on the individual proof cases, and partly because of a new technique for generating the rule's overall case structure.

The basic version of the strategy has been implemented using the λ*Clam* proof planner. The system was extended with a novel proof critics architecture for this purpose. An evaluation shows the strategy is a useful and practical technique, and demonstrates its advantages.

Acknowledgements

Thank you to my supervisors: to Alan Bundy for lending me a huge amount of his wisdom and support, to Ian Green for helping me get started, and to Jacques Fleuriot for helping me stop.

Thanks to the members of the Edinburgh DReaM group for creating such an engaging and friendly research environment. And to Ben Curry, for his advice and constant distraction.

I am grateful to Christoph Walther and the members of the Programming Methodology Group in Darmstadt, who helped me to develop this work during my visit there.

This thesis would not have been completed without the support of my colleagues at UCL, especially Paul Cairns, whose encouragement was invaluable.

Finally, thank you to Allison Mackenzie for her patience and support.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Jeremy Gow*)

Publications

Part of Chapter 6 of this thesis has previously appeared in the Proceedings of the Sixth International Conference on Logic for Programming and Automated Reasoning [Gow et al., 1999].

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The last thing one knows in constructing a work is what to put first.

— BLAISE PASCAL, PENSÉES

Mathematical induction is a technique used extensively in theorem proving systems for proving properties about objects that involve repetition. Because of the ubiquity of iteration and recursion in computer systems, it is especially useful when applied to software and hardware verification.

Given the arduous nature of formal proof development, automated theorem proving has been the subject of research since the inception of Artificial Intelligence in the 1950s. Progress has been especially difficult in automating inductive proof, because of the particular search problems introduced by induction [Boyer and Moore, 1992]. Specifically, finding the appropriate induction rules, lemmas and generalisations for a given problem [Bundy, 2001].

This thesis describes a novel approach to automating the first of these tasks: induction rule creation. It is also concerned with showing how proof planning [Bundy, 1988] can be used to effectively realise these ideas.

1.1 Motivation

Attempts to automate induction rule creation were, like much work in inductive theorem proving, dominated by the seminal work of Robert S. Boyer and J. Strother Moore [Boyer and Moore, 1979] for many years after its publication. The Boyer-Moore Theorem Prover introduced the what has been called *the* induction heuristic: to prove a property of a recursive function, try using an induction rule that has the same recursive structure as that function.

Boyer & Moore's approach is surprisingly powerful, and their theorem prover is still in use over 20 years later, along with systems based on the same essential ideas [Stevens, 1990]. However the late-1980s and 1990s saw a resurgence of work in induction theorem proving. Many recent developments have been based on the idea of rippling [Bundy et al., 1993], a heuristic for guiding proofs of the step case subgoals generated by applying an induction rule. Rippling has brought more sophisticated heuristic control to inductive theorem proving, and been used to get a purchase on some hard search problems, including the choice of induction rule.

Tied up with rippling has been the development of proof planning architectures for automated theorem proving [Bundy, 1988]. Proof planners are ideally suited to realise sophisticated heuristic strategies, and inductive proof via rippling has been a long running test-case, most notably with work on proof critics [Ireland and Bundy, 1996].

The individual and collective success of the rippling and proof planning paradigms has been an important motivation behind this project.

Recent developments in automating induction rule creation have tried to overcome two disadvantages with the Boyer-Moore approach. Firstly, choosing an induction rule at the beginning of the proof is often unreliable, as one cannot easily anticipate how a given choice will work out. [Kraan, 1994] begins a proof with a schematic goal, to see how the proof develops before selecting a known induction rule. Secondly, it is not always enough to use induction rules with the recursive structure taken from predefined recursive functions. [Protzen, 1995] takes a lazy generation approach to delaying the induction choice, dynamically creating a completely new rule with the information gleaned from the proof.

The main weaknesses of these ‘wait and see’ approaches to induction rule creation are:

Lack of heuristic control Although both employ forms of rippling to bring some measure of search control, they also both involve ‘speculative’ steps which can be applied freely and ad-infinitum, causing serious search problems.

Lack of generality The constraints placed on the form of induction rules that can be selected/created are overly restrictive, which limits the inductive problems that can be solved. Specifically:

> **Restrictions on rule style** In [Kraan, 1994] the rule must be constructor style, and in [Protzen, 1995] it must be destructor style — neither technique allows both, or a mixture of styles.
>
> **Restrictions on case structure** The cases of the rule are derived from the recursive functions used in the proof. Solutions with novel case structures cannot be found.

The main motivation for this thesis is to address these weaknesses.

1.2 Aims of the Thesis

Our aim was to design, implement and evaluate a strategy for inductive proof with the following properties:

- The choice of a step case is delayed until the middle of its proof, and this choice is used as a basis for constructing a new valid induction rule. This gives the strategy those advantages over Boyer & Moore's work that were demonstrated in previous research.

- The search is more tightly controlled than previous work on delayed-commitment induction rule creation. This is especially important in dealing with 'speculative' steps.

- The strategy has the ability to create a wider range of useful induction rules, i.e. rules that will allow more problems to be solved. This means lifting constraints on rule style and case structure.

- It is a practical and useful approach to automating inductive proof.

As well as providing a theorem proving strategy 'in the abstract', we aim to show that the proof planning approach provides an excellent architecture in which to implement the strategy. In particular, the techniques of middle-out reasoning and proof critics allow sophisticated search control techniques to be realised in a clean and understandable way.

1.3 An Example

To give the reader a better intuition for the kind of proof strategy we intend to automate, and for some the problems with previous approaches, we now give an illustrative example. Consider the following theorem, taken from [Paulson, 1991]:

$$\forall x{:}\tau.\forall l{:}list(\tau).foldleft\_tr(\circ, x, l) = x \circ foldleft\_tr(\circ, id, l)$$

The theorem holds given the following properties of $\circ$ and definition of *foldleft_tr*, a tail-recursive function that applies a two-argument function over the elements of a list (see Appendix B for a definition of *foldleft_tr* and all other functions that appear in this thesis):

$$\begin{aligned} X \circ (Y \circ Z) &= (X \circ Y) \circ Z \\ X \circ id &= X \\ foldleft\_tr(F, A, nil) &= A \\ foldleft\_tr(F, A, H :: T) &= foldleft\_tr(F, F(A, H), T) \end{aligned}$$

As we will see in Chapter 2, the standard techniques for induction selection (derived from the work of Boyer & Moore) would suggest using structural list induction on l, based on the recursive structure of *foldleft_tr*. However, it turns out that such a proof is unsuccessful, because the term substituted into the right-hand side l cannot be removed — there is no rewrite that move $H :: T$ out of this position.

In [Paulson, 1991] a lemma is introduced in order to prove the theorem:

$$foldleft\_tr(F, A, L <> (X :: nil)) = F(foldleft\_tr(F, A, L), X)$$

where $<>$ is the append function for list (again, see Appendix B for a definition of $<>$). The lemma motivates the invention of a new induction rule, which proves the theorem:

$$\frac{\begin{array}{rl} \vdash & \Phi(nil) \\ \Phi(x) \;\vdash & \Phi(x <> (y :: nil)) \end{array}}{\forall l : list(\tau).\Phi(l)}$$

Interestingly, the induction rule cannot be generated from the given function definitions, which means standard techniques for automation cannot prove the theorem. Instead, the rule is motivated entirely by the lemma.

More advanced induction selection techniques also have difficulty with this example. Middle-out induction [Kraan, 1994] can select the induction rule and complete the proof *if the rule is already known* — this is unlikely as it was created specifically for this proof. Lazy induction [Protzen, 1995] cannot solve the problem because the required induction rule falls outside the class of rules it is able to generate.

The induction strategy presented in this thesis is capable of generating this novel induction rule from the lemma, and proving that the rule is valid. Briefly, it does this by:

1. Generating a step case using an improved version of the middle-out reasoning techniques described in [Kraan, 1994], where the lemma suggest the step case of the induction.

2. Proving the step case is *wellfounded*.

3. Determining the form of the base case, which requires a novel case split to be generated.

4. Proving the base case.

1.4 Contributions

This thesis makes a number of original contributions to the understanding of automating induction rule creation:

- It provides a novel dynamic strategy fulfilling the aims set out in §1.2.

- We give experimental evidence for the effectiveness of the strategy in proving theorems that require novel induction rules.

- The significance of restrictions on induction rule style imposed in previous work is clarified.

We also contribute to automated theorem proving in general:

- We argue that when delaying choices during proof, a schema-based approach has search advantages over a lazy-generation approach.

- State-of-the-art techniques of proof planning are tested, and we make some original contributions to the design and use of proof critics.

- We describe a novel procedure for generating the missing cases of a case analysis. By expressing the problem as one of correcting a faulty conjecture, two previously separate areas of automated deduction are brought together.

- We present $\pi\sigma$-rewriting, an original technique for controlling search during non-confluent rewriting — applicable to e.g. rippling. A proof of the completeness of π-rewriting, a useful restriction of the technique, is given.

- The induction strategy provides a detailed case study of how creative steps in proof can be delayed and these decisions driven by subsequent proof.

1.5 Organisation of the Thesis

Looking from a distance, Chapters 1 to 3 lay out the groundwork to the main thesis, Chapters 4 to 8 set forth a body of novel heuristic techniques for effectively creating induction rules, and Chapters 9 to 15 discuss their implementation, evaluation and subsequent reflection on this work.

In greater detail, the thesis is structured as follows:

Foundations

- Chapter 1, **Introduction**

  The motivations, aims, contributions and structure of the thesis.

- Chapter 2, **Literature Survey**

  A survey of the relevant research literature: some background on inductive theorem proving, the main concepts of rippling and proof planning, and previous work on induction rule creation and validation.

- Chapter 3, **Induction Rule Structure**

  We reflect on what previous research tells us about how inductive proof is affected by the structure of induction rules and variations on the rippling heuristic.

The Induction Strategy

- Chapter 4, **Step Case Creation**

  Describes the strategy for obtaining a step case proof by delaying key choices until the middle of the proof.

- Chapter 5, **Synthesis of Case Structure**

  Here a strategy for the proof that the rule contains all the required cases is described. Failure of the proof can be exploited to generate the missing cases.

- Chapter 6, **Induction Rule Creation**

  We describe how a candidate step case is used as a basis for constructing and validating a new induction rule: an unsuccessful attempt to prove that the rule is valid is analysed in order to complete the rule.

Search Control

- Chapter 7, **Controlling Speculation**

  Speculative ripple steps make decisions about the form of the induction rule, but are non-terminating. We describe a proof critic that controls speculation by using it only when it will fix a failed ripple proof.

- Chapter 8, **Controlling Rewrite Search**

  Rewriting is at the heart of our strategy. This chapter reports how the rewriting search can be pruned by avoiding repetition of orthogonal rippling steps in different orders.

Implementation

- Chapter 9, **A Proof Planner with Critics**

  We describe the extension of the λ*Clam* proof planner with a novel critics mechanism, in order to implement our induction strategy. The planner uses *criticals* to combine critics into complex strategies.

- Chapter 10, **The *Dynamis* System**

  Describes the *Dynamis* system — the implementation of the induction strategy as a set of methods and critics in the λ*Clam* proof planner.

Evaluation & Reflection

- Chapter 11, **Experimental Evaluation**

  The implementation of the induction strategy is tested on a variety of inductive problems.

- Chapter 12, **Case Studies**

  We present some detailed case studies of proof attempts using the *Dynamis* system, and reflect on their success or failure.

- Chapter 13, **Related & Further Work**

  This chapter discusses the induction strategy with respect to Kraan's middle-out induction selection and Protzen's lazy generation of induction rules, along with directions for future research.

- Chapter 14, **Conclusions**

  We assess the contributions made by the thesis, and conclude whether our aims have been met.

Appendices

- Appendix A, **Glossary**

  An explanation of some techincal terms and notation.

- Appendix B, **Datatype & Function Definitions**

  Definitions for all the functions and datatypes used in this thesis.

- Appendix C, ***Dynamis* Documentation**

  Details of running the *Dynamis* system, and the lower level methods.

- Appendix D, ***Dynamis* Traces**

  Full traces from the evaluation, available in electronic form[1].

[1] From `http://homepages.inf.ed.ac.uk/s9362054/thesis`

Chapter 2

Literature Survey

2.1 Introduction

This chapter reviews a range of background material that is related to this thesis. The initial sections give an overview of mathematical induction (§2.2) and its use in mechanised theorem proving (§2.3). §2.4 describes proof planning, a central topic of this thesis.

The automation of inductive proof is surveyed in §2.5 onwards. Rippling, a technique for guiding step case proofs in described in §2.5. The standard approach to induction rule creation, recursion analysis, is examined in §2.6. The state-of-the-art in automating the choice of induction rule works by delaying the choice of rule into the middle of the proof (§2.7) and creating induction rules from this information (§2.8).

§2.9 looks at automating proofs of the construction of well-ordered relations satisfying a given set of constraints. This has mainly been dealt with in the literature in the context of proving program termination.

2.2 Proof by Induction

Mathematical induction can be roughly characterised as an argument which proves a proposition by appealing to some other instance of that proposition, and where it can be argued that this appeal process will eventually stop.

Such arguments have appeared throughout the history of mathematics: from the Pythagoreans of Ancient Greece [van der Waerden, 1961], and the 12th century Arabic mathematician al-Karaji [Rashed, 1994], to later European mathematicians, notably Pierre de Fermat, with his ‘method of infinite descent’ [Burton, 1988], and Blaise Pascal describing his *Triangle Arithmetique* [Pascal, 1665]. However, the first *explicit* formulation of an induction principle (along with the name) was given by Augustus DeMorgan in 1838 [Burton, 1988].

Today, induction is a common proof technique in many areas of mathematics. It very often appears as *Peano induction*, often called ‘the’ principle of mathematical induction, or the more general *complete induction*. These forms can be expressed as the following inference rules:

Peano Induction

$$\frac{\begin{array}{l}\Phi(0)\\ \forall k \in N.\ \Phi(k) \rightarrow \Phi(k+1)\end{array}}{\forall n \in N.\ \Phi(n)}$$

Complete Induction

$$\frac{\forall x \in N.\ (\forall y \in N.\ y < x \rightarrow \Phi(y)) \rightarrow \Phi(x)}{\forall x \in N.\ \Phi(x)}$$

2.2.1 Noetherian Induction

Peano and complete induction are inductions over N, the set of natural numbers. Although less common, induction over other sets appears in the mathematical literature, e.g. induction over the ordinals. Indeed, inductive arguments may be made over any set.

This generality is captured by *Noetherian induction*, also known as *well-founded induction*, a generalisation of complete induction to a set A and relation $\prec$. The relation $\prec$ must be *well-founded* over the set A, defined as there being no infinite descending chains $x_1 \succ x_2 \succ x_3 \succ \ldots$ such that $x_i \in A$ for all i. It can be expressed as the following inference rule:

Noetherian Induction

$$\frac{\forall x \in A.\ (\forall y \in A.\ y \prec x \rightarrow \Phi(y)) \rightarrow \Phi(x)}{\forall x \in A.\ \Phi(x)} \quad \prec \text{ w.f. over } A$$

All induction principles can be derived from this general scheme. For example, to derive structural induction over the natural numbers:

1. Let $A = nat$ and $\prec = <$, this discharges the side condition.

2. Perform a case split on the premise: $x = 0$ or $x = s(u)$.

3. In the $x = s(u)$ case, let $y = u$.

4. Simplify w.r.t. the definition of $<$.

2.3 Inductive Theorem Proving

In this section we briefly survey the use of inductive proof in theorem provers — computer systems that assist with or perform logical proof.

2.3.1 Reasoning about Computer Systems

Inductive reasoning is well-suited to proving properties of objects that contain repetition, and so has found many applications in proving properties about both software and hardware systems. In fact, the use of induction was proposed in one of the earliest papers on this field [McCarthy, 1963]. This work has developed into the *verification by proof* paradigm: the computer system is modelled as a set of mathematical definitions, along with a specification of the expected behaviour as a set of theorems. Proving these theorems verifies the correctness of the system relative to the specification. Note that there may still be a 'gap' between the specification-as-theorem and the system requirements.

Verification proofs can be carried out without machine support, which suffices for small systems e.g. [Burstall, 1969, Paulson, 1991]. However, this technique becomes impractical for anything but toy systems, as one cannot be sure that the proof is correct any more than the original system — although a failed hand proof can still reveal errors. Formalisation of the specification and proof in a particular logic can increase confidence in the correctness of individual steps, but mistakes are still possible, and there is an additional problem of a huge increase in the proof size [Nederpelt et al., 1994].

For these reasons, formal verification proofs of computer systems are often carried out with the aid of a computer. *Proof checking* programs allow a human user to reliably develop formal proofs, whilst also providing computer support for the huge 'book keeping' tasks that such proofs require. Induction is often a core proof technique in theorem proving systems, for example HOL [Gordon and Melham, 1993], ACL2 [Kaufmann and Moore, 1996], PVS [Owre et al., 1996], ISABELLE [Paulson, 1989], COQ [Huet et al., 1997] or NUPRL [Allen et al., 2000].

2.3.2 Formalised Mathematics

Although research into theorem proving, especially inductive theorem proving, has focused heavily on computer system verification, these programs have been used to develop formal logical proofs in many domains. Most notably, theorem proving systems such as Automath [Nederpelt et al., 1994] and Mizar [Trybulec and Blair, 1985] have been used to formalise large areas of mathematics. Given the ubiquity of induction in mathematics, induction can play a large part in these proof developments. For example, Shankar's development of Gödel's Incompleteness Theorems in the NQTHM system [Shankar, 1994]. Shankar chose these results to demonstrate that 'serious' mathematical results are amenable to mechanisation.

For a more detailed survey of formal mathematics and theorem proving systems, see [Harrison, 1996].

2.3.3 Interaction and Automation

Many formal proof tools provide some computer support for the reasoning process itself, e.g. by incorporating decision procedures for common domains. This adds a new dimension to theorem proving systems, with programs ranging from completely user-driven to the totally automatic[1], with many varieties of theorem prover/user interaction between.

Automation has considerabe advantages in reducing the time and effort spent on formal proof development, which can be exceptionally long and tedious [Shankar, 1994]. Automated reasoning has long been a goal of Artificial Intelligence, in particular automating mathematical reasoning [Newell et al., 1956].

[1] Although systems that 'fully automate' proof search often require significant input from their users in the form of system configuration, e.g. Otter [McCune, 1990].

2.3.4 Explicit vs Implicit Induction

Attempts to automate inductive proof have fallen into two distinct camps:

Explicit Induction Proof using inductive inference rules, i.e. special cases of the Noetherian induction rule (see §2.2.1).

Implicit Induction Proving a statement by showing that if assumed true then it does not create an inconsistency, which is equivalent to performing an inductive proof [Comon, 2001]. Also known as *proof by consistency* or *inductionless induction*.

This thesis deals with the automation of explicit induction, and below we survey only work from this area. However, it is first worth noting a few aspects of the implicit induction approach.

Implicit induction techniques differ in how they check the consistency of the specification after the addition of the conjecture. For example, the technique presented in [Jouannaud and Kounalis, 1989] orients an equational specification into a convergent rewrite system and uses Knuth-Bendix completion [Knuth and Bendix, 1970] to check that no previously unequal constructor terms have been made equal.

Research in implicit induction has been a process of gradually lifting the constraints the technique places on the specification. Recently the area has been generalised and extended within a single framework [Comon and Nieuwenhuis, 2000], which requires the specification to be an *I-Axiomatisation*. These restrictions are a disadvantage when compared to explicit induction. Another disadvantage is the relative unintuitiveness of the technique, making it unsuited to interactive systems and difficult to design good heuristics for automation.

Amongst its advantages are the fact that modern versions of the technique are *refutationally complete* [Bachmair, 1991] — they are guaranteed to reject non-theorems

— and the ability to easily handle mutually recursive definitions. Both are areas where explicit induction work has traditionally been weak. Implicit induction has been implemented in systems such as UNICOM [Gramlich, 1990], SPIKE [Bouhoula et al., 1992], and RRL [Kapur and Zhang, 1995] (which is also capable of explicit induction).

The rest of this chapter looks at previous A.I. research on automating explicit induction, which is more relevant to our thesis.

2.3.5 Generalisation & Lemma Speculation

In his survey of automated induction [Bundy, 2001], Bundy identified the three key problems in automating inductive proof: constructing induction rules, introducing intermediate lemmas, and generalising conjectures. This section briefly discusses work on the second two problems, before looking at the first, which is the main subject of this thesis.

In general, intermediate lemmas and generalisations are required in inductive proofs because the *cut rule* is required for inductive theories [Kreisel, 1965]. The cut rule uses a ‘cut formula’ A to prove another formula Δ:

$$\frac{\Gamma, A \vdash \Delta \qquad \Gamma \vdash A}{\Gamma \vdash \Delta}$$

As A could be *any* formula, this rule poses a considerable challenge for automating backwards proof, and special search heuristics are required to find suitable cut formulae. The cut formula A may be a generalisation or a lemma — the distinction is vague, and is based on whether showing that Δ follows from Γ, A is trivial, in that Δ is somehow a simpler form of A [Hesketh, 1991]. However, the distinction is reflected in the separate collections of heuristics that have been developed for each case.

Techniques for finding a cut formula which generalises the current goal date back to [Aubin, 1976], and can be grouped into three main types, all of which have been shown to assist in automatic proof:

Generalising Apart This replaces a single universal variable with two or more new ones. For example, $\forall x.\phi(x,x)$ is generalised to $\forall x.\forall y.\phi(x,y)$. One way to do this uses *primary recursion paths* — variables nested only within recursive argument positions — to find candidates for separation [Aubin, 1976].

Generalising Subterms One or more compound subterms are replaced by a fresh universal variable. This can be done by selecting suitable identical subterms on either side of an equality or implication [Boyer and Moore, 1979].

Generalising Accumulators Here a constant is replaced with a variable in the accumulator argument of a function. This often requires additional term structure to be added elsewhere in the formula, to retain its validity. Early work attempted to guess the extra term structure through trial and error [Aubin, 1976].

In [Castaing, 1985] the mismatch between induction hypothesis and conclusion is used to perform the first two forms of generalisation. For a survey of work on generalisation up to 1990 see [Hummel, 1990].

These three forms of generalisation have been unified within a single framework that delays the choice of generalisation using meta-variables [Hesketh, 1991], using proof planning (see §2.4 below). Building upon this work, Ireland used failed proof attempts to better focus the use of meta-variables when constructing generalisations of accumulator [Ireland and Bundy, 1996], and a similar approach has also been used for the other forms of generalisation [Maclean, 1999]. For more detail see our discussions of proof critics in §2.4.4 and §2.5.5 below.

The discovery of intermediate lemmas has been less well-studied. Ireland's work on proof critics addressed lemma speculation as well as generalisation, and used a failed proof attempt to build a schematic lemma that would assist with the proof [Ireland, 1992, Ireland and Bundy, 1996]. Walsh used patterns of divergence in inductive proofs in the SPIKE system to speculate lemmas that would allow the proof to proceed [Walsh, 1996]. Constraint-based approaches to lemma speculation and accumulator generalisation have been developed in RRL [Kapur and Subramaniam, 1996, Kapur and Sakhanenko, 2003].

2.4 Proof Planning

Proof planning is an approach to automated theorem proving originally designed to reduce proof search by raising it to a meta-level [Bundy, 1988, Bundy et al., 1991]. Classical theorem proving explores step-by-step a search space of inference rules applied 'backwards' to a goal formula. In proof planning the search is conducted with *methods*, A.I.-style planning operators which describe common patterns of reasoning in the object logic via meta-logical pre- and post-conditions. Methods can represent proof steps larger than a single inference. They are applied to *meta-level goals*, which are meta-logical representations of (possibly multiple) goals in the object logic.

Proof planning systems use methods to build an abstract proof tree, or *proof plan*, which can then be used to find an object level proof, e.g. by running tactics corresponding to methods (see §2.4.1). There need not be a guarantee that any corresponding object level proofs can be found or even exist, although most proof planning literature assumes that there is.

Meta-level goals and the meta-logical formulae in method conditions can express both legal and heuristic statements about proof goals. Legal statements are about the form of the object goals, e.g. when a method *could* be applied. Heuristic statements help guide the proof search, e.g. saying when a method *should* be applied. Methods and meta-level goals are usually designed by system authors or users, and typically oriented towards a specific domain where a set of heuristics is known, e.g. summing series [Free, 1992]. In [Bundy, 1991] a methodology for good method design is described, proposing evaluation criteria such as generality and parsimony. There has also been some recent work on automatically learning method sets from examples [Jamnik et al., 2002].

The intended advantage of proof planning is that the planning search space is significantly smaller than the original object level search space. Conversely, the plan space is likely to be incomplete. Both these things depend entirely on the particular method set.

Another aim of proof planning is to provide declarative, as opposed to procedural, specifications of methods which can be reasoned *about* mechanically, not just executed. This facilitates the automatic learning [Jamnik et al., 2002] and adaptation [Huang et al., 1995] of proof methods.

2.4.1 *Clam*: Advance Planning

The first proof planning system was *Clam* [Bundy et al., 1991, van Harmelen, 1996]. It built upon the *tactic* based approach to theorem proving, e.g. the HOL system [Gordon and Melham, 1993], where common patterns of inference rules are captured in tactics, a small program which automates the search for a proof fragment by applying rules according to the given pattern. In *Clam*, a method is considered to be a specification for a tactic, providing conditions for its application and the effects it has on the goal. A given tactic may have multiple methods, corresponding to its use in different situations.

The *Clam* system was designed to work in conjunction with a tactic-based theorem prover, specifically the *Oyster* system, an implementation of Martin-Löf's Type Theory. It constructs a proof plan which is used to guide *Oyster* to a proof, by replacing methods with their corresponding tactics [Bundy et al., 1990b]. Hence planning is done in advance of proving.

The default method set in *Clam* is designed for inductive proof, and is described in detail in §2.5. *Clam* has also successfully been combined with HOL instead of *Oyster*, with minimal adjustment to the default inductive method set [Boulton et al., 1998]. Given that HOL's logic is classical higher-order logic rather than Martin-Löf Type Theory, this illustrates the generality of proof methods.

Clam method conditions are written in Prolog, a logic programming language. This allows both the specification in a declarative style, i.e. as meta-logical statements, and their evaluation as Prolog programs. However, in practice it is possible to write procedural style conditions in Prolog, and *Clam* method designers often do this to e.g. improve their efficiency or implement complex strategies.

2.4.2 λ*Clam*: Methodicals and Higher Order Meta-Logic

λ*Clam* [Richardson et al., 1998, Dennis and Brotherston, 2002] is the successor to the *Clam* system. Like *Clam*, λ*Clam* is an planning system, producing plans to be converted into tactics. Unlike *Clam*, which has *Oyster*, there is no specific underlying tactic-based theorem prover. There are plans to make λ*Clam* more 'logic independent', enabling the same proof plans to be used over a variety of logics[2].

Method conditions are now written in λ-Prolog [Nadathur and Miller, 1998] a higher-

[2] Lucas Dixon, personal communication.

order version of Prolog. Having a higher-order meta-logic has allowed a much more concise, natural and declarative expression of methods.

Another significant aspect of λ*Clam* is the use of *methodicals* to 'join together' methods to specify larger ones, in much the same way that tactics are formed using tacticals. This is extremely useful when describing large and complex strategies — a common problem in *Clam*. It also allows a more declarative specification of such strategies. A semantics for these method expressions, based on continuations, is given in [Richardson and Smaill, 2001].

2.4.3 ΩMEGA: Hierarchical Proof Planning

The ΩMEGA system [Benzmüller et al., 1997], [Kerber, 1998] is another proof planning implementation, but differs from the *Clam* family in a number of important aspects. Most importantly, it does not differentiate between methods, tactics and inference rules: everything is a method. When a method is applied, further planning is carried out to construct a proof that an object level proof exists. This process bottoms out with the application of methods corresponding to inference rules. Hence the proof plan is a hierarchy, both in the normal 'proof tree' sense, and in that some methods can be expanded to another proof plan. The architecture allows planning and proving to be interleaved, rather than planning being done in advance. This lets ΩMEGA recover after forming faulty plans which have no corresponding proof.

Another important difference from *Clam* is the system's division of preconditions into declarative and procedural aspects, as well method slots for posting constraints, and the use of constraint reasoning [Melis et al., 2000].

2.4.4 Proof Critics

Failed proof attempts can often provide useful information about the form a successful proof might take. Proof critics are an extension to the proof planning architecture that embody this idea, and were first proposed in [Ireland, 1992]. Critics analyse failed planning attempts and perform patches to the proof plan which might lead to success. Just as methods describe the common structure of proofs, critics describe exceptions to this structure and how they can be handled.

Ireland developed a set of four *wave critics* [Ireland and Bundy, 1996] which respond to the failure of the *wave* method, from the induction method set in *Clam*. These have been implemented in the *Clam* v3 system. We will look at the wave critics in more detail in §2.5.5.

Critics have also been used to suggest generalisations [Maclean, 1999] and fix divergent proof attempts [Walsh, 1996] in inductive proofs, and to guide co-induction proofs [Dennis et al., 2000]. Their use in improving user-interaction in inductive theorem proving is described in [Ireland et al., 1999] and [Jackson, 1999].

In the remaining sections of this chapter, we look at various techniques used in the automation of inductive proof.

2.5 Rippling

Rippling is a heuristic technique designed to guide rewriting of step case goals during inductive proof [Hutter, 1990, Bundy et al., 1993]. It exploits the common structure in these goals: that both the inductive hypotheses and conclusion are derived from the original goal, and so have a common syntactic structure.

Rippling makes two assumptions. Firstly, that the hypothesis and conclusion differ because of some additional term structure, and that the goal can be solved by removing these differences. Although this is not necessarily true[3], it is a feature of most inductive theorem proving (ITP) systems, which makes rippling widely applicable. The second assumption is that the common syntactic structure is maintained throughout the proof of the subgoal. Aubin was the first to remark that this holds true in many step case proofs [Aubin, 1976].

The key idea is to restrict the manipulation of the conclusion and/or hypotheses so that the proof fits these assumptions. Its aim is to remove differences between hypothesis and conclusion, allowing the hypothesis to be used to prove the conclusion, known as *fertilisation*, and hence prove the step case. A rewrite step is only allowed if it meets the following criteria:

Skeleton Preservation The common syntactic structure between hypothesis and conclusion, known as the *skeleton*, is preserved.

Difference Removal The step helps 'remove the differences', in that unwanted term structure is either

1. Moved towards the top of the term, leaving a subterm which is 'less different' from the skeleton, or
2. Moved towards a position in the conclusion which corresponds to a universal variable in the hypothesis, where it won't prevent fertilisation, or
3. Removed completely.

Because it restricts the rewriting like this, rippling is a heuristic strategy. Most work on rippling has considered a straightforward rewriting environment, although it has

[3] For example, consider the step case $x < y, \Phi(x) \vdash \Phi(y)$.

been successfully integrated with other ATP techniques, e.g. matrix theorem proving [Pientka and Kreitz, 1999].

A number of formalisms have been developed for rippling, namely wave annotation, C-Equations and embeddings. These are described in the next three sections, along with the details of the rippling heuristic, and some variations on it.

2.5.1 Wave Annotation

The wave annotation approach to rippling [Bundy et al., 1993, Basin and Walsh, 1996] introduces new functions (or *wave annotations*) into a term to indicate the differences between it and another target term. The special unary function wf is introduced above term structure that does not appear in the target, and another function wh is introduced above the term structure that does. For example, the difference between the hypothesis and conclusion of the step case (2.1) is indicated by the annotation shown in (2.2). (The uppercase variables indicate meta-variables that have been substituted for universally quantified variables in the hypothesis, a standard technique in ITP.)

$$x+Y=Y+x \quad \vdash \quad s(x)+y=y+s(x) \tag{2.1}$$

$$x+Y=Y+x \quad \vdash \quad \mathsf{wf}(s(\mathsf{wh}(x)))+y=y+\mathsf{wf}(s(\mathsf{wh}(x))) \tag{2.2}$$

The term structure that falls inside a wf and outside a wh is known as a *wave front*, whereas the contents of the function wh is known as a *wave hole*. In (2.2) there are two wave fronts $s(\ldots)$, and two wave holes with contents x.

Another special function snk is used to indicate a *sink* — a position which corresponds to a variable in the target which can be instantiated with any term, i.e. a universal object or meta-variable. (2.3) shows an example of sink annotation, where capital letters denote meta-variables.

$$x + Y = Y + x \quad \vdash \quad \mathsf{wf}(s(\mathsf{wh}(x))) + \mathsf{snk}(y) = \mathsf{snk}(y) + \mathsf{wf}(s(\mathsf{wh}(x))) \qquad (2.3)$$

To make annotated terms more readable, the 'box-and-hole' notation is usually used[4]: wave fronts are enclosed by boxes, wave holes are underlined and sinks are marked with $\lfloor \ldots \rfloor$, e.g. (2.4) depicts (2.3) in box-and-hole notation.

$$x + Y = Y + x \quad \vdash \quad \boxed{s(\underline{x})} + \lfloor y \rfloor = \lfloor y \rfloor + \boxed{s(\underline{x})} \qquad (2.4)$$

Terms can be annotated automatically by difference unification [Basin and Walsh, 1993]. Certain constraints on the placing of annotations ensure terms are *well-annotated*, for instance $\mathsf{wf}(\mathsf{wf}(x))$ is disallowed. Nested wave fronts and multiple wave holes in a wave front are permitted.

The skeleton of a term can be computed by replacing terms wave fronts with the contents of their wave hole, and sinks with the corresponding meta-variable[5]. A term can be annotated with respect to several targets simultaneously by having multiple wave holes in a wave front — this gives a set of skeletons, each resulting from a different choice of wave holes. This is useful when guiding step cases with more than one induction hypothesis.

The criteria of skeleton preservation (see §2.5) is enforced by allowing only rewrite rules which can be annotated so that the skeletons of the left and right sides are identical (or in the case of multiple wave holes, that the right side's skeletons are a nonempty subset of the left's). An annotated rewrite rule is called a *wave rule* — see Figure 2.5.1. Rippling is carried out by rewriting with wave rules, ensuring that the annotations match, modulo equivalent wave annotations. I.e. during search we must normalise the

[4] In fact this notation predates the wf/wh formalism.

[5] Alan Bundy, unpublished research note.

$$\boxed{s(\underline{X})}^{\uparrow} + Y \quad \Rightarrow \quad \boxed{s(\underline{X+Y})}^{\uparrow} \tag{2.5}$$

$$X + \boxed{s(\underline{Y})}^{\uparrow} \quad \Rightarrow \quad \boxed{s(\underline{X+Y})}^{\uparrow} \tag{2.6}$$

$$\boxed{s(\underline{X})}^{\uparrow} = \boxed{s(\underline{Y})}^{\uparrow} \quad \Rightarrow \quad X = Y \tag{2.7}$$

Figure 2.1: Three rewrite rules annotated as measure-decreasing wave rules. Only one of the many possible annotations is shown for each rule. The upwards arrows denote outwards wave fronts, which are moved to the top of the term during rippling. (2.5) and (2.6) preserve skeleton $X+Y$, whereas (2.7) preserves $X=Y$.

annotation with respect to $\lambda x.\mathsf{wh}(\mathsf{wf}(x)) = \lambda x.x$. An example of a rippling proof using wave annotation is shown in Figure 2.5.1.

Difference removal is achieved by marking wave fronts as travelling *outwards* towards the top of the term or *inwards* towards a sink. All wave fronts initially travel outwards, but can be redirected inwards, but not vice-versa. A *wave measure* captures the informal notion of progress described above, taking account of the number and depth of outwards and inwards wave fronts [Basin and Walsh, 1996]. Wave rules are only permitted if they decrease this measure, and hence the rippling strategy is terminating. Various wave measures have been proposed in an attempt to better model this process.

Note that a rewrite rule can produce several wave rules, although only a few will be applicable at any one time. The process can be fully automated. This give rise to a useful solution to the problem with traditional rewriting techniques that need to orient equations in one particular direction to ensure termination: often an equation needs to be used in both directions. For example, associativity axioms. With rippling each use can correspond to the application of a different wave rule, both derived from the same

$$\begin{array}{lcl}
x+Y=Y+x & \vdash & \boxed{s(\underline{x})}^{\uparrow}+\lfloor y\rfloor=\lfloor y\rfloor+\boxed{s(\underline{x})}^{\uparrow} \\
x+Y=Y+x & \vdash & \boxed{s(\underline{x+\lfloor y\rfloor})}^{\uparrow}=\lfloor y\rfloor+\boxed{s(\underline{x})}^{\uparrow} \\
x+Y=Y+x & \vdash & \boxed{s(\underline{x+\lfloor y\rfloor})}^{\uparrow}=\boxed{s(\underline{\lfloor y\rfloor+x})}^{\uparrow} \\
x+Y=Y+x & \vdash & x+\lfloor y\rfloor=y+\lfloor x\rfloor \\
 & true &
\end{array}$$

Figure 2.2: A ripple proof of the step case from the proof of the commutativity of $+$. The proof uses wave rules (2.5), (2.6) and (2.7) (see Figure 2.5.1) in that order. The outwards wave fronts are moved outwards and are eventually removed, allowing fertilisation in the final step. The skeleton $x+Y=Y+x$ is preserved throughout.

equation.

The rippling heuristic is implemented via wave annotation in the *Clam* proof planner (see §2.4.1), where the meta-level representation of formulae allows annotating functions to be added without changing the underlying logic [Bundy et al., 1991]. It has been successfully used as a basis for a inductive method set [Bundy et al., 1991]. [Bundy and Green, 1996] is an experimental comparison of the relative performance of rippling and standard rewriting in *Clam*. For a more extensive explanation of rippling see [Bundy et al., 1993]. A more formal account is given in [Basin and Walsh, 1996].

2.5.2 The $\mathcal{C}$-Calculus

The $\mathcal{C}$-calculus is another rippling formalism which is presented in [Hutter, 1990] and [Hutter, 1997]. Its formulation of rippling is similar to that of wave annotation. The fundamental difference is that annotations are represented by *colouring* individual symbols in a term, rather than by introducing special functions.

For instance the step case (2.4) can be represented in the $\mathcal{C}$-calculus as (2.8), with symbols in the skeleton coloured *sk* and those in the *context*, i.e. within wave fronts, coloured *cx*.

$$x+Y=Y+x \;\vdash\; s^{\mathsf{cx}}(x^{\mathsf{sk}})+^{\mathsf{sk}} y^{\mathsf{sk}} = y^{\mathsf{sk}} +^{\mathsf{sk}} s^{\mathsf{cx}}(x^{\mathsf{sk}}) \tag{2.8}$$

Wave rules are represented using *colour variables* that can take any colour value, or a value restricted by a *colour sort hierarchy*. For example, the wave rule (2.5) is written as the coloured rewrite rule (2.9), where the Greek letters are colour variables.

$$s^{\mathsf{cx}}(X^{\alpha})+^{\mathsf{sk}} Y^{\beta} \;\Rightarrow\; s^{\mathsf{cx}}(X^{\alpha}+^{\mathsf{sk}} Y^{\beta}) \tag{2.9}$$

[Hutter, 1997] defines a unification procedure for coloured terms, allowing coloured rewriting to be defined, which is used to implement rippling. Termination is achieved by orienting the wave rules into a terminating rewrite system on a case-by-case basis, rather than using a universal wave measure, although [Protzen, 1995] gives an account of how the wave measure approach of [Basin and Walsh, 1996] can be formalised in the $\mathcal{C}$-calculus.

The $\mathcal{C}$-calculus has a slightly wider coverage of skeleton preserving proofs, as there exist skeleton preserving rewrite proofs that it can capture that wave annotation cannot. Whether these are useful in practice is not known. A further advantage is the uniqueness of its representation, which avoids having to normalise the annotation during search. The coloured annotation approach has been generalised to other forms of search control [Hutter, 2000].

The calculus has been implemented in the INKA system, an inductive theorem prover [Hutter and Sengler, 1996].

2.5.3 Term Embeddings

A major drawback of the annotation approaches to rippling described above is transferring it to a higher-order setting. Under β-reduction the skeleton of a term can become broken, and wave or coloured annotations give rise to ill-annotated terms, or terms in which the skeleton is not preserved. To overcome this the $\mathcal{C}$-calculus has been extended to cope with higher-order syntax [Hutter and Kohlhase, 1997]. An alternative approach to higher-order rippling using *embeddings* is described in [Smaill and Green, 1996], and has been implemented in the λ*Clam* proof planner [Dennis and Brotherston, 2002].

A term embedding is a mapping from a term tree to another term tree, the target term. Those parts of the target term not in the range of this mapping correspond to wave fronts, and so embeddings can be used to formalise rippling. Because of the higher order setting, quantification, λ-abstraction and functions may all be optionally mapped by the embedding.

The embeddings are represented in λ*Clam* by a tree labelled with term positions: the embedding e embeds term t_1 into term t_2, written $e : t_1 \overset{\subseteq}{\rightarrow} t_2$, iff for a position p in e with label q the symbol at p in t_1 and q in t_2 are identical. Wave fronts are implicit in this representation: they are the term addresses that do not appear on the embedding tree. However, key features of wave annotation (see §2.5.1) can be still replicated in this representation: a wave front can be given a direction by marking the embedding tree node that maps to immediately 'beneath' it in the target term. As a result, embeddings can only represent blocks of wave fronts that all have the same direction. Sinks can also be represented by marking the appropriate leaf nodes of the embedding tree. A wave measure has been developed for embeddings that is similar to the one used for wave annotation [Dennis and Brotherston, 2002].

When applied to a step case proof, each induction hypothesis is embedded into the induction conclusion. A rewrite step is allowed if the hypotheses can be re-embedded into the conclusion, with the possibility of dropping an unembeddable hypothesis from the embedded set as long as this does not make the set empty.

The advantage of the embedding formalism is that the rippling annotation is separate from the term structure, so that β-reduction cannot produce ill-formed annotation (although the embedding will have to be recomputed), and the underlying logic does not need to be modified, e.g. unification does not need to account for wave annotation.

Furthermore, in the case of multiple rippling targets the embeddings approach does not suffer from the problem of 'mixed skeletons'. This problem arises in other rippling approach because the interdependancies between different wave holes are ignored, leading to bogus skeletons that can misguide the search. Yoshida proposed *coloured rippling* [Yoshida et al., 1994] to prevent skeleton mixing in the wave annotation approach.

In a naive embeddings implementation the separation of the annotation and the term means that the entire embedding does have to be recomputed with each step. However, more efficient implementations are possible[6].

2.5.4 Creational Rippling

Standard rippling can be used to guide *constructor style* step-cases, where induction terms, and hence wave fronts, only appear in the induction conclusion. In *destructor style step cases* induction terms are substituted into the hypotheses, therefore wave fronts also appear in the hypotheses. For example, (2.10) shows a destructor style step case from the proof of the commutativity of $+$, with an annotated induction hypothesis.

$$\boxed{p(\underline{x})} + (Y+Z) = (\boxed{p(\underline{x})} + Y) + Z \ \vdash \ x + (y+z) = (x+y)+z \qquad (2.10)$$

[6] Jonathan Whittle, personal communication.

Creational rippling is an extension to rippling that can be used to guide step case proofs with annotated hypotheses [Bundy et al., 1993, Hutter, 1997]. This is done by rippling the conclusion so that a wave front is created there which matches the wave front in the hypothesis. Such a step can lead to non-termination under wave measures, as the number of wave fronts is increased, so additional steps must be taken to ensure termination.

In [Hutter, 1997] creational rippling is called the 'blowing up of terms', using *context-creating* $\mathcal{C}$-equations to introduce the new wave fronts. In [Bundy et al., 1993] creational wave rules are defined, which contain *anti-wave fronts* that can match with wave fronts in the hypotheses — after a creational ripple the matching wave fronts are erased, leaving an 'expanded' skeleton. Although skeleton preservation is violated, the step is acceptable because both hypothesis and conclusion still have a common skeleton. In both formalisms the creational ripple can produce additional wave fronts in the conclusion that need to be rippled away — hence it is also known as *rippling across*, as the hypothesis wave front appears to have been moved across to the conclusion.

The literature on rippling does not describe creational rippling in the same depth as the standard technique, and it has not been formalised to the same extent. Essentially the same approach is outlined in both the wave annotation and $\mathcal{C}$-calculus formalisms: before standard rippling is applied a phase of creational rippling takes place, where a creational step is taken providing some wave front in the hypothesis is matched by the new wave fronts. In the case of multiple induction hypotheses, a creational ripple can introduce wave fronts which match wave fronts in only some of the hypotheses. In this case the *unviable* hypotheses are discarded from the rippling process, i.e. their corresponding wave holes are erased.

Creational rippling has been implemented experimentally in the λ*Clam* system

[Gow and Bundy, 2000]. Protzen criticises the technique for being ‘complex and unintuitive’ [Protzen, 1995] — probably due to its poor theoretical development and description compared to standard rippling.

2.5.5 The Wave Critics

Proof critics are used to describe common exceptions to proof planning methods, responding to a pattern of failed preconditions that suggests a particular amendment to the proof plan (see §2.4.4). Proof critics was first examined in the context of rippling, with the introduction of the *wave critics* [Ireland, 1992, Ireland and Bundy, 1996].

These four critics respond to the failure of the *wave* method, which implements a single ripple rewrite in the *Clam* proof planner. Each critic corresponds to a particular pattern of failure in the wave method’s preconditions. Those preconditions are:

Wave Front The goal contains a wave front.

Wave Rule A wave rule can ripple this wave front.

Condition Any condition on the wave rule can be discharged.

Sinkable Any inwards wave fronts are above for sinks or outwards wave fronts. Recall that wave fronts can be rippled inwards to a sink position where they match a universal variable in a hypothesis (see §2.5.1). Alternatively, they may meet an outward wave front and ‘cancel each other out’.

Table 2.1 shows how the failure of the wave preconditions triggers the various wave critics. Partial success in applying a wave rule means that adding some term structure (say, $s(s(x))$ instead of $s(x)$) would allow the rule to be applied.

The critics respond to rippling failure by generalising the original conjecture, introducing a case analysis into the step case proof, revising an induction term in the

	Generalisation	Case	Induction	Lemma
Precondition		Analysis	Revision	Discovery
Wave Front	Yes	Yes	Yes	Yes
Wave Rule	Yes	Yes	Partial	No
Condition	Yes	No		
Sinkable	No			

Table 2.1: Association between wave method failure and the wave critics. Yes, Partial and No indicate the precondition succeeds, partially succeeds (see main text) or fails respectively. Taken from [Ireland and Bundy, 1996].

induction rule, or by attempting to find a lemma that will allow rippling to continue. With the exception of the case analysis critic, the wave critics perform these tasks using middle-out reasoning, i.e. introducing a solution with one or more meta-variables that are appropriately instantiated later in the proof search.

[Ireland and Bundy, 1996] reports success in using the wave critics to find automatic proofs to many theorems previously unsolvable using rippling, and other inductive techniques. Further development of the generalisation critic is reported in [Ireland and Bundy, 1999].

2.6 Recursion Analysis

We now look at previous work on automating the selection of induction rules. The standard approach to rule selection is *recursion analysis*, based upon techniques developed by Boyer and Moore [Boyer and Moore, 1979]. This uses the *dual* inductions of terminating recursive functions that appear in the goal. The dual induction rule I_f of a recursive function f corresponds to a relation identical to the computation order of f. The termination of f guarantees the well-foundedness of this relation, and hence the validity of the rule.

To prove a particular conjecture a heuristic is used to suggest a set of induction rules: if the function f appears in the conjecture with recursive arguments $x_1,\ldots,x_n$ then use the dual induction rule I_f with induction variables $x_1,\ldots,x_n$ (providing x_i are all universal variables). This is known as the *duality heuristic*[7]. The set of *raw suggestions* given by the heuristic undergoes two more stages of processing. First the system attempts to i) disregard some rules as inherently inferior to others and ii) combine rules together, to form rules superior to their constituents. The notion of superiority of induction rules can vary between systems, as we shall see below.

Finally an induction is selected by considering the induction terms substituted into the conjecture by each rule and the effects this will have on the subsequent proof. An occurrence of an induction term in the conjecture is *flawed* if it prevents the symbolic evaluation of the surrounding term using the recursive definitions, otherwise it is *unflawed*. A rule is selected based on the number of flawed and unflawed induction terms it will produce (see [Stevens, 1988] for details).

2.6.1 Subsumption

The first system to incorporate recursion analysis was NQTHM, also known as the Boyer-Moore Theorem Prover [Boyer and Moore, 1979, Boyer and Moore, 1988]. The system considers an induction rule superior to another if it *subsumes* the other rule. Subsumption can be defined as:

- Rule A is subsumed by rule B iff there is a *repeated form* of A such that each step

[7] It is also known as *the* induction heuristic e.g. [Walther, 1992].

case of this rule is *directly subsumed* by a step case of rule B.

- The N-repeated form of a rule is constructed by applying the substitutions of each step case to each step case of the $(N-1)$-repeated form (see [Stevens, 1990] for details).

- Step case S_A is directly subsumed by step case S_B iff the conditions of S_B imply the conditions of S_A and each hypothesis/conclusion substitution of S_A is a subset of a hypothesis/conclusion substitution in S_B.

Informally, subsumption can be seen as considering rule B superior if it is an extension of N applications of rule A, for some N. NQTHM combines induction rules by *merging*. Merging two valid rules produces a third valid rule which subsumes the original two.

2.6.2 Subsumption Reconstructed

Although NQTHM was very successful at selecting appropriate induction rules and the system was well documented in [Boyer and Moore, 1979], this approach lacked any real theoretical foundation explaining *why* it worked. Because of this Stevens carried out a rational reconstruction of Boyer and Moore's recursion analysis [Stevens, 1988, Stevens, 1990]. He provided theoretical explanations of why these techniques often chose appropriate inductions, which lead him to identify and correct a number of flaws in the original process.

The reconstruction was based upon an informal meta-theory of inductive proofs — explanations about how and why inductive proofs succeed or fail. The key idea of this theory is that appropriate induction rules introduce induction terms that allow hypothesis and conclusion to be rewritten to match each other. These induction terms need to be *dealt with* – we need rewrite rules that involve these terms in the context they have been substituted into. If we use induction I_f dual to function f in the conjecture, then we can use the recursive definition of f to deal with some of the terms introduced by I_f. The danger is that a rule might introduce *side-effects* i.e. terms that cannot be dealt with using the recursive definitions available. Rule B is therefore superior if it subsumes rule A, as it will substitute ‘dealable’ terms into the same places as A, and will possibly allow A’s side-effects to be dealt with as well.

Among the advantages of Stevens’s recursion analysis is the use of the merging algorithm to perform the subsumption test – if rule A subsumes rule B then merging A and B simply returns A. His improved merging algorithm also allows, in some cases, repeated forms of rules to merged, finding a *common subsuming induction rule* for two rules.

2.6.3 Containment

Subsumption is not the only method of measuring the relative superiority of induction rules. Walther has proposed *containment* as an alternative method [Walther, 1993], and from this he developed an alternative set of techniques for improving, disregarding and combining destructor style induction rules[8] suggested during recursion analysis [Walther, 1993, Walther, 1994a]. These have been implemented in a version of the the INKA inductive theorem prover [Hutter and Sengler, 1996].

Containment is defined as: rule A is contained by rule B iff $<_A \subset <_B$, where $<_I$ is the well-founded relation corresponding to valid induction rule I. Hence the rule with the larger relation is considered superior, which is equivalent to preferring the rule with the logically stronger induction hypotheses. This can be seen as a meta-theory of inductive proofs that differs from, but does not necessarily oppose, Stevens’s theory.

[8] A destructor style induction rule only substitutes induction terms into induction hypotheses of step cases.

[Walther, 1993] describes a *containment test* which is sufficient to show containment between two induction rules: rule A is contained by B if for each hypothesis H_A in a step case S_A of A the following formula is true (taking all free variables to be universally quantified):

$$cond(S_A) \rightarrow \bigvee_{S_B \in \mathcal{S}(B)} \left(cond(S_B) \wedge \bigvee_{H_B \in \mathcal{H}(S_B)} \left[\bigwedge_{x \in dom(H_B)} H_B(x) = H_A(x) \right] \right)$$

where $\mathcal{S}(B)$ are the step cases of rule B, $\mathcal{H}(S)$ and $cond(S)$ are the hypotheses and conditions of step case S, $dom(H)$ is the domain of hypothesis H's substitution and $H(x)$ the effect of that substitution on variable x. The test is carried out by passing a set of these *containment formulae* to an inductive theorem prover.

If neither $<_A \subset <_B$ or $<_B \subset <_A$ can be shown using this test, then the rules are combined by taking the *separated union* of A and B [Walther, 1993]. This is an induction rule corresponding to the relation $<_A \cup <_B$ constructed so that the conditions of the step-cases are mutually exclusive. Although this union always exists, it is not guaranteed well-founded. The rule can be shown valid if it passes a *quasi-commutation test*. As with the containment test this involves discharging certain formulae using the inductive theorem prover, but they tend to be harder to prove. In the last resort the system can attempt a direct well-foundedness proof of the separated union.

[Walther, 1993] also defines *range* and *domain generalisations*: operations which modify an individual induction rule A to produce a rule A' that contains A. Both these operations correspond to procedures for extracting an induction rule from a terminating recursive function definition discussed in [Boyer and Moore, 1979] and [Stevens, 1990]. As with separated union, the generalised induction rule is not guaranteed well-founded.

Walther claims that his recursion analysis is superior to Boyer and Moore's approach (see [Walther, 1994a] for his comparison). His techniques are capable, in some cases, of constructing induction rules that require fewer supporting lemmata than Stevens's approach. However unlike Stevens, he does not address repeated forms and non-destructor style induction rules. The relative strengths of the two approaches has not yet been properly investigated, and it is unclear if either is superior, or perhaps if some combination of the two would be optimal.

2.6.4 Ripple Analysis

One of the major advantages of rippling is that it provides a strong normative model of how inductive proofs should proceed, and this model can be used to suggest solutions to other problems in automated induction. For example, when selecting an induction rule we can choose the rule that is most likely to allow subsequent rippling to succeed. *Ripple analysis* [Bundy et al., 1989] is an induction selection technique that takes this approach.

Induction terms appearing in the step-case will be annotated as wave fronts. Given a set of induction rules and a set of wave rules, ripple analysis suggests those rules that introduce wave fronts that can be rippled in the first step of the proof. This provides a set of raw suggestions which can be combined and disregarded using the techniques of recursion analysis. Indeed, ripple analysis can be seen as an extension of recursion analysis, as both consider the effect the induction will have on the first step of the step-case proofs. The former considers a rippling proof, the latter symbolic evaluation with the recursive definitions.

This comparison indicates the advantages this technique has over recursion analysis. Firstly, it may use any lemmata known to the system if they can be annotated as wave rules. They may suggest appropriate inductions different from any dual to functions in the conjecture. Secondly, the restrictions on rippling can disallow an in-appropriate induction even though recursion analysis suggests it [Bundy, 2001]. If an induction term may only be dealt with by rippling in using a recursive definition, but there is no sink position in which to put the wave front (see §2.5) then the proof is likely to fail. Recursion analysis would suggest this induction, ripple analysis would not, given that there is no applicable wave rule.

2.7 Delaying the Choice of Induction Rule

Recursion analysis and ripple analysis are the standard approaches to induction rule selection, but have two significant disadvantages [Bundy, 2001]. Firstly, they can only select an induction rule from a predetermined space, and the suitable choices may not be in that set [Protzen, 1995]. For recursion analysis this ‘dual space’ of induction rules is determined by the recursive functions in the conjecture and the operators for induction rule combination. Ripple analysis can select induction rules not in the conjecture’s dual space, but these must be supplied *a priori*, e.g. by the user.

Their second disadvantage is they must guess the effects of the induction choice using only the structure of the conjecture and a one-step lookahead for each induction term. This can obviously go wrong, as events later in the proof may determine why this a bad choice, and more importantly, what a good choice would be.

In this section we look at an approach to induction rule selection which overcome the second problem by delaying the choice of induction rule until the middle of the proof. In §2.8 two techniques which tackle the first problem are examined.

2.7.1 *Periwinkle*: Middle-Out Induction Selection

In her *Periwinkle* system Ina Kraan used *middle-out reasoning* to select an induction rule from a prestored set [Kraan, 1994, Kraan et al., 1996]. The system performs better than recursion analysis because it looks ahead into the proof by delaying the choice of induction rule, via middle-out reasoning [Bundy et al., 1990a]. This represents unknown terms in a proof with meta-variables, variables that may be instantiated to first or higher-order objects in the object level language. As the proof proceeds, the meta-variables are instantiated to allow proof steps to happen. If the proof is completed then the meta-variables should be instantiated to the required terms. The main difficulty with this technique is controlling instantiation, as without proper control the proof could easily diverge. A model of the structure of successful proofs is required to provide this control [Hesketh, 1991].

In the case of middle-out induction, second-order meta-variables are used to represent the induction terms of the as yet unknown induction rule choice. This gives us a *schematic step case* to be rippled, e.g. [9]

$$\begin{aligned} x+(y+z) &= (x+y)+z \\ \vdash \boxed{\mathcal{A}(\underline{x})} + (\boxed{\mathcal{B}(\underline{y})} + \boxed{\mathcal{C}(\underline{z})}) &= (\boxed{\mathcal{A}(\underline{x})} + \boxed{\mathcal{B}(\underline{y})}) + \boxed{\mathcal{C}(\underline{z})} \end{aligned} \tag{2.11}$$

A proof of this step-case is searched for, instantiating the meta-variables as it proceeds. If successful, this yields a set of induction terms and uses these to select an induction rule from a prestored set. The proofs of the base cases are then completed.

The dashed wave fronts in (2.11) indicate *potential* wave fronts, which can be made *definite* to allow a wave rule to match during the proof. Steps in the rippling proof are either *definite* or *speculative*, depending on whether or not any definite wave fronts

[9] Here the meta-variables are written as $\mathcal{A}, \mathcal{B}, \mathcal{C}, \ldots$

are rippled. An example of a speculative ripple is the application of rule (2.5) to the schematic step-case (2.11) to give the conclusion:

$$\boxed{s(\underline{\boxed{\mathcal{D}(\underline{x})} + (\boxed{\mathcal{B}(\underline{y})} + \boxed{\mathcal{C}(\underline{z})}}))}^{\uparrow} = (\boxed{s(\underline{\boxed{\mathcal{D}(\underline{x})}})}^{\uparrow} + \boxed{\mathcal{B}(\underline{y})}) + \boxed{\mathcal{C}(\underline{z})} \tag{2.12}$$

In this example A has been instantiated to $\lambda u.\, s(\mathcal{D}(u))$ and a definite wave front has been created around these terms. On the LHS this wave front is rippled outwards and there is now the possibility of a definite ripple on the RHS.

Rippling provides much of the control necessary for this kind of middle-out reasoning, as it provides a strong model of step-case proofs and so severely restricts the applicable rewrite rules. However, it is non-terminating in the presence of meta-variables, and so Kraan imposed further restrictions on the step-case proof. Firstly, a definite ripple or the application of the induction hypothesis is always preferred to a speculative ripple. Secondly, only a single speculative ripple is allowed during a proof. This second limitation is somewhat over-restrictive, and Kraan suggests alternative methods need to be developed, such as a middle-out induction proof critic[10]

Another potential problem with middle-out induction is the need for higher-order unification when instantiating meta-variables, as this is only semi-decidable and does not guaranteed a unique most general unifier. Therefore *Periwinkle* is restricted to unifying *higher-order patterns*, a subset of higher-order terms with decidable unification and a unique most general unifier. This subset appears to be sufficient for representing induction terms. In contrast, other approaches to middle-out reasoning have accepted the undecidability of full unification [Hesketh, 1991, Ireland and Bundy, 1996].

Note that *Periwinkle* cannot find a step cases which are destructor style (i.e. with compound induction terms in the hypotheses) or which have multiple inductive hypotheses.

[10] This problem is addressed in Chapter 7.

2.8 Creating Novel Induction Rules

Recall that apart from lack of foresight into the proof, the other major disadvantage of recursion and ripple analysis was their dependence on a space of induction rules predetermined by available recursive functions, and possibly the user. In this section we look at techniques which lift this restriction, in that they can create induction rules 'from scratch'.

2.8.1 Labelled Fragments

The need for novel induction orderings is especially important in proofs of existence theorems. Here an assertion can be made about a recursive function without its recursive structure being known. For example, the following theorem asserts the existence of a quotient q and remainder r for pairs of Peano natural numbers:

$$\forall x, y{:}nat.\, \exists u, v{:}nat.\, y \neq 0 \rightarrow (x = (q \times y) + r \wedge r < y)$$

Proving the theorem involves finding witnesses for the unknowns q and r and showing they satisfy the theorem. This can be done by synthesising the witness during the proof of the theorem. Therefore the form of induction used will determine the recursive structure of each witness. In many cases the appropriate form of induction is not dual to any recursive function given in the problem specification [Bundy, 2001]. Also, although the conjecture may be provable using known induction rules, another witness with a simpler proof may be found with other forms of induction[11]. This is the case for the quotient-remainder example above [Hutter, 1994]. Roughly speaking, we may not even have the appropriate dual induction to hand, as we don't yet know what form

[11] This is especially important when synthesising a program from its specification – an existence theorem – as a more desirable program may correspond to this witness.

the witness will be. Hence approaches such as recursion analysis, which rely on a predetermined space of induction rules, perform badly on existence proofs.

[Hutter, 1994] describes a dynamic approach for constructing an induction ordering appropriate to a given existence conjecture. First a set of induction variables are selected by using abstracted $\mathcal{C}$-equations (see §2.5) called *labelled fragments*. A set of variables is found such that context, or wave fronts in rippling terminology, introduced at these positions could be rippled out. The analysis ignores the form these wave fronts might take and only checks if there are $\mathcal{C}$-equations that could move *some* wave front in the right direction.

A destructor style induction rule is then synthesised by 'blowing up' some part of the conjecture using a context creating $\mathcal{C}$-equation (equivalent to a creational wave rule) and then propagating these wave fronts to the induction variables to give a set of induction terms. The resulting induction rule can lie outside the dual induction space, so the technique may create a 'novel' induction rule. Walther's methods (see §2.9.1) are used to establish this induction as well-founded. Hutter gives various heuristic strategies for creating and moving wave fronts in existence theorems.

2.8.2 Lazy Induction

Lazy induction is another approach designed to generate induction rules not constructed from known function definitions [Protzen, 1994, Protzen, 1995]. It is restricted to destructor-style induction rules, where induction terms are only substituted into the hypotheses.

The technique constructs a destructor style induction rule during the proof search for the rule's base and step cases. It assumes the conclusion of each case is equivalent to the conjecture, then creates and removes wave fronts using ripple-like rewriting, and generates suitable induction hypotheses on demand. Protzen uses Hutter's C-Calculus, adapted to have wave annotation's directed wave fronts. This enables the usual general termination argument (see §2.5).

This lazy generation of induction hypotheses leaves decisions about the form of induction to fertilisation steps. Before fertilisation there is no explicit representation of the unknown induction. This can be constrasted with Kraan's 'schematic step case' approach (see §2.7.1), where these decisions are made by rippling steps and meta-variables store this information explicitly before fertilisation occurs. These different approaches to delayed commitment are contrasted further in §13.5.

To prove a conjecture $\forall x_1, \ldots, x_n. \psi$, lazy induction begins with the conclusion ψ and an empty hypothesis list, and transforms it using the following operations:

Wave Front Introduction: Rewriting the conclusion with a measure increasing wave rule in order to create wave fronts. This is always the first step of the proof.

Rippling: Wave fronts are rippled outwards, or into sinks.

Case Split: Rewriting motivates a case-split. Each case becomes a separate case of the induction rule.

Hypothesis Generation: If an instance of the conjecture can be used to rewrite the conclusion, then it is added as a hypothesis and used for fertilisation.

Equate Induction Variables: If **Hypothesis Generation** can't be applied because two occurrences of an induction variable have to be instantiated to different terms, then attempt to prove that these terms are equal.

Note that **Wave Front Introduction** is always applicable to the conclusion, so further controls are required to prevent divergence. Protzen's thesis [Protzen, 1995] does not deal with this issue.

The well-foundedness of the resulting rule is guaranteed by using Walther's estimation calculus (see §2.9.1) at the end of the proof, or by ensuring:

a) that only defining equations of terminating functions are used by **Wave Front Introduction**,

b) that only p-bounded functions are moved towards induction variables, where p is the argument containing the variable and

c) that a subset of variables always appear in their induction terms, and at least one of these is instantiated to a non-variable term in each hypothesis (this condition is not made explicit by Protzen, but it seems to me to be necessary).

As the case-structure of the induction rule has been constructed by case-splits during the proof, it is guaranteed to be case complete, and hence sound.

[Protzen, 1995] reports that lazy induction was implemented as an extension to the INKA inductive theorem prover [Hutter and Sengler, 1996], although this implementation is no longer available.

An Example of Lazy Induction

To illustrate lazy induction, we now present a proof of the theorem *evenp* from the *Clam* library [van Harmelen, 1996] using destructor style definitions of *even* and $+$ (see Appendix B). We assume the lemma $even(s(s(x))) = even(x)$ is available.

$$\vdash even(x) \wedge even(y) \rightarrow even(x+y)$$

Case Split and **Wave Front Introduction** using the definition of *even* gives three cases:

$$\vdash \quad even(0) \wedge even(y) \rightarrow even(0+y) \tag{2.13}$$

$$\vdash \quad even(s(0)) \wedge even(y) \rightarrow even(s(0)+y) \tag{2.14}$$

$$x \neq 0, x \neq s(0) \quad \vdash \quad even(\boxed{p(p(\underline{x}))}^{\uparrow}) \wedge even(y) \rightarrow even(x+y) \tag{2.15}$$

The cases (2.13) and (2.14) are trivial, and case (2.15) continues with two **Wave Front Introductions** with the definition of $+$:

$$x \neq 0, x \neq s(0) \quad \vdash \quad even(\boxed{p(p(\underline{x}))}^{\uparrow}) \wedge even(y) \rightarrow even(\boxed{s(s(\underline{\boxed{p(p(\underline{x}))}^{\uparrow} + y}))}^{\uparrow})$$

Rippling with the lemma gives us:

$$x \neq 0, x \neq s(0) \quad \vdash \quad even(\boxed{p(p(\underline{x}))}^{\uparrow}) \wedge even(y) \rightarrow even(\boxed{p(p(\underline{x}))}^{\uparrow} + y)$$

Now **Hypothesis Generation** can produce a suitable induction hypothesis, use it to fertilise:

$$x \neq 0, x \neq s(0), even(p(p(x))) \wedge even(y) \rightarrow even(p(p(x))+y) \quad \vdash \quad true$$

The proof satisfies the well-foundedness conditions (a)–(c) given above, so the induction is sound and the proof is complete.

2.9 Termination Analysis

The problem of proving a given induction rule well-founded is similar to proving the termination of a recursive function — both require a well-founded relation to be provided under which the recursive cases decrease. It is in this context that the significant research into automating well-foundedness proofs has been done. Termination of a recursive function[12] is usually established by proving that for some subset P of the function argument positions, there is a *termination function* which is always less by some known well-founded order for the values in the recursive calls than the initial values. More formally, for an n-ary function f there is some fixed $P = \{i_1, \ldots, i_k\} \subseteq \{1, \ldots, n\}$, a termination function m and a well-founded relation $<$, such that for each recursive call in a defining equation:

$$\varphi \rightarrow f(a_1, \ldots, a_n) = \ldots f(b_1, \ldots, b_n) \ldots$$

the following *termination hypothesis* is true:

$$\varphi \rightarrow m(b_{i_1}, \ldots, b_{i_k}) < m(a_{i_1}, \ldots, a_{i_k}) \tag{2.16}$$

In this section we look at three approaches which allow this process to be automated.

2.9.1 The Estimation Calculus

In the Boyer-Moore theorem prover (see §2.6), proving termination of recursive functions was given a degree of automation. However, the system depended entirely on the presence of suitable *induction lemmas* to prove termination. It was up to the user to formulate these lemmas, and this constituted the most difficult part of the process.

Walther's *estimation calculus* [Walther, 1988, Walther, 1994b] attempts to automatically prove termination in a way similar to the Boyer-Moore theorem prover, but has the ability to synthesise suitable induction lemmas without user assistance. It can prove termination of destructor style functions defined over freely generated data types. It uses a single kind of measure function, the *size order* $\#_\tau : \tau \mapsto \mathbb{N}$, which counts the

[12] not including mutual or nested recursive functions.

number of *reflexive constructors* in a constructor ground term of type τ, i.e. the constructors of type $\tau^n \rightarrow \tau$ for any n. For example, the number of occurrences of s in a nat, or $cons$ in a $list(nat)$.

The method depends upon the notion of *argument bounded functions*. A function f is p-bounded iff for all terms $t_1, \ldots, t_n$ of the correct type[13]:

$$f(t_1, \ldots, t_n) \leqslant_{\#} t_p$$

Such properties of functions can be found automatically. Given a p-bounded function f, a *difference function* $\Delta^p f$ is synthesised – a predicate that recognises when the bound $\leqslant_{\#}$ can be made strict. Hence:

$$\Delta^p f(t_p) \rightarrow f(t_1, \ldots, t_n) <_{\#} t_p$$

This will play the rôle of an induction lemma. The actual estimation calculus can be used to deduce that for some term t containing variable x, $t \leqslant_{\#} x$. It decomposes t to a series of subterms by replacing the top p-bounded function by the pth argument, eventually reaching x. The *difference equivalent* $\Delta(t,x)$ is simultaneously constructed by the calculus as the disjunction of the corresponding difference functions applied to each subterm. From this and the induction lemmata, it follows that:

$$\Delta(t,x) \rightarrow t <_{\#} x$$

Showing the termination of single argument destructor style recursive function f involves:

1. Finding each recursive call $\varphi \rightarrow f(x) = \ldots f(t) \ldots$
2. Deducing $t \leqslant_{\#} x$ in the estimation calculus.

[13] where $x \leqslant_{\#} y$ denotes $\#_{\tau}(x) \leqslant_{\mathbb{N}} \#_{\tau}(y)$

3. Proving the *termination hypothesis* $\varphi \rightarrow \Delta(t,x)$.

The technique extends to functions with several arguments. It has been implemented in the INKA inductive theorem prover [Hutter and Sengler, 1996] and shown to be successful in practice. Walther's methods have also been extended for use with arbitrary polynomial-norm measure functions [Giesl, 1995a] and recursive functions defined over non-freely generated data types [Sengler, 1996].

2.9.2 Reducer/Conserver Analysis

[McAllester and Arkoudas, 1996] describes a simplification of the estimation calculus, which is guaranteed terminating for the class of 'Walther recursive' functions. Walther recursion is defined by set of simple syntactic requirements for function and type definitions.

Functions can be classified as *conservers* or *reducers* of their *p*th arguments, corresponding to a non-strict or strict bound on the function by the argument. These are expressed in conserver and reducer lemmas using the $\leqslant_{\#}$ relation – a reducer f can be asserted by a reducer lemma of the form:

$$f(x_1,\ldots,x_n) \leq_{\#} d(x_p)$$

where d is a destructor function[14]. The simplified calculus uses reducer and conserver lemmas in two capacities: as a termination checker or as a way of obtaining new lemmas from known terminating functions. Unlike the estimation calculus it does not produce termination hypotheses that require an inductive theorem prover to discharge, but purely by manipulating reducer and conserver lemmas. Hence it is less powerful than Walther's original methods, but has the advantage of always terminating for Walther recursive functions.

[14] i.e. $d(c(x_1,\ldots,x_n)) = x_i$ for some constructor c and $i \in [n]$.

2.9.3 Using Term Orders

There exists large bodies of research on the termination of logic programs and term rewriting systems (see [Dershowitz, 1987] and [Schreye and Decorte, 1994] respectively). This suggests an alternative route to automating termination proofs of recursive functions, and equivalently well-foundedness proofs of induction rules, than those outlined above: adapt automated techniques from these areas to deal with recursive functions.

Giesl has considered this approach and concluded that [Giesl, 1995c]:

- Techniques for logic program analysis are currently unsuitable, as these are only semi-automatic, i.e. like Boyer and Moore's system they require the user to perform the significant tasks.
- Although there are several automated procedures for term rewriting systems, these are not directly applicable to recursive functions.

The problem with the latter techniques is that they prove termination hypotheses (2.16) using *term orders* – well-founded orders on the terms of the data types $a_1, \ldots, a_n$. In general, this approach is not sound for recursive functions, because different terms may evaluate to the same constructor term, but will not be equivalent under the term order. For example, nil and $delete(0, 0 :: nil)$ are equivalent, but will not be treated as such by a term order. Term orders do not always respect the semantics of functions.

[Giesl, 1995c] describes three possible solutions to this problem:

1. Use term orders which respect the semantics of the recursive functions.
2. Consider recursive functions as term rewriting systems.
3. Eliminate defined functions from the termination hypotheses.

The former two are rejected on the grounds that they impose strong requirements on the termination proofs, which would significantly reduce the power of the approach. Giesl develops the third solution, introducing new undefined functions which bound the defined functions that are to be eliminated. He describes a procedure for transforming a set of termination hypotheses into a set of constraints, where defined functions symbols are replaced by the new undefined function symbols. Any well-founded term order satisfying the constraints will also satisfy the original termination hypotheses.

Automatic techniques for the synthesis of wellfounded term orders can now be used to prove the termination of the recursive function, e.g. those in [Steinbach, 1995], [Giesl, 1995b] or [Dershowitz and Hoot, 1993].

2.10 Summary

This chapter has surveyed the literature on proof planning and automating inductive proof. We draw attention to the following features:

- Proof planning provides a theorem proving architecture that allows a declarative specification of proof strategies. Far greater search control can be exercised than with object-level search.

- Rippling is a heuristic technique for controlling search in inductive step cases. The expectations it provides for the form of the step case proof have allowed researchers to make progress with other problems in automated induction: generalisation, lemma speculation and rule selection.

- The standard technique for induction rule selection is recursion analysis. It is limited to selecting from an incomplete space of induction rules determined by the recursive functions known to the system. Its one-step lookahead into the proof can also be inadequate.

- Improvements on recursion analysis have delayed the choice of induction rule until later in the proof. Some approaches also attempts to generate a novel appropriate rule during the proof [Protzen, 1995]. The main problems with these techniques are poor search control and the restrictions they place on the form of induction rules.

Chapter 3

Induction Rule Structure

When dealing with such a schematic axiom, how can a prover sensibly guess which instances of (the schema) to consider? Without a really good way to answer such questions, one meets with the futility of the British Museum Algorithm[1].

— ROBERT S. BOYER & J STROTHER MOORE, ON THE DIFFICULTY OF AUTOMATING INDUCTIVE REASONING

3.1 Introduction

The literature on automated induction, described in Chapter 2, contains a variety of logical and heuristic theories of inductive proof. The relationships between these theories is not always clear: for example, R-descriptions [Walther, 1992] and rippling [Bundy et al., 1993] are described in quite different terms (see §2.5 and §2.6).

This raises important questions for anyone considering the automation of mathematical induction, namely:

1. What definition of an induction rule should be used? For example, some authors

[1] which “enumerates... all Hilbert style proofs, until it finds a proof of the given theorem, as though visiting the British Museum, where one gets to see at least one example of everything.”

restrict their theories to *destructor style* rules, e.g. [Protzen, 1995], while others also use *constructor style*, e.g. [Bundy et al., 1993].

2. Once an induction rule has been applied, how should the proof of the resulting subgoals be guided? Rippling is a successful approach. However, there are a number of possible variants (see §2.5).

These questions are particularly relevant to this thesis, in considering (i) what kind of induction rules should a system attempt to create and (ii) what constitutes a good choice with respect to the proof search heuristics being used. This chapter provides answers to both these questions[2], providing a theory for the automation of inductive proof. Our theory will be taken as a basis for the remainder of the thesis.

The concept of *simple induction rules* is proposed as a definition of induction rules that is suitable for automated proof, because it is compatible with rippling heuristics. We show that current rippling techniques are easily extended for use with this class of rule, by giving a fresh account of *creational rippling* [Bundy et al., 1993] via term embeddings. Simple induction rules generalise the concepts of induction rule used in much previous work on automated induction, and we argue that this improves automation.

Figure 3.1 shows an example of the kind of induction rule we discuss in this chapter, and illustrates some useful pieces of terminology.

3.2 Syntactic Restrictions

As explained in §2.2.1, all induction rules are derivable from the Noetherian Induction rule. However, the full rule is rarely used in automated theorem provers, because it is

[2] Any answers to these questions depend, in part, on the logical setting in which the inductive reasoning takes place. However, only a sequent-based typed higher-order logic is considered in this thesis.

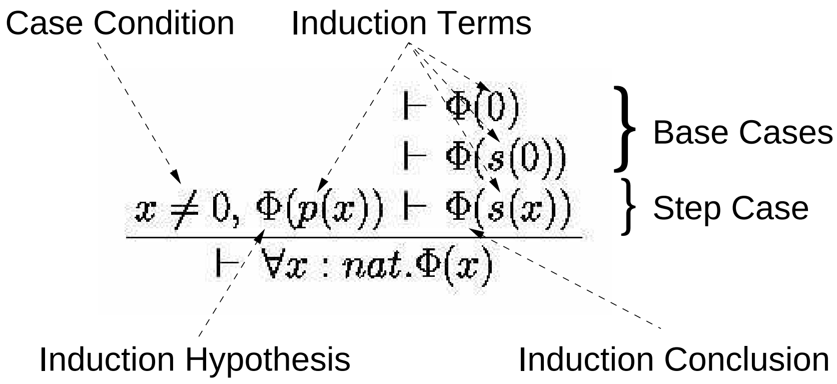

Figure 3.1: An example induction rule, with the common names for various parts.

an axiom with several higher order variables [Boyer and Moore, 1992]. This means it may not be expressible in a system's logic, and when it is, the presence of higher order variables present search and unification problems, unless carely controlled.

A system can get round this problem of expressiveness by computing the necessary instantiation 'behind the scenes' and using the resulting derived rule, e.g. NQTHM's induction rules are expressed in unquantified first-order logic. But still, such systems do not consider the full range of possible instantiations. Instead they typically employ some syntactically restricted class of induction rules.

Examples of such classes are *constructor style* induction rules, which are used by the *Clam* [van Harmelen, 1996] and λ*Clam* [Richardson et al., 2000] proof planners and the RRL system [Kapur and Zhang, 1995], and *destructor style* induction rules, which are used by NQTHM [Boyer and Moore, 1979] and the INKA system [Walther, 1992, Protzen, 1995]. Destructor style induction rules may be formalised using R-Descriptions [Walther, 1992].

The disadvantage with using a restricted class of induction rules is that there may be problems that can only be solved, or can be more easily solved, using an induction rule outside of this class. (We look at the problems of restricting a system to constructor and destructor style in §3.5.) For instance, *Clam* cannot typically solve simple problems about destructor style functions, as the appropriate rule is often destructor style. There is a tension here between generality needed to solve a range of problems, and the search control issues of using an unrestricted definition of induction rule.

3.3 Simple Induction Rules

We now describe *simple induction rules*, a class of induction rules designed to generalise constructor and destructor style rules, whilst still being suitable for automated proof.

Definition 1 (Simple Induction Rule) *A* **simple induction rule** *is an inference rule with a conclusion of the form* $\vdash \forall \mathcal{X}.\,\Phi$*, and premises of the form*

$$C_1,\ \ldots, C_k,\ \theta_1(\forall \mathcal{Y}_1.\Phi),\ \ldots, \theta_h(\forall \mathcal{Y}_h.\Phi) \quad \vdash \quad \sigma(\Phi)$$

for $h, k \geq 0$ *and substitutions* $\theta_1, \ldots, \theta_h, \sigma$ *such that*

1. *For all* $i \in [h]$
$$\mathcal{Y}_i \cup Dom(\theta_i) = Dom(\sigma) = \mathcal{X}$$

2. *For all* $i \in [k]$*, each* **case condition** C_i *is a literal* **not** *of the form* $\theta'(\forall \mathcal{Y}'.\Phi)$ *for any substitution* θ' *and set of variables* $\mathcal{Y}'$.

Informally, the definition gives a schematic description of a premise consisting of case conditions (C_i), induction hypotheses ($\theta_i(\forall \mathcal{Y}_i.\Phi)$) and induction conclusion $\sigma(\Phi)$. Clause (1) insists that in each premise the universally quantified variables in the rule's consequent ($\mathcal{X}$) are substituted for in the conclusion ($Dom(\sigma)$) and either substituted for, or universally quantified, in each hypothesis ($\mathcal{Y}_i \cup Dom(\theta_i)$). Clause (2) ensures that case conditions are not induction hypotheses.

As an example, consider the induction rule from §1.3:

$$\frac{\begin{array}{rcl} & \vdash & \Phi(nil) \\ \Phi(x) & \vdash & \Phi(x <> (y :: nil)) \end{array}}{\forall l : list(\tau).\Phi(l)}$$

This is a simple induction rule. Following Definition 1 it has $\mathcal{X} = \{l\}$. The base case has the parameters $k = 0$, $h = 0$ and $\sigma = \{nil/l\}$. The step case has the parameters $k = 0$, $h = 1$, $\mathcal{Y}_1 = \emptyset$, $\theta_1 = \{x/l\}$ and $\sigma = \{(x <> (y :: nil))/l\}$.

On the other hand, the Noetherian induction rule (see §2.2.1) is *not* a simple induction rule:

$$\frac{\forall x \in A.\ (\forall y \in A.\ y \prec x \rightarrow \Phi(y)) \rightarrow \Phi(x)}{\forall x \in A.\ \Phi(x)}$$

It is not a simple rule because its premise has the wrong syntactic structure to match Definition 1.

Simple induction rules are more general than the constructor and destructor style of induction rules found in the literature. We can obtain constructor (resp. destructor) induction rules by restricting the substitution θ_i (resp. σ) to only introduce atomic terms — variables or constant symbols.

Simple induction rules are suitable for automation because we can use rippling-like heuristics to guide the proof of the resulting subgoals: the induction conclusion $\sigma(\Phi)$ and the induction hypotheses $\theta_i(\forall\mathcal{Y}_i.\Phi)$ are both instances of the same formula, modulo universal quantification. However, before we can use simple rules and rippling together, there are some technical problems to consider.

Firstly, wave-fronts may appear in both the conclusion and hypotheses of step cases. As discussed in §2.5.4, creational rippling has been proposed as a technique for rippling hypothesis wave-fronts, but it is relatively ill-defined. It was presented in the wave-annotation formalism [Bundy et al., 1993], but has not yet been extended to the more general embeddings approach [Smaill and Green, 1996]. This problem is dealt with in the next section.

Secondly, it is possible that hypothesis and conclusion substitutions for a given variable do not share any common subterms, making the calculation of a common skeleton impossible, and so preventing standard rippling. We do not deal with this problem in this thesis, but simply note that there are proposed extensions to rippling to deal with lack of common subterms, e.g. *hole-less wave-fronts*[3]. Consquently, below we will assume that such a common subterm *does* exist.

3.4 Creational Rippling

To define creational rippling in an embeddings framework, consider a step case with conclusion C. Each induction hypothesis H is associated with a set of triples $\langle Sk, e_1, e_2 \rangle$, such that $e_1 : Sk \overset{\subseteq}{\rightarrow} H$ and $e_2 : Sk \overset{\subseteq}{\rightarrow} C$. Sk is a common skeleton that embeds in both this induction hypothesis and the conclusion.

Below we describe how the initial embeddings are computed and how creational rippling takes place. Our account differs significantly from the original wave-annotation presentation [Bundy et al., 1993], although the underlying ideas are the same.

3.4.1 Initial Embeddings

Initially we can assume $H = \theta(\forall \mathcal{Y}.\Phi)$ and $C = \sigma(\Phi)$ for substitutions θ and σ and set of variables $\mathcal{Y}$ (see Definition 1). To compute the initial triples for H we consider each

[3] Alan Bundy, unpublished research note.

$x \in Dom(\sigma) - \mathcal{Y}$ and compute the set S_x of common subterms of $\theta(x)$ and $\sigma(x)$ — as discussed above, we are assuming such a S_x is non-empty. Define a substitution ρ by

1. For $x \in Dom(\sigma) - \mathcal{Y}$ substitute a term from S_x.

2. For $x \in \mathcal{Y}$ substitute $\sigma(x)$.

It is easily shown that a common skeleton for H and C is given by $\rho(\Phi)$: For case (1) S_x is embeds into $\theta(x)$ in the hypothesis and $\sigma(x)$ in the conclusion, as it is a subterm of both. For case (2) $\sigma(x)$ embeds into a variable bound by $\forall\mathcal{Y}$ in the hypothesis — recall that any term embeds into a universal variable of the same type — and $\sigma(x)$ trivially embeds into itself in the conclusion. By Definition 1 these are the only two cases that need to be considered.

Note there may be multiple possible ρs, and so multiple common skeletons. Each skeleton has a corresponding triple, with the embeddings computed in the usual way.

3.4.2 Ripple Steps

We assume that some wave measure *wm* over embeddings is available, such as the one in [Smaill and Green, 1996]. The usual definition of rippling via embeddings is used (see §2.5), extended to cover multiple skeletons/hypotheses. Informally, a successful ripple step requires us to reduce the measure in at least one of the embeddings of the skeleton into the conclusion, and to remain constant in those that are not reduced. If the measure increases for an embedding then we can allow it to be discarded, providing that at least one viable embedding remains.

A ripple step is formally defined as follows:

1. Rewrite the conclusion.

2. Attempt to embed each skeleton into the new conclusion. At least one must embed.

3. Check the step has reduced the *sequent wave measure*.

Definition 2 (Sequent Wave Measure) *The* **sequent wave measure** wm_{seq} *is defined for a sequent* S *as a multiset, containing* $wm_{hyp}(H)$ *for each induction hypothesis* H *in* S.

The **hypothesis wave measure** wm_{hyp} *is defined for an induction hypothesis* H *as a multiset, containing* $wm(e_2)$ *for each triple* $\langle Sk, e_1, e_2 \rangle$ *associated with* H.

As the measure is reduced with each step, rippling is terminating.

The sequent wave measure characterises valid rippling steps, but it is not a good way of comparing two valid rippling steps, because needlessly throwing away hypotheses/skeletons reduces the measure. Instead, one step is prefered over another if it preserves more hypotheses, else if it preserves more skeletons.

3.4.3 Creational Ripple Steps

A creational step is one which introduces extra context into the conclusion to match context already in the hypotheses, and hence is inherently wave-measure increasing. The matching context can be made part of the skeleton, as it is shared by hypothesis and conclusion — a process called *neutralisation*, because the corresponding wavefronts ‘cancel each other out’, leaving the underlying term structure behind. Hence neutralisation expands the skeleton.

A creational ripple step is defined as:

1. Rewrite the conclusion.

2. Attempt to embed each skeleton into the new conclusion. At least one must embed.

3. Check the wave measure has increased.

4. Apply neutralisation exhaustively. It must apply at least once.

After neutralisation the wave measure may have been increased, decreased or remained constant. We now define neutralisation:

Definition 3 (Neutralisation) *A triple of formulae Sk, H and C such that $Sk \overset{\subseteq}{\rightarrow} H$ and $Sk \overset{\subseteq}{\rightarrow} C$, undergo* **neutralisation** *iff another formulae Sk' (the* expanded skeleton*) can be found such that*

1. $e : Sk \overset{\subseteq}{\rightarrow} Sk'$ *for some* $e \neq e_{id}$

2. $Sk' \overset{\subseteq}{\rightarrow} H$

3. $Sk' \overset{\subseteq}{\rightarrow} C$

Note that the above definition does not give an algorithm for neutralisation.

The termination of creational rippling is guaranteed, despite the fact that it increases the wave measure, by the following measure.

Definition 4 (Difference Measure) *The* **sequent difference measure** dm_{seq} *is defined for a sequent S as a multiset, containing $\#(H) - \#(Sk)$ for each triple $\langle Sk, e_1, e_2 \rangle$ associated with an inductive hypothesis H in S, where # measures the size of a term.*

A creational ripple step removes zero or more triples from the sequent, so the difference measure is not increased by the step. Each step is followed by at least one neutralisation, so at least one triple has its skeleton expanded. As $\#(H)$ remains constant for each triple, the value of $\#(H) - \#(Sk)$ is reduced for at least one triple. Hence the difference measure is reduced by neutralisation.

Furthermore, the combination of standard and creational ripple steps is terminating, as standard rippling preserves skeleton, and so cannot increase the difference measure. Hence a lexographic measure of the difference measure, followed by the sequent wave measure, ensures termination.

3.5 A Comparison of Rule Styles

Having defined simple induction rules and the rippling heuristics compatible with them, this section compares their use with that of constructor and destructor style rules. Recall that simple induction rule is a more general definition than both these styles. We argue that there are two significant advantages to using simple induction rules:

1. The approach is not restricted to problems concerning either constructor or destructor style functions.

2. Even if one restricts function definitions to one or the other style, using only the corresponding style of induction rule is inadequate, because of the role that lemmas can play in proofs.

3.5.1 Problem with Function Style

Restricting a system to constructor or destructor style induction makes proofs involving other styles of function definition difficult. This is consistent with Stevens account of inductive proof [Stevens, 1990], which, put simply, is that a dual rule of a relevant function f should be used, and the term structure this introduces must be removed/matched using the definition of f. If f is constructor (resp. destructor) style, the suitable dual rule is constructor (resp. destructor) style. A ‘destructor style only’ system will have difficulty dealing with f, because a suitable destructor style rule will introduce term structure into the step case hypotheses which cannot be removed by the constructor style definition of f. A similar argument holds against ‘constructor style only’ systems.

Of course, this is an over simplification: lemmas could be used to remove/match the problematic term structure, so it may be possible for a destructor style system to work with constructor style functions, and vice versa. However, there is no guarantee that suitable lemmas will be provided, or that they can be easily generated, or even that such lemmas will exist.

3.5.2 Problem with the Use of Lemmas

The problem with function style is not necessarily significant, as many authors choose to work with one particular function style. However, there is a more compelling argument to use simple induction rules, given that Stevens’s theory does not account for all inductive proofs. There are theorems that require non-dual inductions for their solution (see [Protzen, 1995] or §1). One possible scenario is that a non-dual rule can be used because it introduces term structure that can be removed by a given lemma — extending Stevens’s theory, we could say the induction rule was a dual rule to the lemma. By analogy with the functional case, lemmas could be classed as constructor or destructor style, depending on the style of their dual rules.

The problem arises because there is no guarantee that the style of the given lemma will be the same as the chosen function style. For example, a destructor style system may have access to a constructor style lemma about its destructor style functions which suggests a suitable dual rule for a particular problem. Such a system could not find the straightforward solution of using this lemma to suggest the constructor style dual rule. Of course, it may find a solution if it has a suitable destructor style lemma, but as is often the case in inductive theorem proving (or theorem proving in general) this may not be available. Thus the destructor style system fails to take advantage of the lemma resources made available to it. A similar argument holds against constructor style systems. Hence restricting the rule style can reduce the power of a system even if one sticks to the corresponding function style.

3.6 Summary

In this chapter we have:

- Discussed the problem of choosing a suitable definition of induction rule that we have argued is specific enough to avoid the search control problems of the full higher order schema, yet also general enough to have a wide coverage of inductive problems.

- Proposed simple induction rules as a class of induction rules suitable for the automation of inductive proof, due to their compatibility with rippling heuristics.

- Presented creational rippling [Bundy et al., 1993] via a novel formulation that uses term embeddings [Smaill and Green, 1996] instead of the original wave-annotation. This enables the term embeddings formulation of rippling to be used with simple induction rules.

- Argued that restricting a system to constructor or destructor style induction rules has significant disadvantages compared to using simple induction rules. Problems arise even if the system only works with the corresponding function style.

Chapter 4

Step Case Creation

4.1 Introduction

Having outlined our induction proof strategy in Chapter 1, we now describe in detail the techniques used to try to create a successful step case for the inductive proof of a given goal. The central idea is that certain choices about the form of the case are left undecided until the middle of its proof — a technique known as *middle-out reasoning* [Hesketh, 1991]. The effect of such decisions is not known beforehand, but in the middle of the proof attempt more information may be available, making a better choice possible.

We take a 'least commitment' approach of delaying these choices as long as possible — only when the proof attempt cannot progress any further is the strategy forced to commit. The rippling heuristic is used to control rewriting.

Overview

First, in §4.2 we present the *step case schema*, which is used to represent an unknown step case. Considering only constructor style step cases, in §4.3 we describe how the schema may be refined during proof search to give a successful step case formula.

We then consider non-constructor style step cases in §4.4. Using the new formulation of the creational rippling heuristic given in Chapter 3, an extended proof strategy for generating non-constructor style step cases is described.

Some of the ideas presented here build on previous research on inductive proof described in [Kraan, 1994] and [Protzen, 1995]. We comment on this, where appropriate, but defer a fuller comparison to Chapter 13.

4.2 The Step Case Schema

At the beginning of the inductive proof, the unknown step case is represented by a *step case schema*, which uses meta-level variables to represent parts of the object-level formulae which are yet to be determined. Instantiation of these meta-variables (e.g. by unification) will take place during the proof search, yielding an concrete case of an inductive proof.

In [Kraan, 1994], step case schemas were used in a similar way. Those schemas could represent step cases of simple constructor style induction rules with single hypotheses and no non-inductive hypotheses. Simple induction rules were selected in Chapter 3 as a suitable class for automatic proof, and here we generalise the schema approach accordingly.

Because the strategy will generate induction hypotheses dynamically, simple induction rules can be simplified further: induction hypotheses of the form $\theta(\forall\mathcal{Y}.\Phi)$ can be replaced with several of the form $\theta'(\Phi)$ — one for each set of values $\mathcal{Y}$ needs to take during the proof. These can be added during the proof as required, rather than the more general universally quantified version being used each time. We refer to these simple induction rules without universal quantification as *sink-free* simple induction rules, following rippling terminology [Bundy et al., 1993].

Recall from Definition 1 that a simple induction rule with the conclusion $\forall \mathcal{X}.\,\Phi$. We can write this as:

$$\forall x_1{:}\tau_1.\ldots\forall x_n{:}\tau_n.\,\phi(\vec{x})$$

has premises that are sequents with the following parts:

- A single conclusion $\sigma(\Phi)$, which we can write this as $\phi(\vec{t})$ for terms $\vec{t}$. We can represent it schematically as $\phi(T_1(\vec{x}),\ldots,T_n(\vec{x}))$.

- One or more induction hypotheses of the form $\forall\mathcal{Y}.\Phi$, which we can write as $\forall\mathcal{Y}.\phi(\vec{t})$ for terms $\vec{t}$. As argued above, these can each be simplified to $\phi(\vec{t})$, so represented schematically as the first induction hypothesis $\phi(S_1(\vec{x}),\ldots,S_n(\vec{x}))$ and a (possibly empty) list of additional induction hypotheses $IH(\vec{x})$ — each will have the same schematic form.

- Zero or more non-induction hypotheses, known as *case conditions*, represented schematically as $CC(\vec{x})$, which may be trivial i.e. $CC = \lambda\vec{u}.true$.

Hence the step case can be represented by the following schema:

$$\begin{array}{ll} CC(\vec{x}), & \\ \phi(S_1(\vec{x}),\ldots,S_n(\vec{x})), & \\ IH(\vec{x}) & \\ \vdash \quad \phi(T_1(\vec{x}),\ldots,T_n(\vec{x})) & \end{array} \tag{4.1}$$

The schema can be instantiated to give any step case from a sink-free simple induction rule.

Note that our step case schema (4.1) appears to assume that the induction constants in the induction hypothesis and conclusion will be of the same type as the universally quantified variables in the original conjecture, as we use $\vec{x} = x_1, \ldots, x_n$ to denote them both. This cannot be the case, as induction often requires more induction constants than there are universal variables. For example, structural induction on type $list(\tau)$ would use an induction constants of types τ and $list(\tau)$. Given a conjecture $\forall l{:}list(\tau).\phi(l)$, the 'best approximation' schema (4.1) can make to the required step case is:

$$\phi(x) \quad \vdash \quad \phi(T_1(x) :: x)$$

However, we can easily overcome this limitation by ensuring that when the object-level induction rule is constructed from the proof plan, we replace any remaining meta-variables, such as $T_1(x)$, with induction constants of the appropriate type.

Example 1

The goal $\forall x,y{:}nat.\ x+y=y+x$ has the step case schema:

$$\begin{aligned}
&CC(x,y),\\
&S_1(x,y)+S_2(x,y)=S_2(x,y)+S_1(x,y),\\
&IH(x,y)\\
&\vdash \quad T_1(x,y)+T_2(x,y)=T_2(x,y)+T_1(x,y)
\end{aligned} \tag{4.2}$$

4.3 Constructor Schema Refinement

Given the initial step case schema for a goal, an attempt is made to generate a proof of the schema. The schema is refined by instantiating (possibly partially) the meta-variables during certain proof steps.

In the following sections, we describe the various proof steps which are employed by the strategy for creating constructor style step cases. Non-constructor cases are considered in §4.4.

The constructor style schema is:

$$CC(\vec{x}), \quad \phi(\vec{x}), \quad IH(\vec{x}), \qquad \vdash \quad \phi(T_1(\vec{x}), \ldots, T_n(\vec{x})) \tag{4.3}$$

Note that we may have multiple hypotheses in a constructor style step case and that an induction variable may be instantiated to a different constant in two different hypotheses.

4.3.1 Rippling

The step case proof is controlled by the ripple heuristic. Following [Kraan, 1994], rippling may partially instantiate meta-variables as a side-effect of rewriting: the left-hand side of the rule is unified with the redex via higher-order unification [Huet, 1975]. Flexible redexes are forbidden, i.e. rewrite rules are never applied just to meta-variables. This condition prevents a situation where *every* rewrite rule is applicable to *every* schematic term.

Rippling-sideways and -in is not useful if we are using a constructor style schema. If a wave front were rippled in it could not be dealt with, because the induction hypothesis would contain no universal variables or induction terms. To deal with such proofs that use rippling-in a more general non-constructor schema is required — we describe such a schema in §4.4.

Wave Annotation via Embeddings

The embeddings representation of rippling's wave annotations is used, to allow compability with higher-order syntax [Smaill and Green, 1996]. We use the definition of embedding given in §2.5, with the minor adaptation that a term embeds into a meta-variable if and only if their object-level types are compatible. This prevents redundant embeddings, e.g. a list l embedding into the non-list first argument of the schematic term $cons(A(l),H(l))$.

Following Kraan, we denote the wave annotation around meta-variables with a dotted box, called *a potential wave-front*. Hence the initial annotated version of schema (4.3) is:

$$CC(\vec{x}), \quad \phi(\vec{x}), \quad IH(\vec{x}), \qquad \vdash \quad \phi(\boxed{T_1(\vec{x})},\ldots,\boxed{T_n(\vec{x})}) \tag{4.4}$$

Note, however, that unlike Kraan's, our potential wave-fronts have no wave-holes — this is because *any* term may be embedded into a meta-variable, modulo type, not just the elements of $\vec{x}$.

The embeddings formulation also allows a simple treatment of annotation with respect to multiple hypotheses, compared to [Yoshida et al., 1994], for example. See Chapter 3 for details.

Rippling Side Conditions

In addition, the rewrite step may have a side condition. Unless this evaluates to *true* under symbolic evaluation, or it already appears in the case conditions, it is added to the case conditions. This is done by forcing partial instantiation of the meta-variable representing the unknown case conditions, allowing the creation of step cases with case conditions.

For example, consider the following schematic goal:

$$CC(x_1,x_2), \quad \ldots \quad \vdash \quad len(delete(\boxed{T_1(x_1,x_2)},\boxed{T_2(x_1,x_2)})) \leq len(\boxed{T_2(x_1,x_2)})$$

We can apply the following wave rule by assuming the side condition:

$$X \neq H \rightarrow \quad delete(X, \boxed{H :: \underline{T}}^{\uparrow}) \quad \Rightarrow \quad \boxed{H :: \underline{delete(X,T)}}^{\uparrow}$$

The resulting schematic goal is:

$$\begin{array}{rl} & T_1(x_1,x_2) \neq T_2'(x_1,x_2) \wedge CC'(x_1,x_2),\ \ldots \\ \vdash & len(\boxed{T_2'(x_1,x_2) :: \underline{delete(\boxed{T_1(x_1,x_2)}, \boxed{T_2''(x_1,x_2)})}}^{\uparrow}) \\ & \quad \leq \quad len(\boxed{T_2'(x_1,x_2) :: \underline{\boxed{T_2''(x_1,x_2)}}}^{\uparrow}) \end{array}$$

T_2' and T_2'' are the meta-variables remaining after the partial instantiation of T_2, where

$$T_2(x_1,x_2) = T'(x_1,x_2) :: T''(x_1,x_2)$$

The new meta-variable CC' may be instantiated by further side conditions.

In general, if the unknown case conditions are represented by the schematic hypothesis $CC(\vec{x})$, adding the condition $cond(\vec{x})$ gives the following instantiation:

$$CC = \lambda\vec{x}.\ cond(\vec{x}) \wedge CC'(\vec{x})$$

where CC' is the remaining unknown condition.

Speculative and Definite Ripples

As explained in §2.7.1, Kraan distinguished between *speculative* ripple steps, which partially instantiate meta-variables via rewriting, and *definite* ripple steps, which do not. Speculative steps may increase the wave-measure that guarantees the termination of rippling [Basin and Walsh, 1996]. Hence if unbounded speculative steps are permitted, rippling may not terminate.

Kraan's solution of placing a bound on the number of such steps ensures termination, but there may be proofs that require more speculative steps than this bound permits. This is illustrated by examining Example 1 further.

Example 1 (revisited)

The step case schema (4.2) may be rewritten using a wave-rule taken from the definition of $+$:

$$\boxed{s(\underline{U})}^{\uparrow} + V \;\Rightarrow\; \boxed{s(\underline{U+V})}^{\uparrow}$$

The LHS of the conclusion is rewritten:

$$\begin{array}{rcl}
\bbox[border:1px dashed]{C(x,y)} + \bbox[border:1px dashed]{D(x,y)} & = & \bbox[border:1px dashed]{D(x,y)} + \bbox[border:1px dashed]{C(x,y)} \\
 & \Downarrow & \\
\boxed{s(\underline{\bbox[border:1px dashed]{C'(x,y)} + \bbox[border:1px dashed]{D(x,y)}})}^{\uparrow} & = & \bbox[border:1px dashed]{D(x,y)} + \boxed{s(\underline{\bbox[border:1px dashed]{C'(x,y)}})}^{\uparrow}
\end{array}$$

This is a speculative ripple, as C is partially instantiated to $\lambda u.\lambda v.C'(u,v)$ as a by-product. If unbounded, speculative ripples can be applied *ad infinitum*:

$$\begin{array}{rcl}
\boxed{s(\underline{\bbox[border:1px dashed]{C'(x,y)} + \bbox[border:1px dashed]{D(x,y)}})}^{\uparrow} & = & \bbox[border:1px dashed]{D(x,y)} + \boxed{s(\underline{\bbox[border:1px dashed]{C'(x,y)}})}^{\uparrow} \\
 & \Downarrow & \\
\boxed{s(s(\underline{\bbox[border:1px dashed]{C''(x,y)} + \bbox[border:1px dashed]{D(x,y)}}))}^{\uparrow} & = & \bbox[border:1px dashed]{D(x,y)} + \boxed{s(s(\underline{\bbox[border:1px dashed]{C''(x,y)}}))}^{\uparrow} \\
 & \Downarrow & \\
\boxed{s(s(s(\underline{\bbox[border:1px dashed]{C'''(x,y)} + \bbox[border:1px dashed]{D(x,y)}})))}^{\uparrow} & = & \bbox[border:1px dashed]{D(x,y)} + \boxed{s(s(s(\underline{\bbox[border:1px dashed]{C'''(x,y)}})))}^{\uparrow} \\
 & \Downarrow & \\
 & \ldots &
\end{array}$$

This problem is revisited in Chapter 7 — for now we assume some arbitrary bound.

4.3.2 Post-Rippling

Strong and weak fertilisation are identical to the non-schematic case, except that, as during rippling, meta-variables may be instantiated. A hypothesis may be used in weak fertilisation several times, so it is oriented as a rewrite rule after its first use, to prevent looping. A similar technique was employed in the *Clam 2* proof planner [van Harmelen, 1996].

At the end of a successful rippling proof, tidying-up may be required *post*-fertilisation, as some meta-variables may be not fully instantiated. To obtain a non-schematic step case and proof, each meta-variable representing a term is replaced by a fresh object-level variable and those representing propositions to $\lambda\vec{x}.true$.

4.3.3 Multiple Induction Hypotheses

The initial step case schema has a single induction hypothesis, but some proofs involve multiple hypotheses. To generate additional induction hypotheses *Protzen's Heuristic* is employed [Protzen, 1995]: if an instance of the original goal can be used to rewrite the conclusion, add it as an induction hypothesis and apply it. This rewrite step should not instantiate meta-variables in the conclusion. The hypothesis is added to the step case by partially instantiating the meta-variable representing the unknown induction hypotheses.

The main problem with this heuristic is its over-applicability. It can be applied to most goals in several ways, significantly increasing the search space. To control its use in the step case proof, it must be applied after rippling is blocked, and whenever possible, fertilisation is prefered instead.

Example 2

The schematic step case for the goal:

$$\forall t{:}btree(nat).\ sum(t) = sum(flip(t))$$

may be fully rippled to:

$$\begin{array}{rcl} & CC(t), \quad sum(t) & = & sum(flip(t)), \quad IH(t) \\ \vdash & \boxed{\underline{sum(\lfloor L(t) \rfloor)} + \underline{sum(\lfloor R(t) \rfloor)}}^{\uparrow} & = & \boxed{\underline{sum(flip(\lfloor R(t) \rfloor))} + \underline{sum(flip(\lfloor L(t) \rfloor))}}^{\uparrow} \end{array}$$

using the following wave rules from the definitions of sum and $flip$:

$$\begin{array}{rcl} sum(\boxed{node(\underline{X},\underline{Y})}^{\uparrow}) & \Rightarrow & \boxed{\underline{sum(X)} + \underline{sum(Y)}}^{\uparrow} \\ flip(\boxed{node(\underline{X},\underline{Y})}^{\uparrow}) & \Rightarrow & \boxed{node(\underline{flip(Y)}, \underline{flip(X)})}^{\uparrow} \end{array}$$

Weak fertilisation instantiates L to $\lambda u.\ u$, giving:

$$\begin{array}{rcl} & CC(t), \quad sum(t) & = & sum(flip(t)), \quad IH(t) \\ \vdash & \boxed{\underline{sum(t)} + \underline{sum(\lfloor R(t) \rfloor)}}^{\uparrow} & = & \boxed{\underline{sum(flip(\lfloor R(t) \rfloor))} + \underline{sum(t)}}^{\uparrow} \end{array}$$

Repeating weak fertilisation with the same hypothesis completes the proof. The resulting step case is:

$$\Phi(t) \quad \vdash \quad \Phi(node(t,t))$$

Our induction strategy will fail to build a complete induction rule using this step case.

Backtracking over the second fertilisation, Protzen's Heuristic can be applied: an instance $sum(t') = sum(flip(t'))$ of the original goal is added as an induction hypothesis. Applying it instantiates R to $\lambda u.\ t'$ and completes the proof. The resulting step case is:

$$\Phi(t),\ \Phi(t') \quad \vdash \quad \Phi(node(t,t'))$$

This can be used to construct a complete proof.

4.3.4 A Constructor Proof Strategy

The proof strategy for creating constructor style step cases, parameterised by a bound $N > 0$, can be summarised as follows,

1. Construct annotated schema for goal.

2. Ripple-out, with no more than $N > 0$ speculative ripples.

3. Either:

   - Fertilise with known hypothesis.

   - Create hypothesis and fertilise.

4. If goal is open, goto step 2, else collapse remaining meta-variables.

4.4 Extension to Non-Constructor Cases

This section describes extending the techniques for constructor step cases to the full step case schema (4.1). The major differences are that meta-variables in the induction hypotheses can be instantiated by creational rippling or fertilisation, and that rippling-in is permitted.

4.4.1 Creational Rippling

The extra term structure that may appear in the hypotheses in non-constructor step cases has to be removed using creational rippling (see §3.4).

The initial step case schema is annotated as:

$$CC(\vec{x}), \quad \phi(\boxed{S_1(\vec{x})}, \ldots, \boxed{S_n(\vec{x})}), \quad IH(\vec{x}) \quad \vdash \quad \phi(\boxed{T_1(\vec{x})}, \ldots, \boxed{T_n(\vec{x})}) \tag{4.5}$$

Rippling takes place in the conclusion as before, with the wave annotation in the hypotheses being removed by creational rippling. Meta-variables in the hypothesis may be instantiated during a creational ripple by extending the definition of neutralisation (see 3.4): a wave-front $\boxed{f(\underline{\ldots})}^{\uparrow}$ corresponds to a potential wave-front $\boxed{A(\vec{x})}$ in an induction hypothesis iff they are in the same position in the skeleton. The latter term is instantiated by matching against the term $f(U)$, i.e. A is instantiated to $\lambda u. f(A'(u))$ for fresh A'.

A distinction between *speculative* and *definite* creational ripples is made in an analogous way to ripple steps. A speculative creational ripple instantiates a meta-variable, either by unification with the redex or matching during neutralisation. As such ripples introduce non-termination, they are included in the bound on the number of speculative steps.

4.4.2 Rippling-In

With wave-fronts in the induction hypotheses, rippling-in becomes a worthwhile strategy, as a wave-front can be rippled into a position where it neutralises a hypothesis wave-front. Sinks can be used to distinguish term positions in the conclusion which correspond to meta-variables in the induction hypotheses [Bundy et al., 1993], where rippling-in must always move a wave-front towards a sink.

4.4.3 Multiple Induction Hypotheses

As in the constructor case, additional hypotheses are provided by Protzen's heuristic (see §4.3.3 above). Because meta-variables are now allowed in the hypotheses, we can achieve this by simply adding and fertilising with a fresh schematic induction hypothesis, i.e. one with fresh meta-variables.

4.4.4 The Extended Strategy

The full strategy to create sink-free simple step cases is as follows:

1. Construct annotated schema for goal.
2. Creational ripple, or ripple-out, or ripple-in and -out.
3. Either:
   - Fertilise with known hypothesis.
   - Create schematic hypothesis and fertilise.
4. If goal is open, goto step 2, else collapse remaining meta-variables.

4.5 Summary

In this chapter a strategy for step case creation have been described in two parts. The first part deals with constructor style step cases. The second part generalised this strategy to include non-constructor style step cases as well. In Chapter 6 such step cases are used to construct a valid induction rule.

The main points of this chapter were:

- The step case schema is more general than [Kraan, 1994], and a larger class of step cases can be generated.
- The step case strategy is more general than the one described in [Protzen, 1995], although we take a related approach to generating multiple induction hypotheses using *Protzen's heuristic*.
- The induction conclusion and case conditions are created as a by-product of rippling.
- Induction hypotheses are created by creational rippling, and Protzen's heuristic.

- The number of speculative steps must be bounded to prevent rippling diverging. This can eliminate solutions from the search and cause incompleteness, a problem which is discussed further in Chapter 7.

Chapter 5

Synthesis of Case Structure

> *'Enumeration of cases' . . . is one of the duller forms of mathematical argument.*
>
> — G. H. HARDY, A MATHEMATICIAN'S APOLOGY

5.1 Introduction

Having used the techniques of Chapter 4 to generate a suitable step case, we are now faced with the problem of creating a valid induction rule that contains this step case. As part of this process, our strategy will need to create the other 'missing' cases of the induction rule. This chapter describes a suitable strategy for generating missing cases from a partial case analysis. It forms the second component of our induction strategy, which will be presented in full in Chapter 6.

In §5.2 we formalise the concept of missing cases in terms of *case formulae*, and in §5.3, restate the problem of finding such missing cases in terms of correcting faulty case formulae. §5.4 describes a strategy for patching case formulae based on known corrective techniques [Protzen, 1995, Monroy, 2000]. We give some examples of this proof strategy in §5.5, and describe some simple heuristics to improve its performance in §5.6.

5.2 Case Formulae

This section formalises the problem of generating missing cases of a partial case analysis. In the context of this thesis, this means generating missing cases of a partial induction rule. The following (partial) rules illustrate the problem:

$$\frac{\Phi(x) \quad \vdash \quad \Phi(s(s(x)))}{\vdash \forall x_{nat}.\, \Phi(x)} \tag{5.1}$$

$$\frac{x \neq 0, y \neq 0, \Phi(x), \Phi(y) \quad \vdash \quad \Phi(x+y)}{\vdash \forall x_{nat}.\, \Phi(x)} \tag{5.2}$$

Both rules are incomplete because they do not have premises that prove $\Phi(u)$ for $u = 0$ and $u = s(0)$.

The case structure of an induction rule is complete iff it forms an exhaustive case analysis. We can characterise this using a *case formula*.

Definition 5 (Case Formula) *Given a simple induction rule I of the form*

$$\frac{\begin{array}{ccc} C_1, H_1 & \vdash & \Phi(t_1^1, \ldots, t_n^1) \\ & \vdots & \\ C_k, H_k & \vdash & \Phi(t_1^k, \ldots, t_n^k) \end{array}}{\vdash \forall x_1 : \tau_1, \ldots, x_n : \tau_n.\, \Phi(x_1, \ldots, x_n)}$$

where C_i is a set of non-inductive hypotheses and H_i a set of inductive hypotheses, the corresponding **case formula** *$\mathcal{C}(I)$ is of the form*

$$\vdash \forall u : (\tau_1 \times \cdots \times \tau_n).\, (D_1 \vee \cdots \vee D_k) \tag{5.3}$$

where each disjunct D_i is of the form

$$\exists \mathcal{V}_i.\,(C_i \wedge u = (t^i_1,\ldots,t^i_n)) \tag{5.4}$$

and $\mathcal{V}_i$ is the set of free variables in C_i and $t^i_1,\ldots,t^i_n$. If $n = 1$ then the tuple of terms $t^i_1,\ldots,t^i_n$ can be written as a single term t^i_1. Also, C_i may be optionally omitted.

Informally, the case formula says that any u of the given type is ‘covered’ by at least one of the cases of the induction rule, where the disjuncts D_i correspond to the cases. Each disjunct says that u is covered by the corresponding case, in that u matches the pattern $(t^i_1,\ldots,t^i_n)$ under the conditions C_i.

For example, the partial induction rule (5.1) has the case formula

$$\forall u : nat.\,\exists x : nat.\,u = s(s(x))$$

A ‘complete version’ of rule (5.1) has the case formula

$$\forall u : nat.\,\big(u = 0 \,\vee\, u = s(0) \,\vee\, \exists x : nat.\,u = s(s(x))\big)$$

The partial induction rule (5.2) has the case formula

$$\forall u : nat.\,\exists x,y : nat.\,x \neq 0 \wedge y \neq 0 \wedge u = x + y \tag{5.5}$$

Whereas a ‘complete version’ of rule (5.2) has the case formula

$$\forall u : nat.\,\big(u = 0 \,\vee\, u = s(0) \,\vee\, \exists x,y : nat.\,x \neq 0 \wedge y \neq 0 \wedge u = x + y\big) \tag{5.6}$$

For these examples the truth of the complete case formulae is equivalent a corresponding induction rule having exhaustive case analysis. In general, we define case exhaustiveness for simple induction rules as follows:

Definition 6 (Case Exhaustive) *A simple induction rule is* case exhaustive *iff for any n-tuple $\vec{s}$ of type $\vec{\tau}$ there is a case of the rule with case conditions $A(\vec{x})$ and conclusion B such that*

1. *$A(\vec{s})$ holds.*
2. *$\sigma(B) = \Phi(\vec{s})$ for some substitution σ.*
3. *The rule has conclusion $\forall \vec{x}{:}\vec{\tau}.\Phi(\vec{x})$*

It follows that case formulae are equivalent to case exhaustiveness:

Theorem 1 *A simple induction rule I is case exhaustive iff the case formula $\mathcal{C}(I)$ is true.*

Proof Let $\vec{s}$ be some n-tuple of type $\vec{\tau}$. The case exhaustiveness of I is equivalent to one of the premises of Definition 5 satisfying requirements (1) and (2) from Definition 6, and the conclusion from Definition 5 satisfying (3). Equivalently, for this case the case conditions C_i hold and there is a substitution σ such that $\sigma((t^i_1, \ldots, t^i_n)) = \vec{s}$. This is equivalent to one of the disjuncts in (5.4) being true, and so to the truth of the case formula $\mathcal{C}(I)$.

Q.E.D.

5.3 Case Synthesis via Correcting Case Formulae

Theorem 1 lets us show that the cases of an induction rule are exhaustive, by proving its case formula. Conversely, we can establish it has missing cases by disproving the case formula.

This suggests a method for synthesizing the missing cases of an incomplete induction rule. Assume that its case formula

$$\forall u : \tau.(D_1 \vee \cdots \vee D_k) \tag{5.7}$$

is a faulty conjecture, and try to find a correct version of the form

$$\forall u : \tau.(D_1 \vee \cdots \vee D_k \vee D_{k+1} \vee \cdots \vee D_{k+j}) \tag{5.8}$$

Provided the additional disjuncts $D_{k+1}, \ldots, D_{k+j}$ are of the form given by (5.4), a set of additional cases can be extracted to form an exhaustive case analysis.

5.3.1 Corrective Techniques

The correction of faulty conjectures has been investigated in the context of inductive theorem proving [Protzen, 1995, Monroy, 2000]. Given a non-theorem $\forall \vec{x}.g(\vec{x})$, these methods attempt to build a *corrective predicate* p that specifies conditions under which the theorem is true, i.e. p such that $\forall \vec{x}.p(\vec{x}) \rightarrow g(\vec{x})$. A relationship between our proposed approach and this work can be seen by rewriting (5.8) as

$$\forall u : \tau.\big(\neg(D_{k+1} \vee \cdots \vee D_{k+j}) \rightarrow (D_1 \vee \cdots \vee D_k)\big) \tag{5.9}$$

Hence a possible approach to synthesizing cases would be to use a known corrective technique on faulty conjecture (5.7), and to transform the resulting corrective predicate term $p(u)$ to the form $\neg(D_{k+1} \vee \cdots \vee D_{k+j})$. However, we will take the alternative route of adapting the corrective techniques so that they construct the disjuncts D_i, as this is the more direct approach. The process for constructing the D_i is the same in both approaches.

The basic idea behind these corrective techniques is to attempt to prove the faulty conjecture, and to extract a definition for a corrective predicate p from the failed proof[1]. Each successful and failed proof branch gives rise to a case of the definition of p. Briefly, if a proof branch resulting from a case analysis succeeds then p is defined as $true$ under these case conditions, but if it fails then p is defined as $false$. Successful inductive proof branches which use an induction hypothesis give rise to recursive cases of p. We will not go into more detail here, but refer the interested reader to [Protzen, 1995] and [Monroy, 2000].

5.3.2 Problems with Existential Quantifiers

However, these corrective techniques were designed to be applied to conjectures containing only universal quantification, whereas case formulae contain existential quantification. In particular, they use standard induction proving techniques which do not deal with existential quantification (see Chapter 2) to construct a failed proof.

For example, consider an inductive proof of the faulty conjecture (5.5). Unless the existential quantifiers can be dealt with, the only option is to induct on u. Both the base and step cases are immediately blocked.

Clearly, if we are to successfully apply corrective techniques to case formulae, they need to be integrated with techniques for dealing with existential quantification. We deal with this problem in §5.4.2 below.

5.4 A Corrective Strategy for Case Formulae

This section describes a corrective strategy for case formulae that is based the corrective techniques discussed above, combined with the use of *dual skolemisation* to

[1] An advantage of this approach is that it can also be applied to theorems, where the proof can succeed.

handle existential variables (i.e. replacing them in a goal with first-order free variables), a standard technique in automated theorem proving. As suggested above, we will directly extract the missing disjuncts of the case formula from the failed proof, rather than construct a corrective predicate.

The proof attempt proceeds by a standard induction strategy, with *corrective disjuncts* extracted from failed proof branches. Subgoals of the form $\forall x : \tau . x = Y$ are trivially true, and are closed.

5.4.1 Extracting Corrective Disjuncts

For each proof branch, we record the *case conditions*: a pair (C,T) where C are any conditions introduced by case splits (including the cases of inductions) and T is the instantiation of the universally quantified variable of the original case formula (u in (5)).

For each failed branch of the proof we take its case conditions (C,T), and extract a corrective disjunct of the form

$$\exists \mathcal{V}. C \wedge u = T$$

where $\mathcal{V}$ are the free variables in C and T.

This technique of tagging each proof branch with its case information is used in both [Protzen, 1995] and [Monroy, 2000].

5.4.2 Instantiating Free Variables

Recall that dual skolemisation transforms the case formula's existential variables to free variables. This is permitted whilst proving the case formula, as any value substituted into a free variable can become a witness in the final proof. How does one treat these free variables during the proof? The standard approach is to let them become instantiated during rewriting. Unfortunately, this is not compatible with correcting the case formula.

For instance, consider the following correct dual skolemised case formula:

$$\forall u : nat. (u = 0 \lor (X \neq 0 \land u = X + Y) \tag{5.10}$$

X and Y are the free variables that have replaced existential variables. We can rewrite the case formula using the base case of the definition of $+$, instantiating X to 0 in the process:

$$\forall u : nat. (u = 0 \lor (0 \neq 0 \land u = Y) \tag{5.11}$$

The goal, which was previously true, can now be reduced to $false$. Following the corrective approach, we should analyse this failure to produce a corrective predicate. However, the original case formula does not need correcting.

What's going on here? Recall that the corrective approach attempts to identify those proof branches which are false. Reducing a goal to $false$ is interpreted as indicating the original goal is false under the current case conditions. This assumes that the current goal is *equivalent* to the original goal plus the case conditions. But this assumption is incorrect if we use *non-equivalence preserving steps*, where a true goal like (5.10) may have a false subgoal (5.11). If we combine non-equivalence preserving steps and corrective techniques then unnecessary corrections can be made, because true cases are identified as false.

Hence corrective techniques need to ensure that non-equivalence preserving steps are either excluded, or only permitted in successful branches of failed proofs, i.e. if a branch containing such steps fails, one should backtrack rather than correct the original conjecture. Unfortunately, whenever a free variable is instantiated to a term as a side effect of rewriting, e.g.

$$\frac{\Phi(0)}{\Phi(X)} \tag{5.12}$$

the step is non-equivalence preserving, so our failed proof branches will *always* contain such steps.

So we must instantiate free variables via an equivalence preserving step. For instance, an equivalence preserving version of (5.12) is:

$$\frac{\Phi(0) \vee \Phi(s(X'))}{\Phi(X)} \tag{5.13}$$

We can generalise this to a new proof step, the *existential case split*. Given an exhaustive case analysis represented by the following skolemised case formula

$$\forall u : \tau.\, ((\alpha_1(\vec{Y}) \wedge u = \beta_1(\vec{Y}))\ \vee \cdots \vee\ (\alpha_q(\vec{Y}) \wedge u = \beta_q(\vec{Y})) \tag{5.14}$$

Then an existential case split is represented by the following proof step

$$\frac{\begin{array}{c} c(\beta_1(\vec{Y})) \wedge \alpha_1(\vec{Y}) \wedge d(\bar{u}) = t(\beta_1(\vec{Y})) \\ \vee \cdots \vee \\ c(\beta_q(\vec{Y})) \wedge \alpha_q(\vec{Y}) \wedge d(\bar{u}) = t(\beta_q(\vec{Y})) \end{array}}{c(\vec{X}) \wedge d(\bar{u}) = t(\vec{X})} \tag{5.15}$$

where $\vec{X}$ has type τ.

The proof step (5.15) is applied backwards to a particular disjunct in a goal, with the case analysis (5.14) suggested by rewriting. For instance, if we can rewrite with a defining equation of function f, then we use the case analysis associated with the definition of f.

As an example of an existential case split, consider again the correct case formula (5.10):

$$\forall u : nat.\, (u = 0\ \vee\ (X \neq 0 \wedge u = X + Y) \tag{5.16}$$

The definition of $+$ has the following dual skolemised case formula

$$\forall u : nat.\, (u = 0 \ \vee \ u = s(V)) \tag{5.17}$$

This suggests an existential case split according to (5.15) with case analysis (5.17), which gives the subgoal

$$\forall u : nat.\, (u = 0 \ \vee \ (0 \neq 0 \wedge u = 0 + Y) \ \vee \ (s(X') \neq 0 \wedge u = s(X') + Y)) \tag{5.18}$$

Further rewriting gives

$$\forall u : nat.\, (u = 0 \ \vee \ u = s(X' + Y)) \tag{5.19}$$

A case split on u completes the proof, confirming that the case formula (5.16) is true.

5.5 Examples

This section gives some examples of our corrective strategy for case formulae being used to synthesize missing cases of induction rules. The case conditions of each goal are shown (i.e. the pair next to each goal).

For each of the proofs, only the correct derivation is shown, and any alternative steps at each point are ignored. These decisions are justified by a set of heuristics for the corrective strategy described in §5.6.

Example 1

Consider again the rule (5.1)

$$\frac{\Phi(x) \quad \vdash \quad \Phi(s(s(x)))}{\vdash \forall x_{nat}.\, \Phi(x)}$$

This has the faulty skolemised case formula

$$\vdash \quad \forall u : nat.\, u = s(s(X)) \qquad (true, u)$$

Attempting a proof, we try a structural case split on u

$$\vdash \quad 0 = s(s(X)) \qquad (true, 0)$$

$$\vdash \quad \forall v : nat.\, s(v) = s(s(X')) \qquad (true, s(v))$$

This simpiflies to

$$\vdash \quad false \qquad (true, 0)$$

$$\vdash \quad \forall v : nat.\, v = s(X') \qquad (true, s(v))$$

As the first case fails, we extract the corrective disjunct $u = 0$. Continuing with the second case, we apply another structural case split to v

$$\vdash \quad 0 = s(X') \qquad (true, s(0))$$

$$\vdash \quad \forall w : nat.\, s(w) = s(X'') \qquad (true, s(s(w))$$

Simplifying again gives

$$\vdash \quad false \qquad (true, s(0))$$

$$\vdash \quad \forall w : nat.\, w = X'' \qquad (true, s(s(w)))$$

The first case fails, and we extract the corrective disjunct $u = s(0)$. The second case is trivially true.

Adding the corrective disjuncts to the case formula, we obtain

$$\vdash \quad \forall u : nat.\, (u = 0 \ \vee \ u = s(0) \ \vee \ u = s(s(X)))$$

From this the missing cases of the induction rule can be constructed

$$\frac{\begin{array}{rcl} & \vdash & \Phi(0) \\ & \vdash & \Phi(s(0)) \\ \Phi(x) & \vdash & \Phi(s(s(x))) \end{array}}{\vdash \forall x_{nat}.\, \Phi(x)}$$

Example 2

Consider again the partial induction rule (5.2)

$$\frac{x \neq 0, y \neq 0, \Phi(x), \Phi(y) \quad \vdash \quad \Phi(x+y)}{\vdash \forall x_{nat}.\, \Phi(x)}$$

It has the skolemised case formula

$$\vdash \quad \forall u : nat. X \neq 0 \land Y \neq 0 \land u = X + Y \qquad (true, u)$$

We attempt to prove this faulty case formula. We proceed by a structural case analysis on u:

$$\begin{array}{ll} \vdash & X \neq 0 \land Y \neq 0 \land 0 = X + Y \qquad (true, 0) \\ \vdash & \forall v : nat. X' \neq 0 \land Y' \neq 0 \land s(v) = X' + Y' \qquad (true, s(v)) \end{array}$$

The definition of $+$ motivates a existential case split in both cases:

$$\begin{array}{ll} \vdash & 0 \neq 0 \land Y \neq 0 \land 0 = 0 + Y \\ & \lor\, s(Z) \neq 0 \land Y \neq 0 \land 0 = s(Z) + Y \qquad (true, 0) \\ \vdash & \forall v : nat. 0 \neq 0 \land Y \neq 0 \land s(v) = 0 + Y \\ & \lor\, s(Z') \neq 0 \land Y \neq 0 \land s(v) = s(Z') + Y \qquad (true, s(v)) \end{array}$$

Some simplification gives us the subgoals:

$$\vdash \quad false \lor false \qquad (true, 0)$$

$$\vdash \quad \forall v : nat.\, false \lor Y \neq 0 \land v = Z' + Y \qquad (true, s(v))$$

Hence the first case is false, and we extract the corrective disjunct $u = 0$ from it. Continuing with the second case we now perform a structural case split on v:

$$\vdash \quad Y \neq 0 \land 0 = Z' + Y \qquad (true, s(0))$$

$$\vdash \quad \forall w : nat. Y' \neq 0 \land s(w) = Z' + Y' \qquad (true, s(s(w)))$$

We could apply another existential case split in both cases, motivated by the definition of $+$, as we did above. This would lead to a non-terminating proof. To avoid this, we prefer a split motivated by any definition or lemma provided the split variable appears in the 'conditions' (i.e. C_i in (5.4)) and not just the 'main literal' (i.e. $u = (t^i_1, \ldots, t^i_n)$).

The definition-motivated split variable Z' only appears in the 'main literals' in both cases. But there is an alternative existential split, motivated by the following lemma:

$$U + s(V) \quad \Leftrightarrow \quad s(U + V) \tag{5.20}$$

In both cases the variable in the split motivated by (5.20) appear in the 'conditions' as well as the 'main literal', so we prefer this existential split:

$$\begin{aligned} \vdash \quad & 0 \neq 0 \land 0 = Z' + 0 \\ & \lor s(Q) \neq 0 \land 0 = Z' + s(Q) \qquad (true, s(0)) \\ \vdash \quad & \forall w : nat. 0 \neq 0 \land s(w) = Z' + 0 \\ & \lor s(Q') \neq 0 \land s(w) = Z' + s(Q') \qquad (true, s(s(w))) \end{aligned}$$

Simplification gives:

$$\vdash \quad false \lor false \qquad (true, s(0))$$

$$\vdash \quad \forall w : nat.\, false \lor w = Z' + Q' \qquad (true, s(s(w)))$$

Again, the first case is false, and we extract the corrective disjunct $u = s(0)$. The proof of the second case succeeds via a structural induction on w and existential case split in both base and step case, motivated by the definition of $+$. We omit the details here.

Correcting the original case formula gives

$$\forall u : nat. (u = 0 \ \vee \ u = s(0) \ \vee \ X \neq 0 \wedge Y \neq 0 \wedge u = X + Y)$$

Using this to construct the missing cases of the original induction rule, we obtain the following complete rule

$$\frac{\begin{array}{rcl} & \vdash & \Phi(0) \\ & \vdash & \Phi(s(0)) \\ x \neq 0, y \neq 0, \Phi(x), \Phi(y) & \vdash & \Phi(x+y) \end{array}}{\vdash \forall x_{nat}. \Phi(x)}$$

Example 3

Consider the following rule

$$\frac{\Phi(l) \quad \vdash \quad \Phi(app(l, x :: nil))}{\forall l : list(\alpha). \Phi(l)} \tag{5.21}$$

It has the case formula

$$\vdash \quad \forall u : list(\tau). u = app(X, Y :: nil) \qquad (true, u)$$

To attempt a proof, a structural induction on u is applied

$$\begin{array}{rcll} & \vdash & nil = app(X, Y :: nil) & (true, nil) \\ w = app(A, B :: nil) & \vdash & \boxed{v :: \underline{w}}^{\uparrow} = app(X', Y' :: nil) & (true, v :: w) \end{array}$$

In both goals, the definition of *app* suggests an existential case-split

$$\begin{aligned} \vdash\ & nil = app(nil, Y :: nil) \\ & \vee\, nil = app(H :: T, Y :: nil) \qquad (true, nil) \end{aligned}$$

$$\begin{aligned} w = app(A, B :: nil)\ \vdash\ & \boxed{\underline{\boxed{v :: w}^{\uparrow} = app(nil, Y' :: nil)}} \\ & \boxed{\vee\ \underline{\boxed{v :: \underline{w}}^{\uparrow} = app(\boxed{H' :: \underline{T'}}^{\uparrow}, Y' :: nil)}} \qquad (true, v :: w) \end{aligned}$$

Rewriting in the base case, and rippling in the step case, gives

$$\begin{aligned} \vdash\ & nil = Y :: nil\ \vee\ nil = H :: app(T, Y :: nil) \qquad (true, nil) \\ w = app(A, B :: nil)\ \vdash\ & \boxed{w = Y' :: nil\ \vee\ \underline{w = app(T', Y' :: nil)}}^{\uparrow} \qquad (true, v :: w) \end{aligned}$$

Further rewriting and fertilisation gives

$$\begin{aligned} \vdash\ & false \qquad\qquad (true, nil) \\ w = app(A, B :: nil)\ \vdash\ & w = nil \quad \vee \quad true \qquad (true, v :: w) \end{aligned}$$

Hence the base case fails and the step case succeeds. The corrective disjunct $u = nil$ is extracted from the base case's conditions $(true, nil)$. The corrected case formula is

$$\vdash\ \forall u : list(\tau).(u = nil\ \vee\ u = app(X, Y :: nil))$$

Using this to construct the missing cases of the original induction rule gives the complete rule

$$\frac{\begin{aligned} \vdash\ & \Phi(nil) \\ \Phi(l)\ \vdash\ & \Phi(app(l, x :: nil)) \end{aligned}}{\forall l : list(\alpha).\Phi(l)}$$

5.6 Heuristics for the Corrective Strategy

In §5.5 we saw that the corrective strategy for case formulae proceeded by an inductive proof strategy made up of the following proof steps:

- Structural induction
- Case analyses on universal variables, over the constructors for the datatype
- Case analyses on existential variables, over case structure of a function
- Simplification and rippling (forms of rewriting)
- Fertilisation

Examining the examples in §5.5, it is clear that a simple 'waterfall' of these proof steps is not being used. In fact, applying the steps in a fixed order can easily lead to non-termination. This section describes a set of simple heuristics that can help avoid this. It is important to note that these heuristics do *not* guarantee termination, although we have not encountered any problems with non-termination during the work described here.

Simplification, rippling and fertilisation should be applied eagerly, so that every goal is kept in the simplest form possible. This avoids unnecessary case-splits/inductions.

Structural induction and universal case-splits are effectively performing the same task — producing proof branches with different case conditions. If a universal case-split works, then an induction with the same case structure will always work, as the inductive hypotheses need not be used. Hence universal case splits are subsumed by induction, so we always use induction.

Induction (including case splits) is applied to a universal variable u in a disjunct $u = t$ so that rewriting can be applied. We have observed two situations in example proofs. Firstly, the root functor in t is a constructor function and rewriting immediately follows. Secondly, the roof functor in t is a defined function, and an existential casesplit follows, then rewriting. We can restrict induction on case formulae to these situations.

The rôle of existential case splits in these proofs is to instantiate a free variable X so that rewriting can take place. Again, we have identified two situations where this happens. Firstly, X appears in the t of a disjunct $u = t$, where u is a compound term containing no free variables, and t has a defined root functor. Secondly, X can appear in a disjunct not of the form $u = t$. Again, we can restrict the application of an existential casesplit to these situations.

Deciding between alternative existential case splits is done by looking where the variable to be split, say X, appears. Each disjunct will be of the form given by (5.4):

$$\exists \mathcal{V}.(C \wedge u = T)$$

for universal u. We prefer a split where X appears in C and possibly T over one where it only appears in T. The advantage of this was illustrated in §5.5. In general, splitting a variable in C will promote rewriting in C and hence its possible removal. This is desirable, as we would like to end up with a single disjunct $u = T$, because if we can make T variable the proof branch can be closed.

Another helpful addition to the strategy would be the use of valid case formulae as lemmas. Work could be saved by either recognising that:

- a goal matched a valid case formula, and was hence true.
- a goal partially matched a valid case formula, and so could be corrected by adding the disjuncts that were not matched against.

5.7 Summary

In this chapter the problem of automatically finding the missing cases of an induction rule has been addressed. It was shown that the concept of case formula characterises the case exhaustiveness of an induction rule. Hence the problem of finding missing cases can be restated as one of correcting the associated case formulae.

Applying known techniques on correcting faulty conjectures required some extensions to deal with existential variables. We used the standard technique of dual skolemisation, but found that instantiating the resulting free variables is a non-equivalence preserving step that interferes with the corrective methods. Instead, we proposed instantiating the free variables using an existential case analysis, an equivalence preserving step.

A strategy for correcting case formulae was given, based on standard inductive methods extended with existential case analyses and some simple heuristics. Following [Protzen, 1995] the case conditions of a failed proof branch are used to correct the faulty conjecture — in this context, to add extra disjuncts of the case formula, which correspond to the missing cases of the original induction rule.

Chapter 6

Induction Rule Creation

6.1 Introduction

So far this thesis has identified two major subtasks required for the dynamic construction of induction rules, and proposed a detailed solution for each. In Chapter 4 a middle-out strategy that generates candidate base and step cases was described. Chapter 5 provided a strategy for generating a full case analysis based on a given case, along with a proof that the cases are exhaustive. This chapter brings these parts together in a novel strategy for inductive proof. The strategy creates an induction rule dynamically during the proof, and provides a companion proof that this rule is valid.

§6.2 proposes that the validity of the rule is established by proving it is *well-founded* and *case exhaustive*. This allows us to give an induction strategy in §6.3 in terms of three component strategies: REFINE-CASE, EXHAUST-CASES and WELLFOUND-HYPS. The strategy is modular with respect to these components and §6.4 gives specifications which must be met by candidate components.

Candidates for the REFINE-CASE and EXHAUST-CASES components have been proposed in previous chapters. We end the chapter by describing a suitable WELLFOUNDHYPS strategy in §6.5.

6.2 Validating Induction Rules

This section looks at how our strategy can establish that the generated induction rule is valid, regardless of *how* it is generated. Recall that we only consider *simple induction rules* — it was argued in Chapter 3 that this is suitable class of induction rules for automated proof. This places some syntactic restrictions on the induction rule, although such rules are general enough to cover most, if not all, previous work on automated induction. So we may assume that the generated rule is of the form given in Definition 1 (see Chapter 3, p57).

It follows from the definition of simple induction rule that the generated rule will be of the following form:

$$\frac{\begin{array}{ccc} & & \vdots \\ C_1,\ \ldots, C_k,\ \theta_1(\forall \mathcal{Y}_1.\Phi),\ \ldots, \theta_h(\forall \mathcal{Y}_h.\Phi) & \vdash & \sigma(\Phi) \\ & & \vdots \end{array}}{\vdash \forall \mathcal{X}.\Phi} \tag{6.1}$$

where all the premises are of the given form, $h,k \geq 0$, $\mathcal{X}$ and $\mathcal{Y}_i$ are sets of variables, and θ_j and σ are substitutions.

Showing that the induction rule (6.1) is valid can be approached in a number of ways, e.g. by demonstrating that it is derivable from the Noetherian induction rule (see §2.2.1), or by directly proving that the consequent follows from the premises.

However, a more straightforward method of proof is to show that the rule is *well-founded* and *case exhaustive*. For the rule to be well-founded there must exist a well-founded relation $\prec$ under which the tuple of induction terms in every induction hypothesis is smaller than the tuple of induction terms in that case's conclusion. The rule is case exhaustive if its conclusion is proved for all values that the universal quantifiers could take. Equivalently, for all such values, there is a case that proves the conclusion for the values.

Following this proof method, our induction strategy explicitly constructs a validity proof, by stating and proving three types of goal:

Exhaustive Cases This goal is the *case formula* described in Chapter 5. (Details of how to construct and prove this goal are given there.) Proving it establishes that the rule's case analysis is exhaustive.

Well-Founded Hypothesis For each induction hypothesis, we must show that it is less than its conclusion under the relation $\prec$. Following the notation rule (6.1), we have free variables $\{x_1, \dots, x_n\}$ in formula Φ, and a step case of the form:

$$C, \dots, \theta(\forall \mathcal{Y}.\Phi), \dots \quad \vdash \quad \sigma(\Phi) \tag{6.2}$$

For this induction hypothesis the well-founded goal is:

$$\forall \mathcal{V}\big(C \rightarrow \quad (\theta(\beta(x_1)), \dots, \theta(\beta(x_n))) \prec (\sigma(x_1), \dots, \sigma(x_n))\big) \tag{6.3}$$

where β substitutes fresh variables for any $x_i \in \mathcal{Y}$, and $\mathcal{V}$ are the free variables in the goal.

Well-Founded Relation States that the relation $\prec$ is well-founded:

$$wellfound(\prec)$$

The approach adopted here of explicitly stating and proving validity requirements can be contrasted with [Protzen, 1995], which implicitly enforces these through restrictions on the generation of the induction rule. This is discussed further in Chapter 13.

Example

Consider the induction rule

$$\frac{\begin{array}{rl} & \vdash \;\; \Phi(0,0) \\ & \vdash \;\; \Phi(0,s(z)) \\ y \neq 0, \forall w{:}nat.\Phi(x,w) & \vdash \;\; \Phi(x+y,z) \end{array}}{\forall u,v{:}nat.\Phi(u,v)} \tag{6.4}$$

The validity goals for this rule are as follows:

Exhaustive cases Constructed using the case formula method from Chapter 5, the exhaustive cases goal:

$$\begin{array}{ll} \forall u,v{:}nat.\big(& (u,v) = (0,0) \quad \vee \\ & \exists z{:}nat.\ (u,v) = (0,s(z)) \quad \vee \\ & \exists x,y,z{:}nat.\ y \neq 0 \wedge (u,v) = (x+y,z) \quad \big) \end{array}$$

Well-Founded Hypotheses The step case matches (6.2) with the following values: $x_1 = u$, $x_2 = v$, $\mathcal{Y} = \{w\}$, $\theta = \{u/x\}$, $\sigma = \{u/(x+y), v/z\}$. Hence the well-founded hypothesis goal is:

$$\forall x,y,z,n{:}nat.y \neq 0 \rightarrow\ (x,n) \prec (x+y,z) \tag{6.5}$$

6.3 The Induction Strategy

We can now describe the components of our induction strategy, shown in Table 6.1. REFINE-CASE generates a case of the inductive proof by proving a schematic case, refining the schema as a side effect, as described in Chapter 4. The other two components — EXHAUST-CASES and WELLFOUND-HYPS — construct the validity proof of

Strategy	Proves Goal	Side Effect	Described In
REFINE-CASE	Schematic case	Instantiates schema	Chapter 4
EXHAUST-CASES	Exhaustive cases	New cases	Chapter 5
WELLFOUND-HYPS	Well-Found Hyp.	Constraints on $\prec$	§6.5

Table 6.1: Components of the induction strategy.

the induction rule. EXHAUST-CASES proves the 'exhaustive cases' goal, and generates any missing cases, as described in Chapter 5.

The WELLFOUND-HYPS component, which has not been described yet, proves the well-foundedness of the hypotheses and of the relation $\prec$ respectively. For each induction hypothesis WELLFOUND-HYPS generates a set of constraints on the relation $\prec$, such that the hypothesis is well-founded if these constraints are satisfied. The component also provides a constraint solver, which at the end of the inductive proof is used to pick a $\prec$ which satisfies these constraints.

Our induction strategy is given in Figure 6.1, described in terms of the component strategies. This strategy constructs a complete inductive proof of a conjecture, along with a validity proof for the induction rule[1]. Note that this thesis will only describe the implementation of a restricted version of this strategy, in Chapter 10.

The strategy searches for a step case first, rather than a base case. The justification for this is that step cases are nearly always harder to prove than the base cases. Tackling the 'hard part' first can avoid wasted effort on finding base cases, only to be unable to prove the step cases. The *Clam* system [van Harmelen, 1996] uses the same heuristic, attempting to prove step cases first.

Establishing the well-foundedness of the rule takes into account two competing requirements:

[1] Whether this is expressible in the object logic is another matter, discussed in §13.

```
IND-STRAT(GOAL):
    1. Construct an initial schematic step case S for GOAL, with case structure C.
    2. Construct a global constraint store STORE
    3. Apply CASE-STRAT to S.
    4. Apply NEW-CASES to C.
    5. Instantiate $\prec$ with the result of solving STORE.
NEW-CASES(CASES)
    1. Prove CASES exhautive via EXHAUST-CASES, possibly generating new cases NC.
    2. For each X in NC
        (a) Apply CASE-STRAT to X.
        (b) If X has been further refined then apply NEW-CASES to this sub-case.
CASE-STRAT(CASE):
    1. Construct a proof of CASE using REFINE-CASE.
    2. If CASE now contains any induction hypotheses then use WELLFOUND-HYPS to
       produce a proof CASE is well-founded given constraints T on $\prec$ are satisfied.
    3. Add T to STORE.
```

Figure 6.1: The dynamic induction strategy

- We want to eagerly apply WELLFOUND-HYPS in order to have some guarantee that every hypothesis is well-founded before we proceed with the proof. This means WELLFOUND-HYPS occurs early on in the proof.

- We want to delay choosing $\prec$ until after all the induction hypotheses have been generated, so that our choice is not incompatible with any hypotheses that help us to complete the proof. Hence $\prec$ is chosen at the end of the proof.

These requirements are reconciled by representing $\prec$ with a meta-variable and having

WELLFOUND-HYPS produce a proof of well-foundedness for each induction hypothesis that depends on a generated set of constraints on $\prec$ being satisfied, i.e. WELLFOUND-HYPS proves that a certain set of constraints implies the well-foundedness of each hypothesis. At the end of the entire proof the constraint solver provide by WELLFOUND-HYPS instantiates $\prec$ with a well-founded relation that satisfies these constraints.

6.4 Component Specifications

As mentioned above, an advantage of our strategy is its *modularity* with respect to its component strategies (see Table 6.1). This allows individual components to be replaced with alternatives, to yield a variety of inductive strategies — although in this thesis we only suggest a single candidate for each component. For example, this could be done in order to tailor the strategy to a particular domain. In this section we provide detailed specifications for the three components that must be met if the strategy is to work.

The strategy begins with an *initial goal*:

$$\forall x_1{:}\tau_1, \ldots, x_n{:}\tau_n. \quad \Phi(x_1, \ldots, x_n) \tag{6.6}$$

From this we can generate *case schemas*, described in Chapter 4, where meta-variables represent unknown parts of the goal.

6.4.1 REFINE-CASE Specification

The REFINE-CASE component must provide a proof of a *case schema* goal, refining the schema as a side-effect. Its specification is as follows:

Input Case Schema

$$C(\vec{x}),\ H(\vec{x}) \ \vdash\ \Phi(A_1(\vec{x}),\ldots,A_n(\vec{x})) \tag{6.7}$$

Output A partial instantiation of

- C with a conjunction of case conditions (non-inductive hypotheses)
- H with a list of simple induction hypotheses (see Definition 1)
- $A_1,\ldots,A_n$ with induction terms

and a proof of the instantiated case.

6.4.2 Exhaust-Cases Specification

The Exhaust-Cases component must take a set of known proof cases and generate a set of additional cases, such that the union of the two sets forms an exhaustive case analysis. Its specification is as follows:

Input Known proof cases

$$\begin{array}{rcl} C_1(\vec{x}),\ H_1(\vec{x}) & \vdash & \Phi(A_1^1(\vec{x}),\ldots,A_n^1(\vec{x})) \\ & \vdots & \\ C_m(\vec{x}),\ H_m(\vec{x}) & \vdash & \Phi(A_1^m(\vec{x}),\ldots,A_n^m(\vec{x})) \end{array}$$

Output Additional proof cases

$$\begin{array}{rcl} C_{m+1}(\vec{x}),\ H_{m+1}(\vec{x}) & \vdash & \Phi(A_1^{m+1}(\vec{x}),\ldots,A_n^{m+1}(\vec{x})) \\ & \vdots & \\ C_k(\vec{x}),\ H_k(\vec{x}) & \vdash & \Phi(A_1^k(\vec{x}),\ldots,A_n^k(\vec{x})) \end{array}$$

and a proof that these form an exhaustive case structure with the input cases, with partial instantiation of all C_i and A^i_j (with the same restrictions as given in REFINE-CASE). $H_{m+1},\ldots,H_k$ are uninstantiated meta-variables.

6.4.3 WELLFOUND-HYPS Specification

The WELLFOUND-HYPS component must take a *single* induction hypothesis and prove that it is less with respect to $\prec$ than the corresponding step case conclusion, under given conditions. Its specification is as follows:

Input A step case conclusion

$$\Phi(A_1(\vec{x}),\ldots,A_n(\vec{x}))$$

an inductive hypothesis

$$H(B_1(\vec{x}),\ldots,B_n(\vec{x}))$$

and a conjunction of case conditions $C(\vec{x})$.

Output A set S of constraints on $\prec$, and a proof that

$$S \wedge C(\vec{x}) \rightarrow \quad (B_1(\vec{x}),\ldots,B_n(\vec{x})) \quad \prec \quad (A_1(\vec{x}),\ldots,A_n(\vec{x})) \tag{6.8}$$

Also, a constraint solver which generates a wellfounded relation from a set of constraints, along with a proof that such a relation will be wellfounded.

6.5 Validating Hypotheses

The WELLFOUND-HYPS strategy is required to prove well-founded hypothesis goals (see (6.3) or (6.8)) of the form:

$$c \rightarrow \quad (b_1,\ldots,b_n) \quad \prec \quad (a_1,\ldots,a_n) \tag{6.9}$$

where $\prec$ is the unknown well-founded relation. We have decided to use a strategy that chooses $\prec$ by finding a *single* tuple argument position k and a measure function M such that for every goal (6.9):

$$c \rightarrow \quad M(b_k) < M(a_k)$$

There do exist induction rules which cannot be proved well-founded by considering only a single tuple argument i. Whether there is a need for a stronger well-foundedness strategy, and what that strategy would be, is an interesting question, which we leave for further research.

In §6.5.1 we explain how each application of the WELLFOUND-HYPS strategy generates constraints on $\prec$. The basis of the strategy is a simple adaptation of estimation [Walther, 1994b] to unary measure functions, described in §6.5.2 and §6.5.3. A further extension to estimation, required for non-destructor induction rules, is given in §6.5.4. We enhance the basic WELLFOUND-HYPS with an optional *side condition critic* (§6.5.5) that responds to the failure of estimation by adding extra step case conditions.

6.5.1 Constraints on $\prec$

Recall that after each step case proof, the WELLFOUND-HYPS strategy must prove that given some constraints on $\prec$ the resulting well-founded hypothesis goals (6.9) are satisfied (see §6.3), hence avoiding an early commitment to $\prec$. Our strategy does this by delaying commitment to the particular tuple argument position that will justify well-foundedness, although for each tuple argument it commits to a particular measure function from the very first induction hypothesis.

As the induction proof progresses, some tuple positions will become unusable, as for some induction hypothesis they did not reduce under the chosen measure. The

ith tuple position is identified as unusable by the constraint $\mathit{ignore}(i)$ posted to the $\prec$ constraint store.

For each well-founded hypothesis goal (6.9) the WELLFOUND-HYPS strategy produces a set of subgoals, such that for each tuple position i such that the constraint store *does not* contain $\mathit{ignore}(i)$, we have a subgoal:

$$c \rightarrow \quad M_i(b_i) \quad < \quad M_i(a_i) \tag{6.10}$$

for some measure function M_i (see below).

The behaviour of the strategy depends on whether this is the first well-founded hypothesis goal or not. As each successful application of WELLFOUND-HYPS adds constraints to the store, the strategy detects whether this is the first application by testing whether the constraint store is empty or not.

First Induction Hypothesis

For the first hypothesis, the measure function M_i in the goal (6.10) is represented by a fresh meta-variable. The goal is passed to the estimation strategy (see §6.5.2) below), which instantiates it to a measure function during the proof.

If the proof succeeds for the subgoals (6.10) corresponding to the tuple argument positions $p_1, \ldots, p_q$ then the following constraint is posted:

$$\mathit{measure}(p_1, M_{p_1)} \vee \cdots \vee \mathit{measure}(p_q, M_{p_q}) \tag{6.11}$$

For any tuple argument position p for which the corresponding subgoal (6.10) fails, the constraint $\mathit{ignore}(p)$ is posted. The proof fails if all the tuple positions are given *ignore* constraints.

Subsequent Induction Hypotheses

For subsequent induction hypotheses, the measure function M_i in the ith goal (6.10) is instantiated with the measure M from the disjunct $\mathit{measure}(i,M)$ from the constraint (6.11). The goal is passed to the estimation strategy (see §6.5.2 below). Again, for failed position p the constraint $\mathit{ignore}(p)$ is posted, and some positions must always remain *ignore*-free (or the proof fails).

Example (contd)

Consider again the well-founded hypothesis goal (6.5) from the first (and only) induction hypothesis in rule (6.4):

$$y \neq 0 \rightarrow \quad (x,n) \quad \prec \quad (x+y,z)$$

Given this the WELLFOUND-HYPS strategy will produce two subgoals:

$$y \neq 0 \rightarrow \quad M_1(x) \quad < \quad M_1(x+y) \tag{6.12}$$

$$y \neq 0 \rightarrow \quad M_2(n) \quad < \quad M_2(z) \tag{6.13}$$

These are passed to the estimation strategy. As we will see below, subgoal (6.12) succeeds with M_1 instantiated to the size measure $\#_{nat}$. The subgoal (6.13) fails.

Hence the WELLFOUND-HYPS strategy succeeds with the following constraints posted:

$$\mathit{measure}(1,\#_{nat})$$
$$\mathit{ignore}(2)$$

6.5.2 The Estimation Strategy

Our WELLFOUND-HYPS strategy is based on Walther's estimation calculus [Walther, 1994b] (see §2.9.1). It was chosen because it provides an automated method for well-foundedness proofs, and yet is simple enough to be recast into our framework, i.e. as a strategy to guide the construction of an explicit proof, and as part of our ‘constraint based’ wellfoundedness strategy.

Walther’s original calculus proves well-foundedness goals using a well-founded relation based on the *size measure function* — so there is effectively only one possible choice of $\prec$ per datatype [Walther, 1994b]. In [Giesl, 1995a] the calculus is adapted to work with polynomial norm measure functions, provided the measure is chosen beforehand. It may be possible to develop a WELLFOUND-HYPS strategy based on polynomial norms. However, we have chosen instead to use a strategy based on a different extension of the estimation method to arbitary *unary* measure functions M, which is given in §6.5.2 below.

The estimation calculus manipulates formulae of the form $\langle a \leq_M b,\, \Delta\rangle$, which are interpreted as follows:

$$\langle a \leq_M b,\, \Delta\rangle \quad \equiv \qquad a \leq_M b \,\wedge\, \big(\Delta \leftrightarrow a <_M b\big) \tag{6.14}$$

The calculus is used to prove such goals, which establish that some a is less than or equal to some b under a given measure M, and that there is some formula Δ that is equivalent to this bound being strict. Demonstrating welfoundedness is now a matter of showing Δ holds under the current conditions.

The *difference equivalent* Δ is unknown at the beginning of the proof. To prove a strict inequality $c \rightarrow a <_M b$ we apply the calculus to the goal $\langle a \leq_M b,\, \Delta\rangle$, where Δ is the unknown *difference equivalent* that will ensure the inequality is strict. We can use a meta-variable for Δ, which becomes instantiated to a formula during the estimation proof. If the proof succeeds, the strict inequality is established by proving $c \rightarrow \Delta$.

6.5.3 Upper Estimation

The basic operation of our estimation strategy is the application of the following rule:

Upper Estimation Rule For variables $\vec{x}$

$$\frac{\langle a_i \leq_M b,\, \Delta\rangle \qquad \langle f(\vec{x}) \leq_M x_i,\, \Delta^i_M f(\vec{x})\rangle}{\langle f(\vec{a}) \leq_M b,\, \Delta \vee \Delta^i_M f(\vec{a})\rangle} \tag{6.15}$$

The rule is applied backwards — the first premise becoming the new subgoal, whilst the second premise matches a known *argument bound lemma*[2] for f. Argument bounded properties of functions are automatically generated from their definitions before the proof, using the procedure from [Walther, 1994b]. The original calculus includes other rules to perform various trivial reasoning tasks — we simply pass these to a simple rewriting strategy.

This approach is much the same as Walther's original calculus, except an arbitary unary M is used, rather than the size measure. Our generalised rule is easily shown to be sound:

Theorem 2 (Soundness of Upper Estimation) *Rule is (6.15) is sound.*

Proof From the premises we know that

1. $a_i \leq_M b$
2. $\Delta \leftrightarrow a_i <_M b$
3. $f(\vec{a}) \leq_M a_i$
4. $\Delta^i_M f(\vec{a}) \leftrightarrow f(\vec{a}) <_M a_i$

[2] These are Boyer & Moore's *induction lemmata* [Boyer and Moore, 1979], the inspiration for Walther's calculus.

It follows that $f(\vec{a}) \leq_M b$, by (1) and (3). Also:

$$\begin{array}{rcll} \Delta \vee \Delta^i_M f(\vec{a}) & \leftrightarrow & a_i <_M b \vee f(\vec{a}) <_M a_i & \text{By (2) and (4)} \\ & \leftrightarrow & f(\vec{a}) <_M b & \text{By (1) and (3)} \end{array}$$

Hence $\langle f(\vec{a}) \leq_M b,\, \Delta \vee \Delta^i_M f(\vec{a})\rangle$.

Q.E.D.

6.5.4 Lower Estimation

A problem with using upper estimation is that it only works for inequalities of the form $F(x) <_M x$, i.e. the ‘lesser’ term is broken up until a copy of the ‘greater’ term is found. This is useful for showing destructor style induction rules are well-founded, as term structure only appears in the ‘lesser’ term. However, non-destructor inductions will generate well-foundedness goals with term structure in the ‘greater’ term, such as

$$x \neq 0 \rightarrow \quad x \quad <_M \quad x+y$$

The solution is to add a complementary form of estimation for the right-hand side of the equality [Gow et al., 1999]. We call this the *lower estimation* rule:

Lower Estimation Rule For all variable $\vec{x}$

$$\frac{\langle a \leq_M b_i,\, \Delta\rangle \qquad \langle x_i \leq_M f(\vec{x}),\, \Delta^i_M f(\vec{x})\rangle}{\langle a \leq_M f(\vec{b}),\, \Delta \vee \Delta^i_M f(\vec{b})\rangle}$$

The soundness proof is similar to Theorem 2. Similarly, lower argument bound lemmas (which match the second premise of our rule) can be generated automatically before the proof. See [Gow et al., 1999] for further details of lower estimation.

Example (contd)

Consider the goal (6.12) from above:

$$y \neq 0 \rightarrow \quad M_1(x) \quad < \quad M_1(x+y)$$

This is passed to the estimation strategy as:

$$\langle x \leq_{M_1} x+y,\ \Delta\rangle$$

where Δ is a fresh meta-variable. The proof depends on the following lemma, automatically generated from the definition of $+$ beforehand:

$$\langle u \leq_{\#_{nat}} u+v,\ v \neq 0\rangle$$

The lemma allows us to apply lower estimation, which instantiates M to $\#_{nat}$ and Δ to $(y \neq 0) \vee \Delta'$, giving:

$$\langle x \leq_{\#_{nat}} x,\ \Delta'\rangle$$

The goal is trivially discharged with $\Delta' = false$.

The estimation proof is completed by showing that the instantiated difference equivalent Δ follows from the side condition in (6.12):

$$y \neq 0 \quad \rightarrow \quad y \neq 0 \vee false$$

6.5.5 The Side Condition Critic

One way in which the estimation strategy can fail is when the non-strict inequality proof succeeds, but the strict inequality proof fails. This failure occurs because we cannot show that the difference equivalent (Δ in (6.14)) follows from the step case's side conditions.

To try and recover from this kind of failed proof, we use a critic to the estimation strategy which responds to the failure of the difference equivalent proof. It patches the proof by adopting the difference equivalent Δ as a side condition of the corresponding step case. Some simplification of Δ may be possible before we adopt it as a side condition.

Example

Consider the following step case, generated by our induction strategy:

$$\Phi(x) \quad \vdash \quad \Phi(x+y) \tag{6.16}$$

Applying the WELLFOUND-HYPS strategy we soon end up with the estimation subgoal:

$$\langle x \leq_{M_1} x+y,\ \Delta\rangle$$

As in the previous example, this is discharged with $M_1 = \#_{nat}$ and $\Delta = (y \neq 0 \vee false)$. To complete the proof we need to show:

$$true \rightarrow y \neq 0 \vee false$$

The proof fails, and the side condition critic responds by simplifying Δ to $y \neq 0$ and adding this as a side condition to the step case (6.16). The new well-founded step case is:

$$y \neq 0,\ \Phi(x) \quad \vdash \quad \Phi(x+y) \tag{6.17}$$

6.5.6 Choosing $\prec$

After the proof of an exhaustive set of base and step cases has been completed, a constraint solver supplied by WELLFOUND-HYPS is invoked. For the wellfoundedness strategy described above, the solver has a simple task: for the proof to have got this far, there must be at least one tuple argument position i such that:

- $\mathit{measure}(i, M_i)$ appears as a disjunct in the constraint (6.11).
- The constraint $\mathit{ignore}(i)$ does not appear.

The solver need only pick one such i, and instantiate $\prec$ to $\lambda x.\lambda y.(M_i(x) < M_i(y))$. It follows from the WELLFOUND-HYPS strategy that every induction hypothesis must be less than its conclusion under this relation.

The instantiated $\prec$ is also guaranteed to be well-founded, as any relation defined in terms of a measure function in this way is well-founded. (The proof of this is straightforward, and we omit it here.)

6.6 Summary

This chapter has described the induction rule creation strategy in terms of a number of distinct components. The strategy delays the choice of well-founded relation until the end of the proof, reducing the need for unnecessary search.

The strategy is modular with respect to the components, in that any strategy that satisfies the component's specification could be used — providing they are consistent in the constraints on $\prec$. We have suggested candidate strategies for all the components:

- The middle-out strategy of Chapter 4 can be used for REFINE-CASE.
- The case synthesis strategy of Chapter 5 can be used for EXHAUST-CASES.
- The estimation strategy of §6.5 can be used for WELLFOUND-HYPS.

Chapter 7

Controlling Speculation

7.1 Introduction

Speculative ripple steps, discussed in §4.3.1, are those which instantiate a meta-variable in the goal as a side-effect. Kraan noted in her work on middle-out induction selection that speculation caused rippling to be non-terminating (see [Kraan, 1994], also example in §2.7.1). Worse still, non-termination occurs in many simple examples, for both theorems and non-theorems. This chapter proposes a proof critic for controlling speculative ripple steps.

To ensure termination, Kraan's *Periwinkle* system places a fixed bound on the number of speculative steps. Unfortunately, a given theorem may require an arbitrary number of such steps for a middle-out strategy to find a proof, as we cannot put an *a priori* bound on the amount of 'induction term structure' required to prove a theorem, i.e. if we set a bound at 4 steps, there may be a solution only for 5 or more. Hence the bound excludes solutions from the search space.

In this section we propose a *speculation critic* that employs speculative rippling as a patch to overcome the failure of definite (i.e. non-speculative) rippling. This allows speculative rippling to be applied in a controlled way, and significantly reduces the risk of non-termination. The critic is based on the induction revision critic [Ireland and Bundy, 1996].

7.2 Divergent Speculation

The key to controlling speculation is identifying which speculative steps will progress the rippling proof. After the (compulsory) initial speculative step, which introduces a set of initial wave fronts, further speculative steps can only be useful if they help move the existing wave fronts.

Divergent Example

An example of useless speculation causing non-termination is given in Figure 7.1, from the example introduced in Chapter 1. (We abbreviate *foldleft_tr* to fld here.) Wave rule (7.4) is used repeatedly to speculate new wave fronts, which cannot be removed by further rippling. Each speculation contributes another blocked wave front to either side of the conclusion. However, the process will not stop because speculation is always possible, no matter how many blocked wave fronts accumulate. This speculation is useless, as it does not help unblock these wave fronts, and so cannot help the proof.

Such non-termination will occur in *any* schematic step case proof where blocked wave fronts arise that cannot be removed. This often happens during proof attempts of theorems because of a missing lemma, or during the proof of non-theorems.

$$\begin{array}{rcl}
fld(\circ, \fbox{$X(\vec{x})$}, \fbox{$L(\vec{x})$}) & = & y \circ fld(\circ, id, \fbox{$L(\vec{x})$}) \\
 & \Downarrow & (*) \\
fld(\circ, \boxed{\underline{\fbox{$X(\vec{x})$}} \circ L'(\vec{x})}^{\downarrow}, \fbox{$L''(\vec{x})$}) & = & y \circ fld(\circ, id, \boxed{L'(\vec{x}) :: \fbox{$L''(\vec{x})$}}^{\uparrow}) \\
 & \Downarrow & \\
fld(\circ, \boxed{\underline{\fbox{$X(\vec{x})$}} \circ L'(\vec{x})}^{\downarrow}, \fbox{$L''(\vec{x})$}) & = & y \circ fld(\circ, \boxed{\underline{id} \circ L'(\vec{x})}^{\downarrow}, \fbox{$L''(\vec{x})$}) \\
 & \Downarrow & (*) \\
fld(\circ, \boxed{(\underline{\fbox{$X(\vec{x})$}} \circ L'(\vec{x})) \circ L'''(\vec{x})}^{\downarrow}, \fbox{$L''''(\vec{x})$}) & = & y \circ fld(\circ, \boxed{\underline{id} \circ L'(\vec{x})}^{\downarrow}, \boxed{L'''(\vec{x}) :: \underline{\fbox{$L''''(\vec{x})$}}}^{\uparrow}) \\
 & \Downarrow & \\
fld(\circ, \boxed{(\underline{\fbox{$X(\vec{x})$}} \circ L'(\vec{x})) \circ L'''(\vec{x})}^{\downarrow}, \fbox{$L''''(\vec{x})$}) & = & y \circ fld(\circ, \boxed{(\underline{id} \circ L'(\vec{x})) \circ L'''(\vec{x})}^{\downarrow}, \fbox{$L''''(\vec{x})$}) \\
 & \Downarrow & \\
 & etc. &
\end{array}$$

Figure 7.1: Divergent speculation in the schematic step case proof for theorem $\forall x, y : \tau . \forall l{:}list(\tau) . fld(\circ, x, l) = y \circ fld(\circ, id, l)$, using the wave rules from Figure 7.2. Only the induction conclusion is shown. The speculative ripples steps (marked with asterisks) are motivated by wave rule (7.4).

Convergent Example

Now consider Figure 7.3, an example where further speculation is actually useful. The initial goal undergoes one speculative ripple with wave rule (7.1), followed by a definite ripple with (7.2). Both wave fronts are now blocked, but a further speculative ripple with (7.1) provides the extra wave front required to remove the wave fronts with (7.3) and fertilise, finishing the proof. This second speculative step unblocks the wave

$$\begin{aligned}
\boxed{s(\underline{X})}^{\uparrow} + Y &\Rightarrow \boxed{s(\underline{X+Y})}^{\uparrow} && (7.1)\\
X + \boxed{s(\underline{Y})}^{\uparrow} &\Rightarrow \boxed{s(\underline{X+Y})}^{\uparrow} && (7.2)\\
even(\boxed{s(s(\underline{X}))}^{\uparrow}) &\Rightarrow even(X) && (7.3)\\
fld(F,A,\boxed{H :: \underline{T}}^{\uparrow}) &\Rightarrow fld(F,\boxed{F(\underline{A},H)}^{\downarrow},T) && (7.4)
\end{aligned}$$

Figure 7.2: Wave rules used in the speculation examples.

fronts created by the first.

7.3 Ireland & Bundy's Induction Critic

To summarise the last section, after an initial speculative step, the resulting wave fronts may become blocked. Further speculative steps are only useful if they help ripple the existing wave fronts. We can view this in terms of fixing a proof failure (see §2.4.4): when definite rippling fails we can patch it with speculative rippling, which provides the missing wave fronts that allow rippling to continue.

This analysis shows that the problem of speculation is very similar to the situation described in [Ireland and Bundy, 1996], where rippling fails with a wave rule *partially matching* a goal — the wave rule requires some extra wave fronts that do not appear in the goal (see §2.5.5). Ireland and Bundy propose a proof critic which overcomes this failure by revising the induction rule. This is done by creating the necessary missing wave fronts and 'rewinding the proof' to see what induction rule could introduce them.

We can rationally reconstruct the patch from [Ireland and Bundy, 1996] as a four step process:

$$
\begin{array}{rcl}
even(C(\underline{x},y)+D(x,\underline{y})) & \leftrightarrow & even(D(x,\underline{y})+C(\underline{x},y)) \\
 & \Downarrow & (*) \\
even(\boxed{s(\underline{C'(\underline{x},y)+D(x,\underline{y})})}^{\uparrow}) & \leftrightarrow & even(D(x,\underline{y})+\boxed{s(\underline{C'(\underline{x},y)})}^{\uparrow}) \\
 & \Downarrow & \\
even(\boxed{s(\underline{C'(\underline{x},y)+D(x,\underline{y})})}^{\uparrow}) & \leftrightarrow & even(\boxed{s(\underline{D(x,\underline{y})+C'(\underline{x},y)})}^{\uparrow}) \\
 & \Downarrow & (*) \\
even(\boxed{s(s(\underline{C''(\underline{x},y)+D(x,\underline{y})}))}^{\uparrow}) & \leftrightarrow & even(\boxed{s(\underline{D(x,\underline{y})+\boxed{s(\underline{C''(\underline{x},y)})}^{\uparrow}})}^{\uparrow}) \\
 & \Downarrow & \\
even(\boxed{s(s(\underline{C''(\underline{x},y)+D(x,\underline{y})}))}^{\uparrow}) & \leftrightarrow & even(\boxed{s(s(\underline{D(x,\underline{y})+C''(\underline{x},y)}))}^{\uparrow}) \\
 & \Downarrow & \\
even(C''(\underline{x},y)+D(x,\underline{y})) & \leftrightarrow & even(D(x,\underline{y})+C''(\underline{x},y))
\end{array}
$$

Figure 7.3: Convergent speculation leads to a successful step case proof for the theorem $\forall x,y{:}nat.\, even(x+y) \leftrightarrow even(y+x)$, using the wave rules from Figure 7.2. Only the induction conclusion is shown. The speculative ripples steps (marked with asterisks) are motivated by wave rule (7.1).

Insert New Wave Fronts Insert the missing wave fronts and term structure into a copy of the failed goal — where each meta-variable is copied to itself — so that the partially matching wave rule could be applied. Erase the old wave fronts from the copy.

Reverse Ripple Reverse the direction of new wave fronts, so that outwards wave fronts are inwards and vice versa. Ripple the new wave fronts completely inwards using the rewrite relation $\Leftarrow$ instead of $\Rightarrow$. This takes us 'backwards' through the proof[1]. Sinks are used to indicate meta-variables, so that wave fronts may be rippled in towards them (see §2.5.1).

Change Induction Revise the original selection of induction rule so that these rippled-in wave fronts are actually introduced as induction terms by the induction rule. The proof critic selects a suitable rule from a prestored set. Update the partial proof to take account of the new induction.

Continue Proof Continue the proof from the patched goal, and discard the copy made in the first step.

Note that the goal produced by **Insert New Wave Fronts** may be a non-theorem. This is acceptable, because it is a purely meta-level goal that is used to determine a suitable instantiation, and will not appear in the final proof plan.

Convergent Example (contd)

As an example of Ireland and Bundy's induction critic, consider the example theorem from Figure 7.3. If we try ripple analysis on this goal, it suggests structural induction

[1] Although we are already doing backwards proof — trying to find a path from goal to axioms — so this 'backwards' step is actually forwards proof!

for *nat* (the dual induction for $+$). Using the wave rules from Figure 7.2, this proof fails as follows:

$$\begin{array}{c} even(\boxed{s(\underline{x})}^{\uparrow} + y) \;\leftrightarrow\; even(y + \boxed{s(\underline{x})}^{\uparrow}) \\ \Downarrow \\ even(\boxed{s(\underline{x+y})}^{\uparrow}) \;\leftrightarrow\; even(y + \boxed{s(\underline{x})}^{\uparrow}) \\ \Downarrow \\ even(\boxed{s(\underline{x+y})}^{\uparrow}) \;\leftrightarrow\; even(\boxed{s(\underline{y+x})}^{\uparrow}) \end{array}$$

Both wave fronts are now blocked.

However, the induction critic spots that wave rule (7.3) partially matches the left-hand wave front. Inserting the missing wave front into the goal we get:

$$even(\boxed{s(s(\underline{x+y}))}^{\uparrow}) \;\leftrightarrow\; even(\boxed{s(\underline{y+x})}^{\uparrow})$$

Notice that this goal is a non-theorem. This is acceptable, as we are only going to use this goal to determine a suitable instantiation — it will not appear in the final proof plan. The critic now erases the old wave-fronts, turns the new wave front inwards and reverse ripples:

$$\begin{array}{c} even(s(\boxed{s(\underline{x+y})}^{\downarrow})) \;\leftrightarrow\; even(s(y+x)) \\ \Downarrow \\ even(\boxed{s(\underline{x})}^{\downarrow} + y) \;\leftrightarrow\; even(s(y+x)) \end{array}$$

This suggests that the step case requires an additional $\boxed{s(\underline{x})}^{\uparrow}$ in the original induction term. Searching our set of known induction rules, we find that two-step induction on *nat* fufills this requirement, and we choose this induction rule instead.

The proof needs to be updated: an extra base case is required, and additional induction terms are introduced apart from the one we found via reverse rippling. The patched step case goal is:

$$even(\boxed{s(s(\underline{x+y}))}^{\uparrow}) \quad \leftrightarrow \quad even(\boxed{s(\underline{y+\boxed{s(\underline{x})}^{\uparrow}})}^{\uparrow})$$

7.4 A Speculation Critic

The analogy between the Ireland-Bundy critic and our approach is as follows: the new induction terms introduced in the **Change Induction** step are the equivalent of a useful speculative step which instantiates meta-variables, and so create/modify induction terms. This suggests that we can control speculation by adapting the induction revision critic.

We propose the following replacement for the **change induction** step: every wave front in the fully rippled in goal should surround a meta-variable. Providing no meta-variable has two occurrences surrounded by *different* wave fronts, then record each meta-variable/wave front pair $A(x_1,\ldots,x_n)/\boxed{F(\underline{\ldots})}^{\downarrow}$. Allow a speculative ripple only if it instantiates each A to $\lambda u_1.\ldots.\lambda u_n.F(A'(u_1,\ldots,u_n))$ for some fresh A'.

We will refer to the new critic as the *speculation critic* and the old critic as the *induction critic*. The essential difference between them is that in the induction critic the missing wave fronts suggest a particular choice of induction rule from a given set, whilst in the speculation critic they suggest a speculative ripple which will contribute towards creating a suitable induction rule.

Figure 7.4 shows a succient description of the critic in terms of preconditions and effects. The application of the speculation critic is illustrated in Figure 7.5.

Note that the speculation critic as defined here only works with constructor style induction, just as original induction critic did. It is compatible with non-constructor style schematic proofs, but will only suggest patches that correspond to constructor

Critic: Speculation Critic

Preconditions:

- The failed rippling goal G is not fertilisable.
- There is a wave rule R that partially matches G.
- Construct G', a copy of G — the goals now share meta-variables. In G':
    - Insert wave fronts $W_1, \ldots, W_n$ into G' so that R could be applied.
    - Erase any other wave fronts.
    - Turn $W_1, \ldots, W_n$ inwards.
    - Fully ripple-in $W_1, \ldots, W_n$ with backwards ripple steps $S_1, \ldots, S_m$.
    - Check no meta-variable in G' surrounded by different wave fronts.

Effects:

- Instantiate each meta-variable in G' surrounded by wave front — this will also instantiate meta-variables in G.
- Discard G' and its subgoals.
- Ripple out in G by applying normal ripple steps $S_m, \ldots, S_1$.
- Apply R to G and continue rippling.

Figure 7.4: Definition of the speculation critic

style step cases. We do not extend it to the destructor style in this thesis, although we hypothesise that this could be done (see §13.7).

Divergent Example (contd)

To get a clearer idea of how the new critic works, consider again the divergent example from Figure 7.1. After an initial speculative ripple, definite rippling becomes blocked

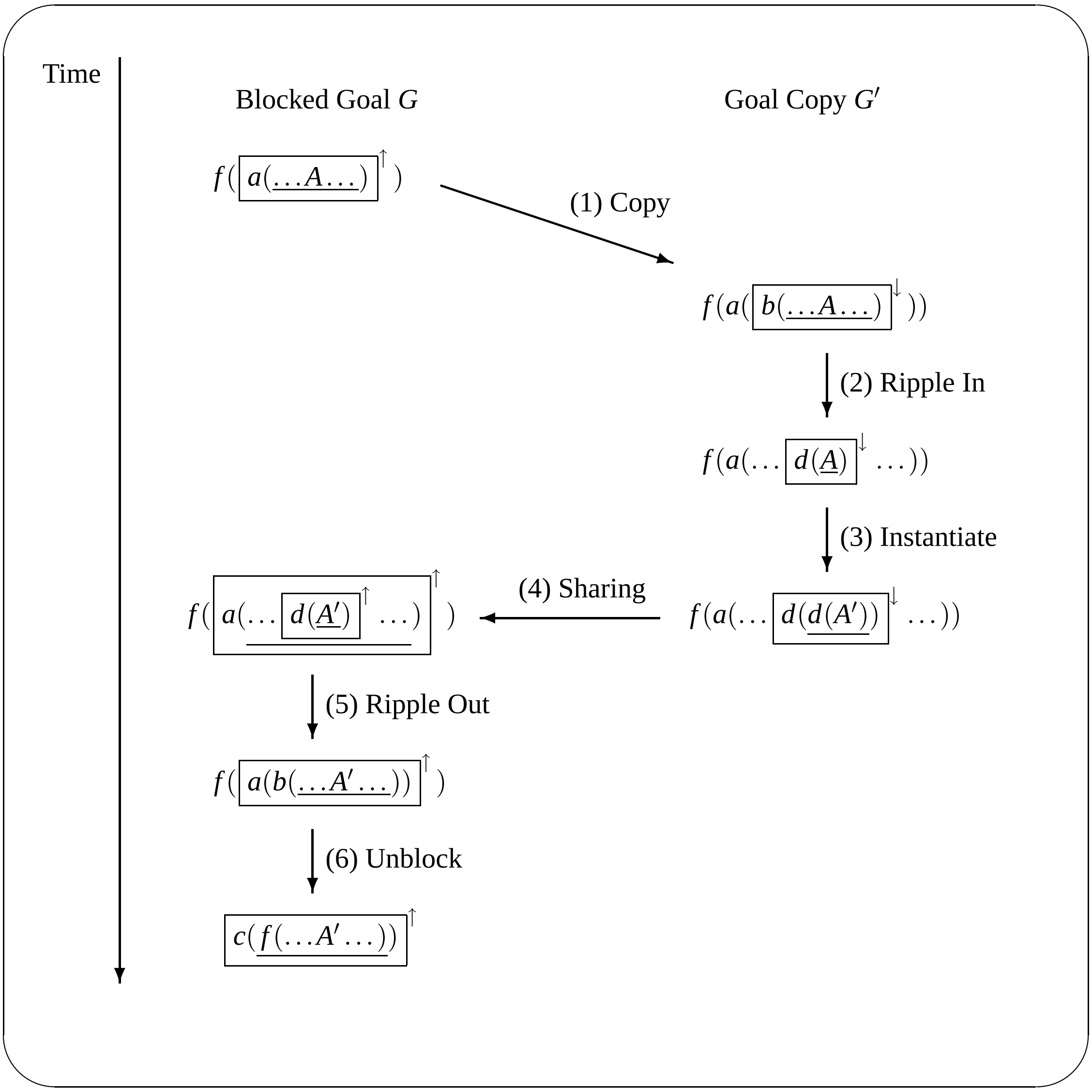

Figure 7.5: An application of the speculation critic: 1) the blocked goal is copied, with new inwards wave fronts 2) which are rippled in; 3) the fully rippled in wave fronts suggest an instantiation $A = d(A')$; 4) meta-variables are shared between both goals, producing new wave fronts in the blocked goal 5) which are rippled out; 6) the new wave fronts allow the goal to be unblocked.

with the following subgoal:

$$fld(\circ, \boxed{\underline{\bbox[border:1px dashed]{X(\vec{x})}} \circ L'(\vec{x})}^{\downarrow}, \bbox[border:1px dashed]{L''(\vec{x})}) \;=\; y \circ fld(\circ, \boxed{\underline{id} \circ L'(\vec{x})}^{\downarrow}, \bbox[border:1px dashed]{L''(\vec{x})})$$

There is no partially matching wave rule for this goal, and the critic is not applied. It cannot fix the ripple proof with further speculation. Hence the divergence illustrated in Figure 7.1 has been avoided.

Convergent Example (contd)

Now let us look at how the new critic handles the example of convergent speculation from Figure 7.3. After the initial speculative ripple, definite rippling becomes blocked with the following subgoal:

$$even(\boxed{s(\underline{\bbox[border:1px dashed]{C'(\underline{x},y)} + \bbox[border:1px dashed]{D(x,\underline{y})}})}^{\uparrow}) \;\leftrightarrow\; even(\boxed{s(\underline{\bbox[border:1px dashed]{D(x,\underline{y})} + \bbox[border:1px dashed]{C'(\underline{x},y)}})}^{\uparrow}) \tag{7.5}$$

The wave rule (7.3) partially matches this goal — it *would* match if we were to insert additional wave fronts into §7.5 as follows:

$$even(\boxed{s(s(\underline{\bbox[border:1px dashed]{C'(\underline{x},y)} + \bbox[border:1px dashed]{D(x,\underline{y})}}))}^{\uparrow}) \;\leftrightarrow\; even(\boxed{s(\underline{\bbox[border:1px dashed]{D(x,\underline{y})} + \bbox[border:1px dashed]{C'(\underline{x},y)}})}^{\uparrow})$$

Because definite rippling has failed with a partial wave rule match, we can invoke the speculation critic. It inserts the 'missing wave front' into the goal (7.5), then reverses its directions to give:

$$even(s(\boxed{s(\underline{\bbox[border:1px dashed]{C'(\underline{x},y)} + \bbox[border:1px dashed]{D(x,\underline{y})}})}^{\downarrow})) \;\leftrightarrow\; even(s(\bbox[border:1px dashed]{D(x,\underline{y})} + \bbox[border:1px dashed]{C'(\underline{x},y)}))$$

These wave fronts are now 'reverse' rippled-in. We have the following 'reverse' version of the wave rule (7.1) available:

$$\boxed{s(\underline{X+Y})}^{\downarrow} \;\Rightarrow\; \boxed{s(\underline{X})}^{\downarrow} + Y \tag{7.6}$$

Rippling-in with (7.6) on the LHS gives:

$$even(s(\boxed{s(\underline{C'(\underline{x},y)})}^{\downarrow} + D(x,\underline{y}))) \;\leftrightarrow\; even(\boxed{s(\underline{D(x,\underline{y}) + C'(\underline{x},y)})}^{\uparrow})$$

The new wave fronts have been fully rippled in. At this stage, the **change induction** step of the old induction critic would look for induction rules which introduced these wave fronts as induction terms. Instead, our new critic spots that one instances of C' is surrounded by $\boxed{s(\underline{\ldots})}^{\downarrow}$. Hence the patch to (7.5) is any speculative ripple that will instantiate C' to $\lambda u.\lambda v.s(C''(u,v))$.

This patch allows the second speculative ripple in Figure 7.3 to go ahead, and the proof to be completed.

7.5 Summary

This chapter looked at the problem of divergent speculation, and:

- Provided an analysis of why speculation may not terminate.
- Adapted Ireland and Bundy's induction critic in order to control speculation.

Chapter 8

Controlling Rewrite Search

8.1 Introduction

Our induction strategy relies heavily on rewriting to obtain a proof. This has the potential to introduce a large amount of search into our approach, and in this chapter we propose *position ordered rewriting* as a technique for reducing redundant search caused by backtracking during rewriting.

A great deal of research into theorem proving by rewriting has concentrated on eliminating the need for search by demonstrating that a set of rewrite rules is, or can be made, *confluent* [Baader and Nipkow, 1998]. Confluence is the property that alternate rewritings of a given term are always *joinable* — they can be rewritten to the same term. There is no need to backtrack over alternative rewritings when using a confluent system, as they all lead to the same result.

However, there are good reasons *not* to restrict a theorem prover to confluent rule sets. A non-confluent ruleset may be the most natural, or only, way to represent a particular problem. Alternative normal forms for a term may represent alternative approaches to solving a problem, and so could be useful to a theorem prover. In other words, there can be genuine choice points during rewriting. This is illustrated by the fact that rewrite rules may be based on non-equivalence preserving lemmas, where a true goal may give rise to a false subgoal. Confluence is not desirable here, as true and false should not be joinable!

Neither the *Clam* or λ*Clam* induction strategies, nor our induction strategy, assume confluent rewrite rule sets.

Overview

In this chapter we identify the problem of redundant search during non-confluent rewriting, and propose a technique for reducing it. §8.2 shows how redundancy arises when normal forms are rederived. It introduces the concept of confluent branches, along with sufficient conditions for identifying them.

In §8.3 we describe how position order rewriting can be used to block alternative paths to a term. The approach is formalised as π- and σ-rewriting. The question of completeness is addressed in §8.4, which gives a proof of the completeness of π-rewriting. Finally, in §8.5 we show the compatibility of the technique with meta-variables, and hence our induction strategy.

8.2 Redundancy in Rewriting

The redundant search caused by rederiving normal forms may be illustrated by the following simple example: consider the rule set $\{a \rightarrow b\}$ and the initial term $f(a,a)$. The term is normalised in two steps:

$$f(a,a) \rightarrow f(b,a) \rightarrow f(b,b)$$

If this normal form turns out to be unsatisfactory, we may backtrack over the first step, and find an alternative normalisation:

$$f(a,a) \to f(a,b) \to f(b,b)$$

The first normal form has been rederived, so this second derivation is redundant.

Similar examples have the potential to cause a combinatorial explosion, as parts of the term can be 'independently' rewritten, and may be in any order. This could produce a significant amount of redundant rewriting search. Another source of redundancy comes from the rôle rewriting plays within the theorem prover. A given normal form may be the input to another, potentially expensive, strategy. Such work will be duplicated if normal forms are rederived. Both these sources of redundancy may be arbitrarily large.

One solution would be to construct an explicit representation of the search space as an acyclic directed graph — assuming rewriting is terminating — and to extract the distinct normal forms from this. This completely avoids the problem of redundant search, but the graph may be infeasibly large, even for simple rewrite systems. Furthermore, constructing an explicit search space does not fit well into many reasoning frameworks, including the λ*Clam* proof planner. The technique we propose below does not have either of these drawbacks.

8.2.1 Confluent Branches

The problem of rederiving normal forms can be restated as follows: non-confluent rule sets may still exhibit 'locally confluent' behavior, in that *some* term may have alternative rewritings that are joinable. In general, there may be a term s which has a *confluent branch*:

$$\begin{tikzcd}
& s \arrow[dl] \arrow[dr] & \\
t_1 \arrow[dr, "*"] & & t_2 \arrow[dl, "*"] \\
& t &
\end{tikzcd}$$

As there are at least two paths to the ‘joining’ term t, it may be visited more than once. If there are more than two alternative reductions of s, then there may be more than two ways to get from s to t, or there may be multiple ‘joining’ terms t. However, we define a confluent branch as involving exactly two alternative reductions, so in these cases s is considered to have more than one confluent branch.

By considering confluent branches, we can formulate a principle for reducing search. At each term try to:

1. Identify confluent branches and

2. For each confluent branch block one of the paths to the joining term.

Below we show how this can be done in certain cases.

8.2.2 Identifying Confluent Branches

Consider a term s with two possible redexes: a at position p and b at position q. Subterm a may be reduced to a'. Subterm b may be reduced to b' by rewrite rule $l \to r$ with substitution σ. Without loss of generality, we may assume exactly one of three cases, shown in Figure 8.1. The case analysis is taken from the proof of the Critical Pair Lemma in [Baader and Nipkow, 1998]. For each case we will consider whether there is a confluent branch.

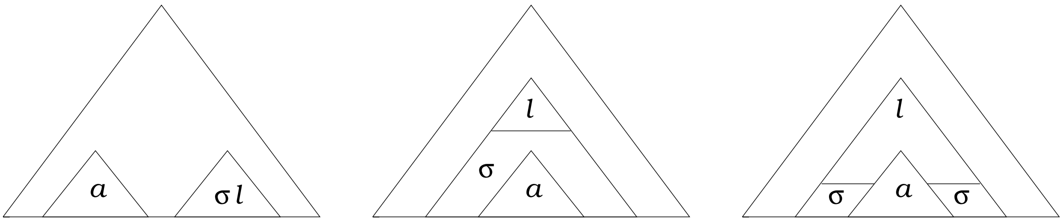

Figure 8.1: A term containing two redexes a and $b = \sigma(l)$. The redexes are either *non-overlapping* (left) and so form parallel subterms, or are *non-critically overlapping* (middle) where a is within the substitution σ, or *critically overlapping* (right) where a overlaps with the left-hand side l. ([Baader and Nipkow, 1998], pp136.)

Case 1: No overlap

In this case the two redexes a and b are parallel subterms of s, shown in the left term in Figure 8.1. The two reductions trivially form a confluent branch, illustrated in Figure 8.2. There are two paths: left (subterm a) then right (subterm b), or right then left.

Case 2: Non-critical overlap

Here one redex is the subterm of the other, but the inner redex a is entirely contained within the substitution σ of the outer reduction, i.e. it is within a subterm $c[a]$ that matches a variable X in l, the lefthand side the outer rewrite rule. This case is illustrated by the middle term of Figure 8.1.

A confluent branch is formed in this case. The first derivation begins with rewriting the outer term $b \longrightarrow b'$. The righthand side of the outer rewrite rule, r, will contain zero or more copies of X. Hence b' will contain subterm $c[a]$ at a set of positions P. Let us rewrite each of these to $c[a']$ with the inner rule, to give a final term equal to b' with

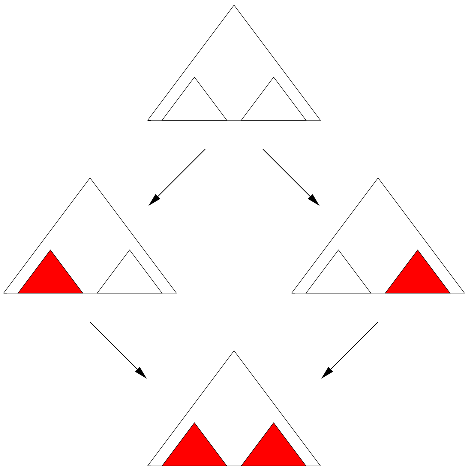

Figure 8.2: Non-overlapping redexes form a confluent branch.

the subterms at positions P replaced with $c[a']$.

The second derivation begins by rewriting each subterm that matched X in the first derivation from $c[a]$ to $c[a']$. Now outer rule $l \to r$ will still apply, but with an amended σ which replaces X with $c[a']$. Applying the rule, we have the term b' with the subterms at positions P replaced with $c[a']$ — the same term as before. Hence there are two alternate routes to the same term, and there is a confluent branch.

Case 3: Critical overlap

This case has one redex as a subterm of the other, shown in the right term of Figure 8.1. However, the inner term is not contained in a variable during the outer redex's reduction. In this case the branch may or may not be confluent — examples of both kinds are easily constructed. Because we cannot definitely identify whether a confluent branch exists, and if so what form it takes, we cannot apply the search reduction principle described above. Hence we ignore this case below.

8.3 Position Ordered Rewriting

To summarise the last section, if there is no overlap or a non-critical overlap between the two redexes — when the reductions are 'independent' of each other — then a confluent branch can be identified. Hence we have sufficient conditions for identifying a confluent branch. We ignore the case of a critical overlap between redexes, because a confluent branch may or not be present.

Given a confluent branch, the next step is to block one of the two paths to the joining term, as discussed in §8.2.1. Recall the possible reduction paths in the two 'independent' cases:

No overlap Left term then right, OR right then left.

Non-critical overlap Outer term then any copies of inner term created, OR all occurrences of inner term needed for outer rule to apply, then outer term.

The number of paths in each case can be cut down by imposing constraints on the reductions, based on the position of the redexes, which only allow a specific ordering of the independent steps. We will consider the following orders[1]:

Parallel Constraint Left cannot follow right.

Subterm Constraint Outer cannot follow inner.

We call this approach *position ordered rewriting*. More formally, we use the rewriting strategy given in Figure 8.3. The strategy prevents the alternate paths being taken

[1] It may be possible to develop techniques based on other orders.

1. The first redex may be chosen freely.
2. Subsequently, for last reduced subterm t and redex t':

   Parallel if t and t' are parallel then t must be to the left of t'.

   Subterm if t is below t' and rewrite rule $a \longrightarrow b$ reduces t' then for any unique (single occurrence) variable X in a, t must not be wholly contained within the subterm that matches X.

Figure 8.3: The position ordered rewriting strategy.

when we have confluent branches with independent redexes. Only the left-first/outer-first path should ‘get through’ to the joining term.

The subterm constraint in Figure 8.3 requires some explanation: it says that if we follow a reduction with another higher up the term, the result of the inner one cannot entirely be contained within a *unique* variable in the lefthand side of the outer rewrite rule, i.e. a variable that occurs only once. The next section works through some simple examples, and illustrates why this variable has to be unique.

Restricting rewriting in this way is obviously sound, but it is not obvious whether or not it is complete with respect to the original rewrite relation. §8.3.2 lays the groundwork for a proof of completeness, by giving a formal presentation of position ordered rewriting. In §8.4 we prove that using the parallel constraint alone gives a complete restriction.

8.3.1 Examples

This section illustrates position ordered rewriting with several examples.

The Parallel Constraint

Recall the motivating example mentioned at the beginning of §8.2: the single rewrite rule $a \longrightarrow b$ is applied to the term $f(a,a)$. Two derivations of the normal form $f(b,b)$ are possible. However, the parallel constraint blocks one of these paths, as follows (in each term the redex is underlined):

$$\begin{array}{ccccc} f(\underline{a},a) & \longrightarrow & f(b,\underline{a}) & \longrightarrow & f(b,b) \\ f(a,\underline{a}) & \longrightarrow & f(\underline{a},b) & \not\longrightarrow & f(b,b) \end{array}$$

This step is blocked as the redex is to the left of the last reduced subterm.

The Subterm Constraint

Now consider the two rule rewrite set $\{a \longrightarrow b,\ g(X) \longrightarrow h(X)\}$ applied to the term $g(a)$. The normal form $h(b)$ can be derived via two separate paths, one of which is blocked by the subterm constraint, as follows:

$$\begin{array}{ccccc} \underline{g(a)} & \longrightarrow & h(\underline{a}) & \longrightarrow & h(b) \\ g(\underline{a}) & \longrightarrow & \underline{g(b)} & \not\longrightarrow & h(b) \end{array}$$

This step is blocked as the last reduced subterm entirely within the subterm that matches the unique variable X when we match the lefthand side $g(X)$ to $g(b)$.

Why A Unique Variable?

This example shows why the subterm constraint specifies a *unique* variable, i.e. one with a single occurrence in the term. Allowing the constraint to work with an arbitrary variable would prevent redundant search in a greater number of cases. Unfortunately, this form of the subterm constraint is incomplete with respect to the original rewrite relation.

To see why, consider the two rule rewrite set $\{a \rightarrow b,\ f(X,X) \rightarrow g(X)\}$ applied to the term $f(a,b)$. If we use the non-unique version of the subterm constraint, then:

$$f(\underline{a},b) \quad \rightarrow \quad \underline{f(b,b)} \quad \rightarrow \quad g(b)$$

The step is blocked because the last reduced subterm is entirely within the subterm that matches the variable X when we match the lefthand side $f(X,X)$ to $f(b,b)$. There are no other derivations of $g(b)$, so the term has been pruned from the search space — hence the non-unique subterm constraint is incomplete.

It may be possible to design a complete position ordered rewriting strategy that blocks a step when the last reduced subterm is entirely within a subterm that matches *any* variable. We speculate that this would involve taking into account the positions of several previous rewrite steps, rather than just the last reduced subterm.

8.3.2 π- and σ-Rewriting

In order to formalise position ordered rewriting, we introduce notation for some formal rewriting concepts. Where possible, we have followed the standard notation of [Baader and Nipkow, 1998].

Positions A position p of a subterm s of a term t is the list of positive integers that determines a path from the root of t to the root of s. pq is the list p appended to the list q. ε is the empty (root) position. $posn(s,t)$ returns the set of positions of subterm s within t.

Above/Below The order $<$ is defined on positions as $p < q$ iff there exists $p' \neq \varepsilon$ such that $q = pp'$, i.e. when p is above q in the term tree. $>$, $\leq$ and $\geq$ are defined in the obvious manner.

Parallel If $p \neq q$, $p \not< q$ and $p \not> q$ then p is parallel to q, written $p \,||\, q$. Note that $p = q$, $p < q$, $p > q$ and $p \,||\, q$ are mutually exclusive and exhaustive cases.

Before/After The order $\prec$ is defined as the lexicographic ordering on positions. This means $p \prec q$ iff p comes before q in a depth-first traversal of the term tree. $\succ$, $\preceq$ and $\succeq$ are defined in the obvious manner.

Rewrite Step For position p and rewrite rule r, the rewrite step $\gamma = [p,r]$ is the transformation $t \rightarrow_r^p t'$. We use the functional notation $\gamma : t \mapsto t'$ and $\gamma(t) = t'$.

Sequence A sequence of rewrite steps $\phi = [p_1, r_1], \ldots, [p_n, r_n]$ is applied to a term by iteratively applying the steps to the term in the given order. As with rewrite steps, we use the functional notation $\phi : t \mapsto t'$ and $\phi(t) = t'$. ε is the empty sequence.

Definition 7 ($\pi\sigma$-sequence) *A sequence of rewrite steps is a* $\pi\sigma$**-sequence** *iff for any two consecutive rewrite steps $[p,r]$ and $[q,s]$, where $s = (a \rightarrow b)$*

(π) *if $p \,||\, q$ then $p \preceq q$*

(σ) *if $p > q$ and variable $a|_u$ occurs only once in a then $p \not\geq qu$*

Enforcing these constraints on rewriting is called $\pi\sigma$**-rewriting**. If only the (π) constraint is enforced a rewrite sequence is called a π-sequence, and restricting rewriting in this way is called π-rewriting. σ-sequence and σ-rewriting are defined analogously.

8.4 Completeness

Having formalised position ordered rewriting we can now consider its completeness. By completeness we mean that using $\pi\sigma$-rewriting does not prevent any terms being derived. More formally, if $t \rightarrow^{*} t'$ then there exists a $\pi\sigma$-sequence ϕ such that $\phi(t) = t'$. In other words, any term that has a particular path to it blocked by the $\pi\sigma$ restrictions can be reached by some other acceptable path.

This is not the only form of completeness that could be considered. For example, the completeness with respect to *normal forms* [Baader and Nipkow, 1998], i.e. that exhaustively applying $\pi\sigma$-rewriting is equivalent to normal rewriting. However, we have not found such results any simpler to prove than our stronger notion of completeness given above.

In this section we prove that π-rewriting is complete, and discuss the possible completeness of σ- and $\pi\sigma$-rewriting. First we provide some additional concepts that will simplify our proofs:

Composition We write $\phi_1\phi_2$ to denote the sequence obtained by applying ϕ_1 then ϕ_2.

Equivalence Two sequences ϕ_1, ϕ_2 are equivalent (written $\phi_1 \equiv \phi_2$) iff $\phi_1(t) = \phi_2(t)$ for any term t.

Length $|\phi|$ is the number of steps in ϕ.

Segment A sequence ϕ' is called a segment of a sequence ϕ if $\phi = \phi_A\phi'\phi_B$ for some ϕ_A, ϕ_B.

8.4.1 π-Rewriting is Complete

In this section we prove the completeness of π-rewriting: for any sequence there exists an equivalent π-sequence. Our proof treats the given sequence as a ‘broken’ π-sequence which can be ‘fixed’.

The proof requires three lemmas, the first of which is trivial.

Lemma 1 (Segment Lemma) *Any segment of a π-sequence, σ-sequence or $\pi\sigma$-sequence is also a sequence of this type.*

The following simple lemma shows that swapping the order of the steps 'fixes' a 'broken' π-sequence of length 2.

Lemma 2 (π-Swap Lemma) *For rewrite steps γ_1 and γ_2, if $\gamma_1\gamma_2$ is not a π-sequence then $\gamma_2\gamma_1$ is, and $\gamma_2\gamma_1 \equiv \gamma_1\gamma_2$*

Proof $\gamma_1\gamma_2 = [p,r][q,s]$ is not a π-sequence, so $p \mid\mid q$ and $p \succ q$. Hence $q \mid\mid p$ and $q \preceq p$, so $[q,s][p,r]$ is a π-sequence. Also $[q,s][p,r] \equiv [p,r][q,s]$ because $p \mid\mid q$.

Q.E.D.

We now introduce *k-broken π-sequences* where the kth step breaks the π-constraint, and removing it gives a valid π-sequence. The definition is followed by a lemma which shows that we can always fix k-broken π-sequences. This definition and lemma are motivated by the step case of the inductive completeness proof which follows, where a k-broken π-sequence arises and is fixed.

Definition 8 (k-Broken π-Sequence) *A sequence ϕ is a k-*broken π-sequence *for $k \geq 2$ iff there is a rewrite step γ such that $\phi = \phi_A\gamma\phi_B$ and $|\phi_A\gamma| = k$ for some sequences ϕ_A, ϕ_B, and that:*

1. *ϕ_A is a π-sequence.*

2. *$\phi_A\phi_B$ is a π-sequence.*

3. *$\phi_A\gamma$ is* **not** *a π-sequence.*

Lemma 3 (π-Fix Lemma) *For any k-broken π-sequence there exists an equivalent π-sequence.*

Proof Induction on k, the position of the broken rewrite step.

Case $n = 2$. Let $\phi = \gamma_1\gamma_2\phi_B$ be a 2-broken π-sequence. Following the definition:

A1. γ_1 is a π-sequence, which is trivially true anyway.

A2. $\gamma_1\phi_B$ is a π-sequence.

A3. $\gamma_1\gamma_2$ is not a π-sequence.

By (A3) and the π-swap lemma $\gamma_2\gamma_1$ is a π-sequence and $\gamma_1\gamma_2 \equiv \gamma_2\gamma_1$. Therefore by (A2) $\gamma_2\gamma_1\phi_B$ is a π-sequence equivalent to ϕ.

Step Case. Assume that we can fix any k-broken π-sequence. Consider a $(k+1)$-broken π-sequence ϕ. As ϕ must have at least three steps, we may write $\phi = \phi_A\gamma_1\gamma_2\phi_B$ where $|\phi_A\gamma_1\gamma_2| = k+1$. It follows from the definition that:

B1. $\phi_A\gamma_1$ is a π-sequence.

B2. $\phi_A\gamma_1\phi_B$ is a π-sequence.

B3. $\phi_A\gamma_1\gamma_2$ is not a π-sequence.

By (B1) and (B3) $\gamma_1\gamma_2$ is not a π-sequence. Hence $\gamma_2\gamma_1 \equiv \gamma_1\gamma_2$ by the π-swap lemma. Define $\phi' = \phi_A\gamma_2\gamma_1\phi_B \equiv \phi$. Now if ϕ' is a π-sequence then we are done, so let us assume it is not. Therefore:

C1. ϕ_A is a π-sequence, by (B1) and the π-segment lemma.

C2. $\phi_A\gamma_1\phi_B$ is a π-sequence, which is (B2).

C3. $\phi_A\gamma_2$ is not a π-sequence. If it were, $\phi_A\gamma_2\gamma_1\phi_B = \phi'$ would be by (B2) and the π-segment lemma, a contradiction.

Given that $|\phi_A\gamma_2\gamma_1| = k+1$ then $|\phi_A\gamma_2| = k$, so ϕ' is a k-broken π-sequence. Hence by the inductive hypothesis there exists a π-sequence $\phi'' \equiv \phi' \equiv \phi$.

Q.E.D.

Theorem 3 (π-rewriting is complete) *For any sequence there exists an equivalent π-sequence.*

Proof By induction on n, the length of the sequence. The cases $n = 0$ and $n = 1$ are trivial. The case $n = 2$ was shown by the π-swap lemma.

Step Case. Assume the $n = k$ case, and consider a sequence ϕ of length $k+1$. Let $\phi = \phi_A\gamma$. By the inductive hypothesis, there exists a π-sequence $\phi'_A \equiv \phi_A$. Now define $\phi' = \phi'_A\gamma \equiv \phi$. If ϕ' is a π-sequence we are done, so let us assume it is not. Observe that ϕ' must now be a $(|\phi'_A|+1)$-broken π-sequence, so by the π-fix lemma there exists a π-sequence $\phi'' \equiv \phi' \equiv \phi$.

Q.E.D.

Note that the equivalent π-sequences are constructed by reordering the original sequences, so we can conclude that they are of the same length as the originals. Hence π-rewriting will not cause inefficiency by eliminating the shortest path to a term — a π-sequence of equal length will exist.

8.4.2 Towards $\pi\sigma$-Completeness

In this section we provide a swap lemma for σ-rewriting, as part of an attempted proof of the completeness of σ-rewriting. In the π case lemma 2 put the ‘left’ rewrite step before the ‘right’ step. In the σ case lemma 4 puts the ‘higher’ step before the ‘lower’ step. As a result, the ‘lower’ redex may change position and be duplicated. This makes the statement of the lemma more complex than in the π case.

However, we have so far been unable to prove the completeness of σ-rewriting by the route of a corresponding σ-Fix lemma. We leave its completeness, and the completeness of πσ-rewriting, as open conjectures.

Lemma 4 (σ-Swap Lemma) *If $[p,r][q,s]$ is not a σ-sequence then there exists an equivalent σ-sequence $[q,s]\overset{m}{\underset{i=1}{\Phi}}[qu_iv,r]$ for certain positions $u_1,\ldots,u_m,v$.*

Proof If the (σ) constraint is broken, then by definition $p > q$ and for $s = (a \longrightarrow b)$ there is some variable $x = a|_u$ such that $p \geq qu$. Let $posn(x, rhs(s)) = \{u_1,\ldots,u_m\}$. Without loss of generality, assume that $i < j \Rightarrow u_i \prec u_j$. The u_i are the positions of a variable, and so must be mutually parallel. Also, as we know $p > q$, let v be such that $p = quv$.

First, we show that the given sequence is a σ-sequence: as the u_i are mutually parallel, so must qu_iv for $i \in [1,m]$. Furthermore, $i < j \Rightarrow qu_iv \prec qu_jv$, so $\overset{m}{\underset{i=1}{\Phi}}[qu_iv,r]$ is a σ-sequence. Now, as $q \not\geq qu_1v$, $[q,s]\overset{m}{\underset{i=1}{\Phi}}[qu_iv,r]$ is also a σ-sequence.

Next, we show that the given sequence is equivalent to $[p,r][q,s]$: Suppose the original sequence was applied to a term t_0, such that $[p,r] : t_0 \mapsto t_1$ and $[q,s] : t_1 \mapsto t_2$.

t_1 is the result of one step and the input to another, so it follows that there are terms a, b, c and d for which $[\varepsilon,r] : a \longrightarrow b$, $[\varepsilon,s] : c \longrightarrow d$ and $t_0|_p = a$, $t_1|_p = b$, $t_1|_q = c$ and $t_2|_q = d$.

Let $t \downarrow_p [t']$ denote a term t with a subterm t' at p. We know that $p = quv$ and that the subterm $t_1|_{qu}$ matches the variable x and is copied by $[q,s]$ to positions $qu_1,\ldots,qu_m$, so we may write

$$\begin{aligned}
t_0 &= t_0 \downarrow_q [c \downarrow_u [e \downarrow_v [a]]] \\
t_1 &= t_0 \downarrow_q [c \downarrow_u [e \downarrow_v [b]]] \\
t_2 &= t_0 \downarrow_q [d \downarrow_{\{u_1,\ldots,u_m\}} [e \downarrow_v [b]]]
\end{aligned}$$

where $e = t_0|_{qu}$. Now

$$\begin{aligned} [q,s](t_0) &= [q,s](t_0 \downarrow_q [c \downarrow_u [e \downarrow_v [a]]]) \\ &= t_0 \downarrow_q [d \downarrow_{\{u_1,\ldots,u_m\}} [e \downarrow_v [a]]] \end{aligned}$$

and

$$\begin{aligned} \overset{m}{\underset{i=1}{\Phi}}[qu_iv,r](t_0 \downarrow_q [d \downarrow_{\{u_1,\ldots,u_m\}} [e \downarrow_v [a]]]) &= t_0 \downarrow_q \left[\overset{m}{\underset{i=1}{\Phi}}[u_iv,r](d \downarrow_{\{u_1,\ldots,u_m\}} [e \downarrow_v [a]])\right] \\ &= t_0 \downarrow_q [d \downarrow_{\{u_1,\ldots,u_m\}} [[v,r](e \downarrow_v [a])]] \\ &= t_0 \downarrow_q [d \downarrow_{\{u_1,\ldots,u_m\}} [e \downarrow_v [b]]] \\ &= t_2 \end{aligned}$$

Hence $[q,s]\overset{m}{\underset{i=1}{\Phi}} : t_0 \mapsto t_2$ and so is equivalent to $[p,r][q,s]$.

Q.E.D.

8.5 Compatibility with Meta-variables

As it stands, position ordered rewriting is incompatible with terms containing meta-variables, as it can be shown to be incomplete. The following example[2] illustrates the problem. Consider the rewrite system:

$$\begin{aligned} a &\rightarrow b \\ g(a) &\rightarrow c \end{aligned}$$

Following the use of meta-variables throughout this thesis, we let rewriting instantiate them on the condition that the redex is never meta-flexible, i.e. so we cannot just endlessly rewrite a meta-variable subterm.

[2] Alan Smaill, private communication.

Taking the initial term $p(X,g(X))$ we can uniquely derive the normal form $p(b,c)$ via the following derivation (in each term the redex is underlined):

$$p(X,\underline{g(X)}) \quad \rightarrow \quad p(\underline{a},c) \quad \rightarrow \quad p(b,c)$$

However, this is not a π-sequence, and so is blocked by position ordered rewriting. This is because the second redex is to the left of the first redex (i.e. $2 \mid\mid 1$ and $2 \prec 1$), which is disallowed by the definition of π-rewriting. Because there is no other way to derive $p(b,c)$ from the initial term, a normal form has been excluded from the search space.

The general problem is that a rewrite step may instantiate a meta-variable that has other occurrences in parts of the term in which πσ-rewriting disallows rewriting. A solution to this problem is to treat meta-substitutions as unrestricted rewrite steps. That is, instantiating a meta-variable at position p is considered as a rewriting at p. For the example above:

$$p(\underline{X},\underline{g(X)}) \quad \rightarrow \quad p(\underline{a},c) \quad \rightarrow \quad p(b,c)$$

Note that the first term is now reduced at two positions simultaneously. This is ok, as we can regard the leftmost/highest position (the smallest by $\prec$) as the ‘real’ position. By this definition the above is now a πσ-sequence.

This approach overcomes the known problems with using position ordered rewriting with our induction strategy, and other techniques based on meta-variables. Based on this we conjecture that the technique is complete in the presence of meta-variables.

8.6 Summary

This chapter examined the problem of redundant search during non-confluent rewriting, and:

- Provided an analysis of redundant rewriting in non-confluent systems based on *confluent branches*.
- Introduced position ordered rewriting as an approach to reducing redundancy, and formalised it as $\pi\sigma$-rewriting.
- Proved the completeness of π-rewriting.
- Showed how it can be made compatible with middle-out reasoning.

Chapter 9

A Proof Planner with Critics

9.1 Introduction

Having laid out the various components of our induction strategy in Chapters 4 to 8, we now consider its implementation. The λ*Clam* proof planner was chosen, and extended with proof critics, for this purpose.

After explaining in §9.2 why an extended λ*Clam* was used, the rest of the chapter describes the novel critics-based proof planning architecture implemented in the system. The main features of the architecture are *planning instructions* (see §9.4) to allow more flexible specification of when critics should be applied , and *criticals* (see §9.5) which can be used to specify critic strategies in a analogous manner to *methodicals* [Richardson and Smaill, 2001]. §9.6 briefly describes a planner based on these techniques. The architecture is general enough to be of use to a wide variety of proof planning strategies.

The λ*Clam* system has undergone development since the implementation of out new critics architecture, and in §9.7 we briefly describe how the current implementation relates to the our design presented here.

9.2 Why λ*Clam*?

Our induction strategy is essentially a *non-uniform* collection of heuristics for guiding proof search, i.e. different parts of the proof require very different guidance. This is especially true for our use of meta-variables. Proof planners are designed for the implementation of such non-uniform strategies, and a wide variety of such examples have already been implemented, e.g. [Kraan, 1994], [Cheikhrouhou and Siekmann, 1998], [Melis and Meier, 2000]. For the purposes of prototyping our strategy, using a proof planner is simpler than the adapting another theorem prover or writing a stand-alone system. Furthermore, some parts of the strategy are even described in terms of proof planning operators, e.g. the speculation critic (see Chapter 7). For these reasons, it was decided that a proof planning system would be used.

The choice of which proof planner to use came down to the ΩMEGA system and one of the *Clam* planners (see §2.4). The *Clam* planners were chosen because, unlike ΩMEGA, they have already been successfully used for the implementation of a number of inductive strategies. It is easier to build upon this work than begin a new implementation in ΩMEGA — although this would be an interesting exercise.

Of the *Clam* family of planners, only the *Clam* v3 system has provision for proof critics, and our strategy specifies two proof critics: the speculation critic (see §7.4) and the side condition critic (see §6.5.5). However, of these systems, only λ*Clam* is being actively maintained and developed. It also has other advantages for our induction strategy: its higher-order meta-logic provides built-in unification for higher-order meta-variables, and methodicals greatly facilitate the specification of complex strategies (see §2.4.2). Hence it was decided to implement a suitable critics mechanism in the λ*Clam* system, in order to implement our induction strategy.

9.3 Defining Proof Critics

In this section we describe how proof critics are defined in *Clam* v3 [Ireland, 1992] [Ireland and Bundy, 1996] and in our new architecture. Each proof critic is associated with a proof planning method (by virtue of sharing its name), and when a method fails the planner attempts to apply an associated critic. A critic is defined by a 4-tuple:

Method The name of the associated method.

Input The partial proof plan, including the method's failed preconditions.

Preconditions Conditions under which the critic is applied.

Effects Instructions to modify the partial plan.

Figure 9.1 shows an example of such a critic definition, for a *wave* critic. Once a critic is chosen, its preconditions are tested, and if they are satisfied, its effects are executed. As well as the partial proof plan, the critic has access to the failed preconditions of its associated method, allowing it to provide an appropriate patch for a particular kind of method failure.

To handle methods with multiple associated critics, a preference order is defined for a set of critics. For example, for *wave* method's critics, the critic with the most general preconditions is chosen [Ireland and Bundy, 1996]. The planner may backtrack over this choice.

```
critic(wave,
       Plan,
       [preconds(Plan, [], [P4: sinkable(Pos,G,SPos)])],
       [speculate_lemma(Pos, SPos, G, Rn:Lemma),
        add_wave_rules(Lemma),
        insert_method(Plan, [], wave(Pos,[Rn,_]))]).
```

Figure 9.1: The *Clam* v3 definition of the lemma speculation *wave* critic (from [Ireland, 1992]).

9.3.1 Critic Definition in λ*Clam*

In the λ*Clam* critics architecture we adopt a variation of the definition of proof critic outlined above. Firstly, the critic is named independantly of its associated method. This allows a critic to be associated with several methods, e.g. several variations of the *wave* method could be served by the same critic, or to have different occurrences of the same method associated with different critics. This is discussed further in §9.4.

Secondly, there is a slot in the defining tuple representing the output plan. This brings the definition of critics in line with methods. Hence critic preconditions and effects are declarative statements that relate the input and output slots.

Thirdly, the critic definition also relates an input and output planning agenda, i.e. the list of open nodes in the plan tree. This gives critics the ability to further control the search for a proof plan by changing the agenda. For example, to change the current attention of the planning search. Figure 9.2 shows how the *wave* critic definition from Figure 9.1 might appear in the new format.

Our definition of critics clarifies their function: they are declarative[1] planning operators that work on a global level, i.e. the whole plan and the agenda. This complements

[1] Declarative in theory — λProlog can be used to write non-declarative programs.

```
critic (lemma_speculation Pos Lemma)
       Plan
       Agenda
       (preconds Plan [] [(sinkable Pos G SPos)])
       (speculate_lemma Pos SPos G Rn:Lemma,
        add_wave_rules Lemma,
        insert_method Plan Agenda [] (wave Pos [Rn,_]) NewPlan NewAgenda)
       NewPlan
       NewAgenda.
```

Figure 9.2: How the lemma speculation *wave* critic from Figure 9.1 would be defined in our new critics architecture. The output plan and input/output agendas are explicitly represented, and the critic's name differs from its associated method.

methods which are planning operators that work on a local level, i.e. the individual nodes of the plan tree. Hence our definition of critics brings them more into line with methods.

9.4 Planning Instructions

A feature of clam v3 critics is that a critic is invoked if and only if its associated method fails. However, critics may be of more general utility than this: a proof strategy could invoke a critic without a method failure, in order to *positively critique* a partial plan. For example, a global change to the proof could be part of a proof strategy's design, rather than an exception to it. Furthermore, it would be useful if a critic could have a *contextual association* with a method, i.e. it is invoked only in certain strategic contexts[2].

[2] Ian Green and Alan Smaill, personal communication.

In *Clam* v3 such features do not make sense, as it does not explicitly represent proof strategies above the method level. But in λ*Clam*, proof strategies are represented by compound methods. Hence our λ*Clam* critics planner provides support for both positive critiquing and contextual association with a method. Both are achieved by extending the method expression language with *planning instructions* which modify the planner's behaviour, rather than being applied to the current goal. Planning instructions are treated as atomic methods by the methodical transformations used to obtain the 'next method' from a method expression [Richardson and Smaill, 2001].

9.4.1 Postive Critiques: `crit_inst`

The planning instruction (`crit_inst C`) may be included in a method expression to invoke a positive critique of the plan. The planner interrupts normal planning and applies the critic `C` to the partial plan. For example, the following method expression would apply the `wave` method and then would *always* apply the `lemma_speculation` critic (not a sensible strategy...) :

```
(then_meth (wave Pos [T,D])
           (crit_inst (lemma_speculation Pos Lemma)))
```

9.4.2 Contextual Method/Critic Association: `patch_inst`

The planning instruction (`patch_inst M C`) may be included in a method expression to invoke an association with failure of the atomic method `M`. The planner attempts to apply method `M`. If it succeeds then planning continues as normal. If it fails the critic `C` is applied to the partial plan. In λ*Clam*, the method's evaluated preconditions are stored at the corresponding plan node, so the critic has access to them in order to analyse the method failure. To illustrate, the following method expression would apply the `wave` method, and if it failed apply the `lemma_speculation` critic:

```
(patch_inst (wave Pos [T,D]) (lemma_speculation Pos Lemma))
```

Note that using `(patch_inst M C)` in a method expression is the only way a critic can be associated with a method in the λ*Clam* critics planner, and it does not universally associate a critic with a method. This is not a serious restriction, as e.g. a new compound method `wave2` could be defined with the above method expression, which would behave like the `wave` method with a universally associated `lemma_speculation` critic.

9.5 Criticals

λ*Clam* uses methodicals to compose methods into compound methods, in an analogous way to the composition of tactics via tacticals [Richardson and Smaill, 2001]. The advantage of this is that complex proof strategies involving multiple methods may be explicitly defined, allowing a declarative reading of methods, and making them easier to write.

Another novel aspect of our proof planning architecture is criticals, which allow critics to be composed in an analogous way to methodicals. Using criticals, critic strategies can be built from critics. As well as making complex critics easier to write by breaking them down into small, conceptually simple critics, it allows critic strategies (such as the ‘most general preconditions’ strategy mentioned in §9.3) to be explicitly declared, rather than hard-coded in the planner.

Critic expressions are defined as critics composed via criticals. A critical expression is either *atomic*, containing no criticals, else it is *compound*. Table 9.1 describes

Critical	Type	Description
id_crit	$crit$	Do nothing
$orelse\_crit$	$crit \rightarrow crit \rightarrow crit$	Apply first or second
$then\_crit$	$crit \rightarrow crit \rightarrow crit$	Apply first then second
$repeat\_crit$	$crit \rightarrow crit$	Iterate at least once
try_crit	$crit \rightarrow crit$	Apply or do nothing
$cond\_crit$	$(plan \rightarrow bool) \rightarrow crit$ $\rightarrow crit \rightarrow crit$	First if condition, else second
sub_crit	$ad \rightarrow crit \rightarrow crit$	Apply to subplan at address
$some\_crit$	$(A \rightarrow crit) \rightarrow crit$	Apply for some substitution

Table 9.1: Types and descriptions of criticals. The base types are of critics ($crit$), methods ($meth$), proof plans ($plan$), plan node addresses (ad) and the boolean type ($bool$).

the various criticals available in the λ*Clam* critics planner, and their types.

Following the definition of methodicals [Richardson and Smaill, 2001], we define a meta-interpreter for criticals by a set of rules, given in Figure 9.3. The notation $C : P \mapsto Q$ is taken here to mean critic C applied to P may return Q, where P and Q are critic inputs and outputs. The rules in Figure 9.3 give an inductive definition of $\mapsto$. The order of the rules in the figure indicates the order in which they should be applied. The λ*Clam* critics planner uses these rules to evaluate critic expressions.

Most of the criticals have analogs in λ*Clam*'s methodical set and are quite straightforward. The exceptions are sub_crit, which applies a critic to a specified subplan of the current partial plan, and $some\_crit$ which provides existential quantification for variables in the given critic. This allows variables that are quantified within λ*Clam*'s plan structure to be mentioned in the arguments of critics that are applied to the plan

$$\frac{}{\mathit{id\_crit} : P \mapsto P}$$

$$\frac{C_1 : P \mapsto Q}{\mathit{orelse\_crit}\ C_1\ C_2 : P \mapsto Q}$$

$$\frac{C_2 : P \mapsto Q}{\mathit{orelse\_crit}\ C_1\ C_2 : P \mapsto Q}$$

$$\frac{C_1 : P \mapsto R \quad C_2 : R \mapsto Q}{\mathit{then\_crit}\ C_1\ C_2 : P \mapsto Q}$$

$$\frac{C : P \mapsto R \quad \mathit{repeat\_crit}\ C : R \mapsto Q}{\mathit{repeat\_crit}\ C : P \mapsto Q}$$

$$\frac{}{\mathit{repeat\_crit}\ C : P \mapsto P}$$

$$\frac{C : P \mapsto Q}{\mathit{try\_crit}\ C : P \mapsto Q}$$

$$\frac{}{\mathit{try\_crit}\ C : P \mapsto P}$$

$$\frac{C_1 : P \mapsto Q}{\mathit{cond\_crit}\ (\lambda x.\Delta)\ C_1\ C_2 : P \mapsto Q}\ \text{if } \Delta[P/x] \text{ holds}$$

$$\frac{C_2 : P \mapsto Q}{\mathit{cond\_crit}\ (\lambda x.\Delta)\ C_1\ C_2 : P \mapsto Q}\ \text{if } \neg\Delta[P/x] \text{ holds}$$

$$\frac{C : Q \mapsto Q'}{\mathit{sub\_crit}\ \alpha\, C : P[Q]_\alpha \mapsto P[Q']_\alpha}$$

$$\frac{C[v/x] : P \mapsto Q}{\mathit{some\_crit}\ (\lambda x.C) : P \mapsto Q}\ v \text{ not in } C$$

Figure 9.3: Rules for interpreting criticals

```
plan_goal Goal Method Plan :-
      construct_root Goal Method Root,
      planner [nil] Root Plan.

planner [] Plan Plan.

planner Agenda Plan FinalPlan :-
      Agenda = [Address|_],
      expand_node Address Plan ExpandedPlan Critic,
      apply_critic Critic Agenda ExpandedPlan NewAgenda NewPlan,
      planner NewAgenda NewPlan FinalPlan.
```

Figure 9.4: The main loop of a depth-first proof planner with critics

below the variable's binder, whilst still permitting these critics to appear in compound critics which are applied *above* it[3].

9.6 A Critics Planner

In this section we provide a more detailed description of a depth-first planner that uses the techniques outlined above, in order to more precisely specify the intended behaviour. Figure 9.4 shows the main planning loop of the planner as a simple λProlog program. The planner is called via plan_goal, which constructs the root node of the proof plan and initiates the planning loop. The loop applies *planning steps* until the agenda is empty, via planner. Each planning step consists of two actions: expand_node followed by apply_critic.

[3] In fact, we subsequently extended λ*Clam* with a similar some_meth methodical as part of building the *Dynamis* system (see the next chapter).

9.6.1 Expand Node

The planner takes the first address on the agenda and finds the corresponding node of the partial proof plan. The method expression at this node is evaluated to find the next atomic method, following the rules in [Richardson and Smaill, 2001]. In our planner this may return a planning instruction instead. If so, the remainder of the method expression is stored at the current node.

A new plan and critic are computed as follows:

- For a method, the planner attempts to apply the method. If successful, then the child nodes are added to this node. The critic is taken to be `(children A)`, where `A` is a list of the new child node addresses.

- For a `(patch_inst M C)`, the planner attempts to apply the method `M` as above. But if `M` fails, the planner stores the failed preconditions in the plan node, and the critic is taken to be `C`. The failure of `M` on backtracking causes `expand_node` to fail.

- For a `(crit_inst C)`, the critic is taken to be `C`.

9.6.2 Apply Critic

The critic from `expand_node` is a critic expression of the form `(children A)`, indicating a method has already been applied. If the latter is true then normal depth-first proof planning continues: the address at the top of the agenda is removed and replaced with the new child addresses `A`.

Otherwise the critic expression is used to transform the agenda/partial proof plan, using the rules given in Figure 9.3. Note that in this case the agenda has not necessarily been changed by the planning step. Unless the critic explicitly alters the top of the agenda, the next planning step will return to the same node. However, the method expression has been changed, so this step will not necessarily be repeated.

9.7 Development in λ*Clam*

The critics planner is implemented in λ*Clam* version 2.0. Although the actual implementation was complicated by other considerations (e.g. tracing, alternative search strategies) its behaviour was essentially the same as described in §9.6.

Subsequent development of the λ*Clam* system by a number of other authors has changed the implementation of the planner, but has not affected the critics functionality. In the next chapter we use λ*Clam* version 4.0. The most significant change is the move to a *context planner*, which replaces an explicit λProlog term representation of the partial proof plan with an implicit representation using asserted facts.

As a result the critic definitions have no explicit plan input/output slot. Instead of an input plan slot the preconditions are used to access the plan. The output plan slot takes the form of an add/delete node list. The critics are described in Chapter 10.

9.8 Summary

This chapter presented a novel proof planning architecture based on proof critics extended with *criticals* and *planning instructions*. The advantages of this architecture are:

- A critic may be specified as being associated with the failure of multiple methods or a method in a specific strategic context, using the $patch\_inst$ planning instruction.

- A critic need not be tied to a method failure, instead being invoked as part of a strategy's normal execution, using the $crit\_meth$ planning instruction.

- Complex critics and critic strategies can specified in a modular and declarative manner using criticals.

- By changing the planning agenda a critic can influence the proof search.

Our definition of critics brings them more into line with methods. The λ*Clam* proof planner was extended with this critics architecure in order to implement our strategy.

Chapter 10

The Dynamis System

10.1 Introduction

In order to test the inductive theorem proving strategy described in this thesis, we implemented it as a set of methods and critics in the λ*Clam* proof planner (version 4.0) [Dennis and Brotherston, 2002]. This chapter describes the implementation, which we have called *Dynamis*.

Dynamis's method/critic architecture is based on the three part modular structure described in Chapter 6:

REFINE-CASE A middle-out strategy for constructing a suitable step case. This is implemented in the `mo_step_case` method, described in §10.3.

WELLFOUND-HYPS A strategy for proving there exists a wellfounded relation under which each inductive hypothesis is less than its conclusion. This is implemented in the `wellfound_strat` method, described in §10.4.

EXHAUST-CASES A corrective strategy that shows that the induction cases are exhaustive. This is implemented in the `case_strat_basic` and `case_strat_rec` methods, described in §10.5.

In addition, a fourth strategy is used to discharge base cases and post-fertilisation goals. This is implemented in the `waterfall` method, described in §10.6. These strategies can be employed by a small number of top-level methods that direct the search for a proof plan of an inductive conjectures, which we describe in §10.2.

This chapter gives the λProlog definitions for all *Dynamis*'s compound methods, the speculation critic and some of the key atomic methods. Most of the lower-level atomic methods are omitted, as they are not essential for understanding the overall operation of the system. More information about the *Dynamis* system, including the omitted methods and explanations of the λProlog predicates used in the method conditions, is given in Appendix C.

10.1.1 What's Not Implemented

Several aspects of our strategy have not been realised in this implementation:

- Planning step cases with multiple induction hypotheses (see Chapter 4).
- Generating induction rules with multiple step cases (see Chapter 6).
- The side condition critic for the wellfoundedness strategy (see Chapter 6).

Also, the rippling and rewriting methods do not perform case splits. All of these features were not implemented due to a lack of time only — we foresee no difficulties in principle.

10.2 The Top-Level Strategy

The top-level strategy coordinates the search for a complete plan. It is implemented via the method `dynamis_main`, which can be configured to use a variety of submethods for certain parts of the proof. `dynamis_main` is defined in terms of the methods `schematic_induction`, `construct_cases` and `wellfounded`. This section describes all four methods.

The *Dynamis Knowledge Base* is used to store the following global information about the planning attempt:

- a list of the types of the leading universal quantified variables in the original conjecture;
- a list of the inductive proof's base and step cases;
- the wellfounded relation used to justify the induction;
- a list of constraints on this relation and the corresponding constraint solver.

See the predicates `dkb_types` etc. in Appendix C for details about accessing the knowledge base.

Method: `schematic_induction`

The `schematic_induction` method, shown in Figure 10.1, is the key method in *Dynamis*'s strategy. It corresponds to the application of the (as yet unknown) induction rule. The precondition (1) succeeds if the input goal `Goal` is a sequent with a universally quantified conclusion. It constructs a schematic step case `StepGoal` and the *Dynamis* knowledge base `KB`. The postconditions (2) and (3) construct respectively a wellfoundedness goal `WellGoal` for this step case, and a goal `CaseGoal` stating that

Method: `(schematic_induction KB)`

Goal: `Goal`

Pre: `(schematic_stepcase Goal KB StepGoal)    (1)`

Post:

```
(wellfound_goal Goal KB WellGoal,           (2)
 exhaustive_goal Goal KB CaseGoal)          (3)
```

SubGoal:

```
(StepGoal ** (WellGoal ** (CaseGoal
 ** ((maybeCases Goal KB) ** (wfGoal KB)))))
```

Figure 10.1: `schematic_induction` is the first atomic method applied by *Dynamis*, parameterised by the knowledge base `KB`. Compare with `dynamis_main` (Figure 10.3) to see how each subgoal is planned.

---

this step case is an exhaustive case analysis[1]. Note that λ*Clam* uses `**` to represent goal conjunction.

The method also produces a subgoal `(maybeCases Goal KB)` that will be transformed into any additional proof cases that are found later in the planning attempt, and a subgoal `(wfGoal KB)` that represents the satisfiability of the constraints on the wellfounded relation used to justify the induction. This last meta-level goal is distinctive in that it will not be mapped onto an object-level goal in any execution of the proof plan. It plays a purely meta-level rôle in the final stage of planning, when a wellfounded relation that satisfies the constraints is selected (see §10.2).

If the proof plans were used to produce object-level proofs, the wellfoundedness

[1] This goal will always fail, with the failed proof used to find the missing cases. See Chapter 5, and §10.5 below.

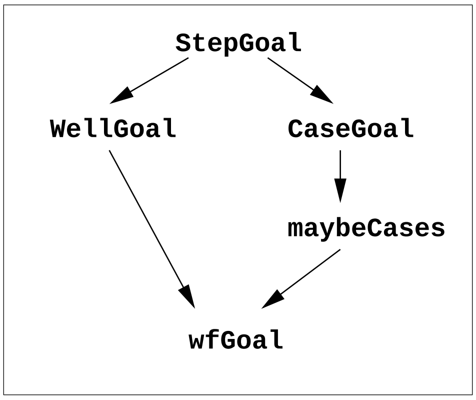

Figure 10.2: Goal ordering for the `schematic_induction` method (see Figure 10.1).

of $\prec$ would have to be shown. This could be done by providing a prefabricated proof that any relation defined using a measure function is wellfounded. However, no proof search would be required for this either on the object or meta-level, so there is no need to associate this with `wfGoal` or the method which plans it. It could be associated e.g. with the `schematic_induction` method.

The ordering of `schematic_induction`'s subgoals is important, and assumes the use of a depth-first planner, or at least one that respects the goal order. This is the result of dependencies between the subgoals that require certain branches to be completed before others are planned, else the final plan will not represent a valid inductive proof. For example, if the `caseGoal` is planned before the `StepGoal` it will be trivially discharged, as no meta-variables have been instantiated. Any wellfounded step case found by planning `StepGoal` will then be considered case exhaustive, and the strategy will terminate with a only one step case and no base cases. Violations of other goal dependancies cause similar problems.

The order requirements are shown in Figure 10.2. *Dynamis* uses λ*Clam*'s depth-

Method: `(dynamis_main StepStrat WellStrat CaseStrat BaseStrat)`

```
(complete_meth
(then_meths (schematic_induction KB)
            (pair_meth (then_meth StepStrat BaseStrat)
            (pair_meth WellStrat
            (pair_meth CaseStrat
            (pair_meth (then_meth construct_cases BaseStrat)
                       (wellfounded Relation)))))))
```

Figure 10.3: `dynamis_main` is *Dynamis*'s main top-level method. It is parameterised by four methods.

---

first or iterative-deepening planner to ensure this order in the method definition is respected. A more sophisticated approach could represent this information in a declarative manner, but this cannot be done within λ*Clam*'s planning framework.

The goal order necessary for soundness does not specify when `WellGoal` is planned relative to `CaseGoal` and `maybeCases`. As discussed in Chapter 6, it is better to plan `WellGoal` as soon as possible, in order to avoid wasting time on step cases that cannot be shown to be wellfounded. Hence, `WellGoal` is planned immediately after `StepGoal`.

Method: `dynamis_main`

The main top-level method for *Dynamis* is `dynamis_main`, shown in Figure 10.3. It first applies `schematic_induction`, followed by an appropriate method for each of the resulting subgoals — compare Figure 10.3 and Figure 10.1 to see the mapping between goals and methods.

Method: `dynamis_crit`

```
(dynamis_main (mo_step_case spec_critic_ripple)
              wellfound_strat
              case_strat
              (waterfall dynamis_crit))
```

Method: `dynamis_crit_once`

```
(dynamis_main (mo_step_case spec_critic_ripple)
              wellfound_strat
              case_strat
              (waterfall (ind_strat normal_ind)))
```

Method: `(dynamis_lim N)`

```
(dynamis_main (mo_step_case (n_spec_ripples N))
              wellfound_strat
              case_strat
              (waterfall (dynamis_lim N)))
```

Method: `(dynamis_lim_once N)`

```
(dynamis_main (mo_step_case (n_spec_ripples N))
              wellfound_strat
              case_strat
              (waterfall (ind_strat normal_ind)))
```

Figure 10.4: Some configurations of the the top-level method `dynamis_main`.

In our implementation `dynamis_main` is actually a methodical, parameterised by four methods. Hence it acts as a template for a range of top-level methods. In order, these parameters are:

- `StepStrat`, a method for the initial schematic step case, e.g. `mo_stepcase` (see §10.3).
- `WellStrat`, a method for the wellfoundedness proof, e.g. `wellfound_strat` (see §10.4).
- `CaseStrat`, a method for the exhaustive cases proof, e.g. `case_strat` (see §10.5).
- `BaseStrat`, a method used to plan base cases and post-fertilisation subgoals e.g. `(waterfall IndStrat)` (see §10.6).

Some possible configurations of `dynamis_main` are shown in Figure 10.4. For instance, `dynamis_crit` uses the speculation critic in the step case, and a rewriting-generalisation-induction waterfall to discharge base cases, where it may be called recursively to plan nested inductions. Contrast this with `dynamis_lim_once` which uses a fixed number of speculation steps in the step case, and uses the standard λ*Clam* induction methods to plan nested inductions. Allowing *Dynamis* to use a variety of methods gives us a straightforward way of comparing various combinations of strategies.

Method: `construct_cases`

The `(maybeCases Goal KB)` goal produced by `schematic_induction` is passed to the `construct_cases` method, shown in Figure 10.5. The latter constructs the additional proof cases that the exhaustive cases strategy has identified as missing and already added to the knowledge base.

Method: `construct_cases`

Goal: `(maybeCases Goal KB)`

Pre: `true`

Post:

```
(dkb_cases KB [_|Cases],
 list_to_goal Cases (new_case Goal) NewGoals)
```

Subgoal: `NewGoals`

Figure 10.5: The `construct_cases` method generates subgoals `NewGoals` corresponding to the proof cases `Cases` that have been added to the knowledge base during the case synthesis strategy.

The preconditions are trivial, so the method always applies. The postconditions retrieve the added cases `Cases` from the knowledge base `KB`, ignoring the initial step case that has already been proven. Each case is mapped onto a new subgoal, constructed by restricting the original goal `Goal` to that case. `NewGoals` is a conjunction of these new proof cases.

Method: `wellfounded`

The last step during a successful planning attempt is always the application of the `wellfounded` method to the final subgoal produced by `schematic_induction`. The `wellfounded` method is shown in Figure 10.6. It chooses a wellfounded relation which satisfies the constraints built up during during the proof. The preconditions retrieve the relation, constraints and constraint solver from the knowledge base. Applying the

Method: `(wellfounded Relation)`

Goal: `(wfGoal KB)`

Pre:

```
(dkb_solver KB Solver,
 dkb_relation KB Relation,
 dkb_constraints KB Constraints,
 Solver Relation Constraints)
```

Post: `true`

Subgoal: `trueGoal`

Figure 10.6: The `wellfounded` method applies a constraint solver to the constraints on the wellfounded relation and the meta-variable representing this relation, instantiating the latter.

solver to the other data instantiates the relation.

As mentioned above, this is a purely meta-level step which would not have any underlying object-level proof if the proof plan were executed[2]. However, it does instantiate the meta-variable representing the wellfounded relation in the plan, which is required to give a executable plan.

10.3 The Step Case Strategy

Recall from Figure 10.1 that the `schematic_induction` method sets up a schematic step case goal using the precondition

```
(schematic_stepcase Goal KB StepGoal)
```

[2] A general object-level proof of wellfoundedness can be given, and need not be associated with this method.

Method: `(mo_step_case Ripple)`

```
(then_meth embed_hypothesis
(then_meth Ripple
(then_meth mo_fertilise
          (try_meth (repeat_meth redundant)))))
```

Figure 10.7: The `mo_step_case` method.

where `Goal` is the original universally quantified goal. This instantiates `StepGoal` to a `caseSchema` subgoal where the universal variables are replaced by meta-variables. This is best illustrated by example: if `Goal` has a single universal quantifier, i.e. it is if the form:

```
(seqGoal (H >>> (app forall [T, (abs F)])))
```

then `StepGoal` is of the form

```
(allGoal T x\ (caseSchema (C x) H
                        (preRippleHyps (F x) [(F (A x))]) (F (B x))))
```

The `caseSchema` contains a meta-variable condition on the step case `(C x)`, the step case skeleton `(F x)`, a single induction hypothesis `(F (A x))` and an step case conclusion `(F (B x))`. Note that the induction terms in hypothesis and conclusion are represented by meta-variables `A` and `B` respectively. The types of the induction terms and a representation of this proof case are entered into the knowledge base `KB`.

The method works in a similar way for goals with more than one universal quantifier.

Method: `mo_step_case`

The strategy for planning a proof of the schematic step case goal is implemented by the compound `mo_step_case` method, shown in Figure 10.7. Like `dynamis_main`, it is a methodical, taking a rippling method as an argument. The method follows the standard step case proof plan outline: embed the hypotheses in the conclusion, ripple, then fertilise. Redundant universal quantifiers are removed post-fertilisation.

Suitable choices for the `Ripple` method would be `n_spec_ripples`, which allows a fixed number of speculative ripple steps, or `spec_critic_ripple`, which uses a critic to control speculation. Both are described in §10.3.2.

10.3.1 Embeddings

Embeddings in *Dynamis* are implemented in a slightly different way from the standard λ*Clam* methods. In λ*Clam*, an embedding is a tree which has the same structure as the skeleton term syntax tree, and has, at each node, a term address that indicates where this node is mapped to on the target term. Wave fronts are implicit in this representation: they correspond to the parts of the target term syntax tree which are not referenced by address in the embedding.

Because our strategy involves a lot of explicit computation with wave fronts, e.g. neutralisation, we have modified embeddings to explicitly represent wave fronts with a constructor. This saves a great deal of effort by avoiding the repeated reconstruction of this information.

Wave fronts may be of varying thickness in our representation, so for example a wave front of thickness $n+m$ will be equivalent to two wave fronts of thickness n and m. Hence, wave fronts may be merged and split. We keep wave fronts in maximally merged form.

Method: `(n_spec_ripples N)`

```
(then_meth (cond_meth isSideCond solve_sidecond
                      (speculative_ripple _ _ _))
           (cond_meth (g\ (N > 1, M is N - 1))
                      (n_spec_ripples M)
                      definite_rippling))
```

Figure 10.8: The `n_spec_ripples` method.

Explicitly representing wave fronts increases the number of possible embeddings, as wave fronts may now be individually directed. In order to reduce the number of possible embeddings, we apply the following constraints:

- Blocks of wave fronts must all have the same direction.
- Outwards wave fronts may not appear below inwards wave fronts.
- Variables may only be embedded in or into by terms of the same type. This constraint is not enforced in the version of λ*Clam* used, leading to the possibility of spurious embeddings.

Method: `embed_hypothesis`

The `embed_hypothesis` method embeds the step case skeleton into the schema's induction hypothesis and conclusion. The embeddings and their weights are stored in the step case goal. This method is described in greater detail in §C.2.

10.3.2 Speculative Rippling

Speculative rippling decides the form of the step case by instantiating the step case meta-variables. We have two alternative methods for doing this: a strategy with a fixed

Method: `spec_critic_ripple`

```
(then_meth (speculative_ripple _ _)
   (repeat_meth
      (cond_meth isSideCond solve_sidecond
         (some_meth2 rule\ ad\
            (patch_meth (definite_ripple rule ad)
                        speculation_critic)))))
```

Figure 10.9: The `spec_critic_ripple` method.

number of speculative steps (`n_spec_ripples`), and a more flexible critic-based strategy (`spec_critic_ripple`). They are shown in Figures 10.8 and 10.9 respectively.

The `n_spec_ripples` method, takes an integer argument `N` and applies the atomic method `speculative_ripple` `N` times before applying `definite_rippling`. Side conditions are passed to `solve_sidecond` (see §10.3.4).

In contrast, the `spec_critic_ripple` method applies `speculative_ripple` once, then repeatedly applies definite_ripple. If this fails, the speculation critic is applied.

Method: `speculative_ripple`

The `speculative_ripple` method, shown in Figure 10.10, performs a speculative ripple step, i.e. one where meta-variables are instantiated. As it is such a key atomic method, we now describe the preconditions in greater detail:

1. The goal's meta-variables `Vars` are identified.

2. The conclusion `Conc` is rewritten to `NewC` with a wave rule `Rule` and side condition `Cond`. The rewrite relation `rewrite_unif` is used, which allows the meta-variables in `Conc` to be instantiated.

Method: `(speculative_ripple Rule Ad)`

Goal: `(caseSchema Case Hs (rippleHyps [IndHyp]) Conc)`

Pre:

```
(IndHyp = (annHyp Hyp Skel EH1 _ _ _ _),
 meta_variables Conc [] ConcVars,
 meta_variables Hyp ConcVars Vars,                                     (1)
 wave_rule_list Rules,
 rewrite_inner (rewr_list Rules rewr_unif) Rule _ Conc NewC Cond Ad,  (2)
 embedding Skel EC1 NewC,                                              (3)
 reverse Ad At,
 speculative_rule Rule Flag,                                           (4)
 cancel_context Flag At Skel NewSkel Hyp EH1 EH2 NewC EC1 EC2,         (5)
 reembed NewSkel bool Hyp bool EH2 EH3,
 not_all_meta_vars Vars)                                               (6)
```

Post:

```
(tidy_hyp_context EH3 EH4 HW,
 tidy_conc_context EC2 outward EC3 Out In,
 NewIndHyp = (annHyp Hyp NewSkel EH4 HW EC3 Out In),
 Main = (caseSchema Case Hs (rippleHyps [NewIndHyp]) NewC),
 condition_goal Cond Case Hs
               (c\ (caseSchema Case Hs sideCond c)) Main SubGoal)
```

Subgoal: `SubGoal`

Figure 10.10: The `speculative_ripple` method.

Critic: `speculation_critic`

Agenda: `(active_agenda [PlanAd|Agenda])`

Pre:

```
(get_goal PlanAd Goal,
 not (fertilisable Goal),
 Goal = (caseSchema Case Hyps (rippleHyps IndHyps) Conc),
 partial_lhs Rule LHS PartLHS Dir,
 rewrite_so PartLHS LHS Dir Conc ReqConc TermAd,
 once (embedding Conc E1 ReqConc))
```

Post:

```
(tidy_conc_context E1 inward E2 _ NewIn,
 get_continuation PlanAd Continue,
 InGoal = (caseSchema Case Hyps (blockedGoal Conc E2 NewIn) ReqConc),
 InMeth = (then_meth (ripple_in_and_speculate Ripples) CurrentPlan),
 PatchMeth = (ripple_patch Ripples Rule TermAd Continue))
```

Add/Delete:

```
[(add_node [1|PlanAd] (and_node InGoal [1|PlanAd] InMeth _ _ _)),
 (add_node [2|PlanAd] (and_node Goal [2|PlanAd] PatchMeth _ _ _))]
```

New Agenda: `(active_agenda [[1|PlanAd]|[[2|PlanAd]|Agenda]])`

Figure 10.11: The `speculation_critic` critic.

3. The rewrite step is checked to be skeleton preserving by reembedding the skeleton `Skel` into `NewC`.

4. If `Rule` is classified as non-constructor , then `Flag` is switched on, which indicates neutralisation may instantiate meta-variables in `Hyp`.

5. Zero or more corresponding wave fronts in hypothesis and conclusion are cancelled out via neutralisation (`cancel_context`). New embeddings, `EH3` and `EC2`, are found for hypothesis and conclusion, along with an expanded skeleton, `NewSkel`.

6. A check that at least one meta-variable in `Vars` has been instantiated.

The postconditions simply construct the new subgoal, adding an optional subgoal for the rewrite side condition `Cond` if it is trivially true or false.

Critic: `speculation_critic`

Figure 10.11 shows the critic `speculation_critic`. Its preconditions first check that the failed goal is not fertilisable. They then find a partially matching wave rule `Rule` that would apply if the conclusion `Conc` were of the form `ReqConc`. The postconditions set up two goals: the first `InGoal` which aims to ripple `ReqConc` inwards to match `Conc`, via the method `ripple_in_and_speculate` (see Figure 10.12); the second `Goal` tries to continue the ripple proof with the method `ripple_patch` (see Figure 10.13), which will apply, in reverse, the ripple steps `Ripples` used to solve `InGoal`.

10.3.3 Definite Rippling

Figure 10.14 shows the `definite_rippling` method, which implements definite (normal) rippling, where no meta-variable instantiation takes place. It repeatedly applies

Method: `(ripple_in_and_speculate RipplePlan)`

```
(orelse_meth
   (speculate_wavefronts Ripples RipplePlan)
   (then_meth
      (repeat_meth
         (some_meth2 rule\ ad\ (forwards_ripple rule ad Ripple)))
      (speculate_wavefronts Ripples RipplePlan)))
```

Figure 10.12: The `ripple_in_and_speculate` method.

Method: `(ripple_patch Ripples Rule TermAd Continue)`

```
(then_meth redo_embeddings
(then_meth Ripples
(then_meth (definite_ripple Rule TermAd)
           Continue)))
```

Figure 10.13: The `ripple_patch` method.

Method: `definite_rippling`

```
(repeat_meth
   (cond_meth isSideCond solve_sidecond
              (orelse_meth meta_ripple
                           (some_meth2 definite_ripple))))
```

Figure 10.14: The `definite_rippling` method.

`definite_ripple` and `meta_ripple`. Side conditions are passed to `solve_sidecond` (see §10.3.4).

Method: `definite_ripple`

The `definite_ripple` method peforms a single definite ripple step. This can be either a wave measure decreasing ripple, or a creational ripple that remove hypothesis wave fronts. In both cases, the conclusion is rewritten with the relation `rewr_match`, which does not instantiate metavariables. This method is described in more detail in §C.2,

Method: `meta_ripple`

The `meta_ripple` method replaces the conclusion embedding with one that is smaller under the wave measure, without rewriting the conclusion. See §C.2 for the method definition.

10.3.4 Side Conditions

Side conditions are defined as goals of the form

```
(caseSchema _ _ sideCond _)
```

and are sometimes generated by the atomic ripple methods. The compound method `solve_sidecond` is used to solve these goals, by repeated application of the atomic method `simplify_sidecond` (see Appendix C for both methods). This simplifies or discharges a side condition goal in one of a the following ways:

- Discharge a trivially true goal.

- Discharge using the step case condition.

Method: `mo_fertilise`

```
(orelse_meth strong_fertilise
            (then_meth (orelse_meth (weak_fertilise _)
                                    strong_fertilise_prop)
                       replace_metavariables))
```

Figure 10.15: The `mo_fertilise` method.

- Discharge using a hypothesis.
- Simplify using propositional rules.
- Simplify using symbolic evaluation.
- Discharge by assuming it is true. This partially instantiates the meta-variable part of the step case condition.

No search is allowed over these options to avoid a side condition being repeatedly solved during backtracking and causing unnecessary search. If a cut methodical had been available in λ*Clam* it would have been possible to have a more modular implementation that made use of generic rewriting methods and still avoided backtracking, rather than the extremely special-purpose method used here.

10.3.5 Fertilisation

The `mo_fertilise` method, shown in Figure 10.15, applies either a) strong fertilisation or b) weak fertilisation or strong fertilsation which leaves a residue goal. The latter is followed by `replace_metavariables`, which transforms the schematic step case goal into a sequent goal without meta-variables.

Method: `wellfound_strat`

```
(then_meth (construct_wf_goals Consts)
           (orelse_meth estimation_strat
                        (some_meth ignore_position)))
```

Figure 10.16: The `wellfound_strat` method.

These fertilsiation submethods are reimplementations of standard inductive methods, and are described in full in §C.2. Briefly, `strong_fertilise` unifies the hypothesis and conclusion, whereas (`strong_fertilise_prop`) uses the induction hypothesis to rewrite an arbitary subproposition of the conclusion. Weak fertilisation rewrites one side of an equality or iff with an induction hypothesis (`weak_fertilise`).

10.4 The Wellfoundedness Strategy

The `schematic_induction` method sets up a wellfoundedness goal `WellGoal` for the step case, using the query (`wellfound_goal Goal KB WellGoal`), where `Goal` is the original universally quantified goal and `KB` is the knowledge base. For a `Goal` of the form

```
(seqGoal (Hyps >>> Conc))
```

the wellfoundedness goal is of the form

```
(stepReduces Hyps KB)
```

This is simply a dummy meta-level goal which acts as a place holder for the wellfoundedness goals, as we do not know the form they should take until after the step case is complete. The case may involve an arbitary number of induction hypotheses.

The actual explicit construction of the reduction goals is delayed until the application of the `wellfound_strat` method, shown in Figure 10.16. This first applies the `construct_wf_goals` method, which builds the wellfoundedness goals from information stored in the knowledge base, i.e. the induction cases created so far, the types of the potential induction positions and the constraints on the wellfounded relation.

The predicate `wellfound_goals` (see Appendix C) is used to turn each induction case into a set of wellfoundedness goals. For each case and for each induction position, a goal is constructed which states that this position is reduced under some measure. Therefore, taken together, the goals for a given case state that *every* induction position reduces under some measure. This is clearly an unnecessarily strong requirement, so we allow some of these goals to be ignored, *provided that the constraints on the wellfounded relation remain satisfiable*. A goal can be ignored if it cannot be planned using `estimation_strat` (see below) — instead the `ignore_position` method (see §C.3) is applied to end the plan branch. The method a) adds an `ignore` constraint to the knowledge base, indicating that the induction position must be ignored and b) checks that constraints remain satisfiable. This prevents the system ignoring all the induction positions, and so producing a plan that fails to validate the induction.

10.4.1 Estimation

The estimation strategy discharges wellfoundedness goals, and is implemented via the `estimation_strat` method, shown in Figure 10.17. It uses four submethods: `begin_estimation`, `lower_estimate`, `upper_estimate` and `trivial_estimate` (see §C.3). These implement Walther's estimation technique, extended with upper estimation, in a straightforward manner (see §6.5).

Method: `estimation_strat`

```
(then_meths (begin_estimation N)
            (pair_meth
                (then_meth (repeat_meth (orelse_meth lower_estimate
                                                     upper_estimate))
                           trivial_estimate)
                (then_meth abstract_metavars
                           rewrite)))
```

Figure 10.17: The `estimation_strat` method.

Submethod `begin_estimation` converts the initial meta-level `redGoal` to two subgoals: an `estGoal`, representating the estimation goal, and a sequent, which states that the difference equivalent `Diff` generated by the estimation proof plan is true. Note that `begin_estimation` also adds to the knowledge base constraints on the measure for the corresponding induction position.

Submethod `lower_estimate` and `upper_estimate` implement the lower and upper estimation rules respectively. Finally, `trivial_estimate` terminates trivial estimation plan branches.

These definitions of these submethods are given in §C.3.

10.5 The Case Synthesis Strategy

The case synthesis strategy described in Chapter 5 is implemented by the compound methodical `case_strat` (see Figure 10.18). It relies on the following submethods:

- `set_conditions` instantiates to `trueP` any remaining meta-variable part of the side-conditions of the known step case, i.e. no more conditions can be imposed

Method: `case_strat`

```
(then_meth set_conditions
(then_meth (case_equiv _)
(then_meth (case_induction (tuple_split _))
           (repeat_meth
              (then_meth (some_meth case_equiv)
                         (orelse_meth trivial_case
                         (orelse_meth (some_meth missing_case)
                         (orelse_meth (some_meth exists_casesplit)
                                      case_indstrat)))))))
```

Figure 10.18: The `case_strat` method.

on the step case.

- the `case_equiv` method simplifies the case synthesis goal.
- the `trivial_case` method, which identifies trivial plan branches.
- the `missing_case` method, which identifies failed plan branches, to be patched by adding the missing case(s).
- an existential casesplit (see 5.4.2) method `exists_casesplit`.
- the `case_indstrat` tries to solve the case synthesis goal with induction or a case split.

All these methods are given in §C.4.

Figure 10.19 shows the `case_indstrat` method, which performs induction and case splits in the case synthesis strategy. After applying an induction or case split with the `case_induction` method, the method tries rippling and fertilisation, either of

Method: `case_indstrat`

```
(then_meth (some_meth case_induction)
(then_meth (try_meth (some_meth exists_casesplit))
(then_meth (try_meth (then_meth (repeat_meth (some_meth case_ripple))
                                (repeat_meth case_fertilisation)))
           remove_case_hyps)))
```

Figure 10.19: The `case_indstrat` method.

which may fail without causing the method to fail. These submethods are defined in §C.4.

The submethods `case_ripple` and `case_fertilisation` are reimplementations of standard inductive methods in the context of the case synthesis proof. It should be possible to use these here instead of special-purpose methods.

10.6 The Base Case Strategy

For base case and post-fertilisation subgoals, a rewriting/generalisation/induction waterfall is used [Boyer and Moore, 1979]. It is implemented in the `waterfall` method, presented in Figure 10.20. The submethods used in this definition can be found in §C.5.

Rewriting and generalisation are performed by the `rewrite` and `generalise` methods, whereas the induction method is passed to `waterfall` as a parameter, allowing a variety of inductive strategies to be used. Note that `all_e_nf` is used to reintroduce stripped universal quantifiers for generalisation and induction.

Method: `(waterfall IndStrat)`

```
(then_meth (try_meth rewrite)
(then_meth (normalise all_e_nf)
(then_meth (try_meth (repeat_meth generalise))
(then_meth (cond_meth univ_quantified Induction fail_meth)
           (waterfall IndStrat)))))
```

Figure 10.20: The `waterfall` method.

10.7 Summary

In this chapter we detailed the implementation of our inductive proof strategy as a set of λ*Clam* methods and critics. This makes concrete the theoretical ideas outlined in previous chapters, and allows us to test these theories, as described in the next two chapters.

Chapter 11

Experimental Evaluation

11.1 Introduction

In this chapter we report on the evaluation of our induction strategy by experimental testing of the *Dynamis* system. The test set was made up of problems that could not be solved using recursion analysis, either gathered from or inspired by the literature. The experiments were intended both to test the strategy and compare it with lazy induction [Protzen, 1995], the previous state-of-the-art in induction selection.

The current implementation of the strategy in *Dynamis* can construct only induction rules with single step cases containing single induction hypotheses. This limited the scope of the evaluation to theorems that can be solved with such induction rules.

The hypotheses under consideration were:

1. The induction strategy works as described, automatically generating induction rules to plan proofs for a range of theorems which recursion/ripple analysis cannot solve, or for which it selects a sub-optimal rule.

2. There are theorems the strategy proves using non-destructor[1] style induction that cannot be proved by destructor-only lazy induction.

3. All theorems proved by Protzen's lazy induction can also be proved by the strategy.

Unfortunately, direct comparison with a lazy induction system was not possible because the original implementation of the technique in INKA was not available, and the published description [Protzen, 1995] was not detailed enough for a faithful reconstruction. Consequently, evaluation of hypothesis (3) was limited to a comparison based on the results of four theorems published in [Protzen, 1995].

The rest of the chapter is structured as follows: §11.2 describes the methodology adopted for these experiments. In §11.3 we report on the results, and in §11.4 discuss to what extent they support the hypotheses presented above.

11.2 Methodology

A collection of 24 theorems was compiled, selected on the basis that they are not solvable using recursion/ripple analysis given the lemmas provided. The set included eight theorems taken from the literature. These were used as inspiration in designing the rest of the set. Their unsolvability by using recursion/ripple analysis was checked by hand. Although this could have been checked automatically — for example, using λ*Clam* — we have found that, in general, simulation of a technique by hand is more likely to produce a proof than an actual implementation, because it is not subject to the particular idiosyncrasies of the system. It is more relevant to this experiment that a

[1] i.e. induction rules which are destructor style, or neither constructor nor destructor, i.e. they have term structure in both induction hypothesis and conclusion (see §3.2).

Theorem	Statement	Source
D1	$half(s(x)+y) \leq x+y$	[Protzen, 1995]
D2	$odd(x+y) \leftrightarrow \neg odd(s(y+x))$	[Protzen, 1995]
D3	$odd(x+y) \leftrightarrow odd(y+x)$	Variant D2
D4	$sum(l,x) = sum(l,0)+x$	Variant T15
D5	$last(qsort(smaller(n,l))) \leq n$	[Protzen, 1995]

Table 11.1: The development theorem set.

recursion/ripple analysis cannot prove a theorem *in principle*, rather than in the context of a single system.

The theorems fall into four main groups:

Arithmetic (D1 to D3, T1 to T8) — Theorems about Peano arithmetic.

Lists (D4, D5, T9 to T14) — Theorems about list length and order.

Folding (T15 to T17) — Higher order theorems about list folding functions.

Gilbreath Card Trick (T18, T19) — Two theorems about lists over the $red/black$ datatype [Huet, 1991].

We use the following naming system: D or T indicates a development or test theorem (see below). Each of these sets of theorems are numbered (e.g. D1). Finally, each the theorem identifier is followed by a C or a D to indicate whether constructor style or destructor style function definitions were used (e.g. D1C).

Three of the theorems — T14, T18 and T19 — required proofs with multiple step cases. However, we included them in our evaluation to see if *Dynamis* could construct the initial step case.

Theorem	Statement	Source
T1	$x \times y = y \times x$ without $U \times s(V) = (U \times V) + U$	Original
T2	$half(s(x)) \leq x$	Original
T3	$half(n + s(m)) \leq n + m$	Variant D1
T4	$half(n + m) \leq m + n$	Variant D1
T5	$half(quot(n,m)) = quot(half(n),m)$	[Protzen, 1995]
T6	$even(x+y) \wedge even(y+z) \;\rightarrow\; even(x+z)$	Original
T7	$x \neq 0 \rightarrow (odd(x+y) \leftrightarrow \neg odd(y + p(x)))$	Variant D2
T8	$y \neq 0 \rightarrow (odd(x+y) \leftrightarrow \neg odd(p(y) + x))$	Variant D2
T9	$rotate(len(l), l <> k) = (k <> l)$	[Ireland and Bundy, 1996]
T10	$half(len(l)) \leq half(len(l <> m))$	Original
T11	$len(oddelems(l <> m) \leq len(m <> l)$	Original
T12	$len(evenelems(l <> m)) \leq len(m <> l)$	Original
T13	$len(evenelems(l <> x :: m)) \leq len(l <> m)$	Original
T14	$perm(x,y) = perm(y,x)$	[Protzen, 1995]
T15	$x \circ id = x \;\wedge\; x \circ (y \circ z) = (x \circ y) \circ z$ $\rightarrow\; foldleft\_tr(\circ, x, l) = (x \circ foldleft\_tr(\circ, id, l))$	[Paulson, 1991]
T16	$foldleft\_tr(\circ, x, l) = foldleft(\circ, x, rev(l))$	Original
T17	$foldright\_tr(\circ, x, l) = foldright(\circ, x, rev(l))$	Original
T18	$shuffle(x,y,z) \wedge alter(x <> y) \wedge even(len(x <> y))$ $\wedge head(x) \neq head(y) \rightarrow paired(z)$	[Huet, 1991]
T19	$shuffle(x,y,z) \wedge alter(x <> y) \wedge even(len(x <> y))$ $\wedge head(x) = head(y) \rightarrow paired(tail(z) <> head(z) :: nil)$	[Huet, 1991]

Table 11.2: The test theorem set.

We divided the collection into a small development set and a larger test set. The first set was used to improve the performance of *Dynamis* during its development, and is shown in Table 11.1. The development set also included a number of theorems which could be solved by recursion analysis, e.g. the associativity and commmutativity of plus, which we omit here. The system was developed in order to improve the performance on all these problems. For the main test phase, the development of *Dynamis* was halted. The system was then run for the first time on the test theorems, shown in Table 11.2. This two-phase approach was chosen to avoid the development process 'tuning' the system to these particular examples.

The *Dynamis* system was compiled and run using the Teyjus λProlog version 1.0 (beta 33-MRG)[2]. All the timings are from representative runs on a Dell Optiplex GX240 PC with a 1.8GHz Pentium 4 processor running RedHat Linux 8.0.

11.2.1 Configuring *Dynamis*

In this section we describe how *Dynamis* was configured for the experiments.

Both development and test theorems were tried with both constructor and destructor style function definitions, each in separate test runs. For each theorem, the system was run with a variety of lemma configurations, and, if successful, a minimal configuration was determined. If unsuccessful, we tried to determine the lemmas which enabled the system to make the most progress. For constructor (destructor) style definitions we used lemmas that gave a constructor (destructor) style induction.

Some argument bounded lemmas (see §6.5.2) and rewrites rules related to datatypes were made available to the system. The latter group fell into four categories:

[2] Available from the Mathematical Reasoning Group, University of Edinburgh, `http://dream.dai.ed.ac.uk/`

Destructor definitions e.g. $p(s(X)) = X$

Equality axioms e.g. $s(X) = s(Y) \leftrightarrow X = Y$

Inequality axioms e.g. $s(X) \neq 0$

Exhaustive casesplits e.g. $\forall n{:}nat.((n = 0) \vee \exists x : nat.(n = s(x)))$

Definitions and lemmas may be loaded into λ*Clam* as rewriting and/or wave rules. Each rule was classified by hand as being suitable for general rewriting and/or rippling. A rule was classified for use with general rewriting if it maintained the termination of the rule set. For example, recursive cases of destructor style definitions were not accepted. A rule was classified as a wave-rule if it could be annotated as such, with the additional constraint that definitional rules for a function f had the skeleton $f(X_1,\ldots,X_n)$. The order of the rewrite and wave rules was not specified exactly, although definitions were placed before lemmas and 'simpler' rules came first.

For constructor style problems the default strategy was `dynamis_crit` (see §10.2), which uses the speculation critic. If this failed, we attempted to plan the theorem with this strategy modified in one of the following ways:

- Use a fixed number of speculative steps instead of the speculation critic. Top-level methods `(dynamis_lim 1)` and `(dynamis_lim 2)` use one and two steps respectively (see Chapter 10).

- Use the standard λ*Clam* induction methods for nested inductions. This is achieved using the top-level methods `dynamis_crit_once`, `(dynamis_lim_once 1)` and `(dynamis_lim_once 2)` (see Chapter 10).

Using these alternative strategies allowed us to diagnose problems with the default strategy — for example, we can test whether the speculation critic was the cause of particular failure by rerunning the test with a fixed speculation strategy.

For destructor style problems, the strategy `dynamis_lim1` was used by default, as the speculation critic was designed for constructor style induction only. If the single step strategy failed, the double step `dynamis_lim2` was tried — this was also considered as a default approach, simulating the iterative increase of the bound on the number of speculation steps.

Overall, there were three ways the configuration could be modified during the experiment, if the initial default settings failed:

- Adding and removing lemmas.
- Forcing a lemma to be tried before the definitions, using the *Dynamis* clause `needs_priority/5`.
- Modifying the default strategy in one of the predetermined ways outlined above.

In the results below we describe the configurations used for each theorem.

11.3 Results

The results of the evaluation are summarised in Table 11.3. Each theorem has results obtained with constructor and destructor style definitions, and we indicate this with a C or D after the theorem name, e.g. theorem T9 is considered to be two theorems T9C and T9D.

The full results are shown in Tables 11.4 to 11.6. Table 11.4 gives the results for the development theorems. Table 11.5 and Table 11.6 show the results for the constructor and destructor style test theorems respectively. The lemmas used in the evaluation are given in Table 11.7 and Table 11.8.

Set	Style	# Theorems	# Planned	# With Default
Development	Both	10	8	4
	Cons.	5	4	3
	Dest.	5	4	1
Test	Both	38	27	19
	Cons.	19	14	9
	Dest.	19	13	10
Overall	Both	48	35	23
	Cons.	24	18	12
	Dest.	24	17	11

Table 11.3: Results summary.

Tables 11.4 to 11.6 show whether a plan was found for each theorem, and if so the time taken, the number of speculative steps and the minimal set of lemmas used. The constructor style default strategy used the speculation critic, while the destructor style default was the fixed speculation limit strategy. Each table also indicates if an alternative to the default strategy was needed. Alternative strategies involved one or more of the following variations:

Lemma One lemma (marked *) is considered before definitions during rewriting.

Nest Nested inductions are handled by the standard λ*Clam* induction strategy.

Limit For constructor style examples, a fixed speculation strategy was used instead of the critic.

For destructor style (DS) theorems, additional lemmas were required to the ones shown in the results tables, in the form of constructor style (CS) definitions. For exam-

Theorem	Plan	Time (sec)	Alt. Strategy	Spec.	Lemmas
D1C	Yes	4	—	2	L2, L3, L10
D1D	Yes	15	—	2	L2, L3, L4, L8, L9, L10
D2C	Yes	23	—	2	L2, L3
D2D	Yes	22	Nest	2	L2, L3, L4
D3C	Yes	4	—	2	L2, L3
D3D	Yes	14	Nest	2	L2, L3, L4
D4C	Yes	35	Lemma	1	L15a*, L18
D4D	Yes	58	Lemma	1	L2, L4, L15b*, L18
D5C	No	—	—	—	L21, L22
D5D	No	—	—	—	L21, L22

Table 11.4: Development results. 8 of 10 theorems were planned, 4 with the default strategy. * = lemma considered first by rewriting.

ple, if we define the functions in theorem T4 as DS, the CS version of the definitions of $half$ and $\leq$ are still required to ripple out wavefronts ‘generated’ by the definition of $+$. We do not include these particular kind of CS lemmas in the reported results, because, in theory, all the ones used here could be automatically generated from the DS definitions by replacing type destructors with type constructors.

11.4 Analysis

In this section we assess to what extent the test set results support or refute our hypotheses. Recall that these were:

1. The induction strategy works as described, automatically generating induction

Theorem	Plan	Time (sec)	Alt. Strategy	Spec.	Lemmas
T1C	No	—	—	—	L5, L6a*, L7a
T2C	Yes	1	—	2	L8
T3C	Yes	10	—	2	L2, L3, L4
T4C	Yes	10	—	2	L2, L3, L4
T5C	No	—	—	—	L2, L3, L11, L13, L14
T6C	Yes	55	—	2	
T7C	Yes	26	Lemma/Nest	2	L1, L2, L3
T8C	Yes	26	Lemma/Nest	2	L2, L3
T9C	Yes	7	—	1	L17
T10C	Yes	7	—	2	
T11C	Yes	21	—	2	L16, L19, L23
T12C	Yes	23	—	2	L8, L16, L19, L24
T13C	Yes	30	—	2	L16, L24
T14C	No	—	—	—	
T15C	Yes	6	Lemma	1	L28*
T16C	Yes	4	Limit	1	L25, L28, L31
T17C	Yes	4	Limit	1	L25, L27, L31
T18C	No	—	—	—	
T19C	No	—	—	—	

Table 11.5: Constructor style test results. 14 of 19 theorems were planned, 9 with the default strategy. * = lemma considered first by rewriting.

Theorem	Plan	Time (sec)	Alt. Strategy	Spec.	Lemmas
T1D	No	—	—	—	L5, L6b*, L7b
T2D	No	—	—	—	
T3D	Yes	15	—	2	L2, L3, L4, L9, L10
T4D	Yes	16	—	2	L2, L3, L4, L9, L10
T5D	No	—	—	—	L2, L3, L12, L13, L14
T6D	Yes	159	—	2	
T7D	Yes	41	Nest	2	L2, L3, L4
T8D	Yes	40	Nest	2	L2, L3, L4
T9D	Yes	10	—	1	L17, L32
T10D	Yes	15	—	2	
T11D	Yes	114	—	2	L16, L20, L23
T12D	Yes	113	—	2	L8, L16, L20, L24
T13D	Yes	143	—	2	L16, L24
T14D	No	—	—	—	
T15D	Yes	4	Lemma	1	L30*
T16D	Yes	5	—	1	L26, L30, L31
T17D	Yes	5	—	1	L26, L29, L31
T18D	No	—	—	—	
T19D	No	—	—	—	

Table 11.6: Destructor style test results. 13 of 19 theorems were planned, 10 with the default strategy. * = lemma considered first by rewriting.

Lemma	Statement
L1	$x \neq 0 \rightarrow p(s(x)) = s(p(x))$
L2	$x + 0 = 0$
L3	$x + s(y) = s(x + y)$
L4	$y \neq 0 \rightarrow x + y = s(x + p(y))$
L5	$u = x \wedge v = y \rightarrow u + v = x + y$
L6a	$x \times (y + z) = (x \times y) + (x \times z)$
L6b	$x \times y = ((x - z) \times y) + (y \times z)$
L7a	$(x + z) \times y = (x \times y) + (y \times z)$
L7b	$x \times y = (x \times (y - z)) + (x \times z)$
L8	$x \leq y \rightarrow x \leq s(y)$
L9	$half(x) \leq x$
L10	$half(s(x)) \leq x$
L11	$half((n + m) + m) = half(n) + m$
L12	$(m + m) \leq n \rightarrow half(n) = half((n - m) - m) + m$
L13	$quot(0, y) = y$
L14	$half(quot(s(0))) = 0$

Table 11.7: Arithmetic lemmas.

Lemma	Statement
L15a	$sum(h :: t,x) = sum(t,x) + h$
L15b	$l \neq nil \rightarrow sum(l,x) = sum(tail(l),x) + head(l)$
L16	$(l <> nil) = l$
L17	$(l <> m) <> n = l <> (m <> n)$
L18	$nil \neq l <> x :: nil$
L19	$len(l <> x :: m) = s(len(l <> m))$
L20	$m \neq nil \rightarrow len(l <> m) = s(len(l <> tail(m)))$
L21	$y \neq 0 \rightarrow last(oapp(x,y)) = last(y)$
L22	$bigger(x, smaller(y,l)) = smaller(bigger(x,l))$
L23	$len(evenelems(x :: l)) \leq len(l)$
L24	$len(oddelems(x :: l)) \leq s(len(l))$
L25	$rev(l <> x :: nil) = x :: rev(l)$
L26	$l \neq nil \rightarrow rev(l) = last(l) :: rev(chop(l))$
L27	$foldright\_tr(f,x,l <> y :: nil) = f(y, foldright\_tr(f,x,l))$
L28	$foldleft\_tr(f,x,l <> y :: nil) = f(foldleft\_tr(f,x,l),y)$
L29	$l \neq nil \rightarrow foldright\_tr(f,x,l) = f(last(l), foldleft\_tr(f,x,chop(l)))$
L30	$l \neq nil \rightarrow foldleft\_tr(f,x,l) = f(foldleft\_tr(f,x,chop(l)), last(l))$
L31	$x = u \wedge y = v \rightarrow x \circ y = u \circ v$
L32	$head(l) :: tail(l) = l$

Table 11.8: List and folding lemmas.

rules to plan proofs for a range of theorems not solved by recursion analysis.

2. There are theorems the strategy proves using non-destructor induction that cannot be proved by destructor-only lazy induction.

3. Theorems proved by lazy induction can also be proved by the strategy.

Overall the system planned 27 of the 38 test theorems, but only 19 of these were with the default strategy, i.e. fully automatically. Recall that the default is different for constructor and destructor style problems: constructor problems use the critic-based strategy.

Evaluating the hypotheses involves assessing the strategy based on the performance of its implementation in *Dynamis*. Hence we need to examine *why* the system failed completely on 11 theorems, and required human intervention on a further 8.

We can classify these failures of the default strategies into six categories:

- Failure of the wellfoundedness proof (T1C/D).
- Failure to generate missing cases (T5C).
- Divergent applications of the speculation critic (T7C, T8C).
- Lack of case splitting during rewriting (T7D, T8D).
- Failure due to runtime errors (T15C/D, T16C, T17C).
- The need for multiple step case/hypotheses (T14C/D, T18C/D, T19C/D).

Theorems T2D and T5D also failed, for which there is no clear explanation other than the inadequacy of our strategy. We discuss the various categories of failure below.

11.4.1 Non-wellfounded step cases

The default strategies successfully constructs the step case for theorems T1C and T1D. If we give the system lemmas L6a and L7a this step case is the constructor style:

$$\Phi(x) \quad \vdash \quad \Phi(x+y)$$

If instead we use lemmas L6b and L7b we get the destructor style:

$$\Phi(x-y) \quad \vdash \quad \Phi(x)$$

In both cases the step case is not wellfounded, as y may be zero. Consequently, the estimation strategy produces the following unsolvable difference equivalent:

$$\vdash \quad y \neq 0$$

Dynamis tries to apply rewriting, and the planning attempt fails.

This problem could be overcome by implementing the side condition critic proposed in §6.5.5, which is not currently part of *Dynamis*'s wellfoundedness strategy. The critic would respond to such unsolved difference equivalents by adding them to the conditions on the step case. In the case of T1, the critic would have made the step case wellfounded, and the theorem would have been planned successfully.

11.4.2 Failure to generate missing cases

For one theorem (T5C) planning failed at the exhaustive cases proof, after the following a wellfounded step case had been generated:

$$y \neq 0,\, \Phi(x,y) \quad \vdash \quad \Phi((x+y)+y,y)$$

The case strategy fails to find missing cases to complete the induction rule.

11.4.3 Divergent speculation critiques

Theorems T7C and T8C are the only two examples to expose a weakness of the speculation critic: repeated applications can sometimes diverge. The theorems are very similar, and essentially the same problem arises in both. We illustrate this with theorem T7C.

The step case is blocked immediately after the the first speculative ripple, with the following conclusion (meta-variables are shown as x, y etc. for simplicity):

$$\boxed{s(\underline{x'})}^{\uparrow} \neq 0 \rightarrow \quad odd(\boxed{s(\underline{x'+y})}^{\uparrow}) \leftrightarrow \neg odd(y + p(\boxed{s(\underline{x'})}^{\uparrow})) \tag{11.1}$$

The speculation critic is applied to unblock the ripple proof, and it succeeds following the definition of odd. A wavefront is inserted above $x'+y$ and rippled inwards to suggest an instantiation. As a result either x' or y is instantiated — the x' branch fails, so y is chosen. The post-critic conclusion ripples to:

$$\boxed{s(\underline{x'})}^{\uparrow} \neq 0 \rightarrow \quad odd(x'+y') \leftrightarrow \neg odd(\boxed{s(\underline{y' + p(\boxed{s(\underline{x'})}^{\uparrow}))}}^{\uparrow})$$

Rippling is blocked once again, and the speculation critic succeeds again with the definition of odd, this time on the right-hand side. The critic instantiates the meta-variable y' and the post-critic conclusion ripples to:

$$\boxed{s(\underline{x'})}^{\uparrow} \neq 0 \rightarrow \quad odd(\boxed{s(\underline{x'+y''})}^{\uparrow}) \leftrightarrow \neg odd(y'' + p(\boxed{s(\underline{x'})}^{\uparrow}))$$

This goal is of the same form as the original blocked goal (11.1), and so the ripple proof continues indefinitely, cycling through these two applications of the speculation critic.

This problem was avoided by prioritising the lemma L3, which is then used for the initial speculation instead of the definition of $+$. However, the proof still fails, because of the case split problem described in the next section.

11.4.4 No case splitting

Dynamis successfully constructs wellfounded step cases, and generates base cases for theorems T7D and T8D, and theorems T7C and T8C with prioritised lemma L3 (discussed in the previous section). However, in all these examples it fails to plan the base cases. The problematic base case goals are all of the form:

$$x \neq 0 \rightarrow \quad odd(x) \leftrightarrow \neg odd(p(x)) \tag{11.2}$$

Dynamis tries schematic induction. After the initial speculative ripple, the conclusion is as follows:

$$\boxed{s(s(\underline{x'}))}^{\uparrow} \neq 0 \rightarrow \quad odd(x') \leftrightarrow \neg odd(p(\boxed{s(s(\underline{x'}))}^{\uparrow}))$$

This is blocked, as the wavefronts cannot be rippled past the term $p(\ldots)$, and weak fertilisation is blocked because of the implication. No further progress can be made.

Interestingly, the standard λ*Clam* induction methods can solve this base case. The key difference is that when a ripple proof fails without fertilisation, the goal is passed directly to rewriting. This simplifies our blocked goal to:

$$odd(x') \leftrightarrow \neg odd(s(x')) \tag{11.3}$$

This is easily proved by induction.

By allowing induction without fertilisation, λ*Clam* is essentially performing a case split on (11.2), suggested by the definition of odd. Whether this is a sensible strategy in general is questionable — if the abandoned induction cannot be simplified by rewriting, but if induction is applicable again, the strategy could diverge — but the ability to apply case splits is clearly useful.

A safer approach in general would be to apply a case split during rewriting, provided each case can be reduced. For the goal (11.2) we could split over the definition of p, and we would end up with the solvable subgoal (11.3).

Dynamis can plan the theorems T7 and T8 with a strategy that uses λ*Clam* to perform nested inductions, taking advantage of the ability to case split that our implementation lacks.

11.4.5 λProlog errors

Theorems T16C and T17C both fail because of runtime errors. The following error message is given in both cases:

```
Attempting atomic critic speculation_critic
Access to unmapped memory!
/home/jeremy/dynamis//bin/dynamis: line 9:  1095 Aborted
$TEYJUS_HOME/tjsim --solve "lclam." ${1:-dynamis}
```

This is clearly a memory problem with the underlying λProlog implementation, and needs to be investigated further. The theorems are planned successfully without the critic, using `dynamis_lim2`.

The plan search for theorem T15 begins with a single speculative step using the definition of *foldleft_tr*. In order to find a proof, the planner needs to backtrack over this choice, and speculate with the lemma L28/L30 for a constructor/destructor style induction. However, the initial search branch is never completed — a runtime memory error occurs after about 15 minutes of search — and this backtrack never occurs. It is not clear whether this search branch was divergent, or whether it would have bottomed-out and allowed the backtrack. A plan is found if the lemma L28 or L30 is given priority over the definitions.

Clues to what might happen without the runtime error can be found by looking at the post-fertilisation goals produced during the search. The first such goal is:

$$(x \circ y) \circ \mathit{foldleft\_tr}(\circ, id, l) \quad = \quad x \circ \mathit{foldleft\_tr}(\circ, id \circ y, l) \tag{11.4}$$

Dynamis performs a nested induction, again speculating with the definition of *foldleft_tr*. The post-fertilisation goal is:

$$\begin{aligned}&(x \circ y) \circ \mathit{foldleft}_t r(\circ, (id \circ z), l) \\ &= x \circ \mathit{foldleft\_tr}(\circ, ((id \circ y) \circ z), l) \qquad (11.5)\end{aligned}$$

The proof of this subgoal fails, but identical goals are produced repeatedly on backtracking — a process which accounts for the majority of the runtime. Eventually, the system produces a different post-fertilsation goal for the nested induction:

$$\begin{aligned}&((x \circ y) \circ \mathit{foldleft}_t r(\circ, (id \circ z), l)) \circ w \\ &= (x \circ \mathit{foldleft\_tr}(\circ, ((id \circ y) \circ z), l)) \circ w \qquad (11.6)\end{aligned}$$

We conjecture that as the subgoal (11.5) failed, the subgoal (11.6) will also fail, forcing the system to backtrack over (11.4), and hence apply the successful speculation step. However, we will need to address the memory error before we can establish if this is actually what will happen.

11.4.6 Multiple step cases/hypotheses required

Recall that we did not expect a plan to be found for theorems T14, T18 and T19, as they require multiple induction hypotheses and step cases. They were included to see whether *Dynamis* made as much progress as could be expected under its current restriction of a single step case/induction hypothesis.

The closest *Dynamis* gets to a proof plan for T14 with constructor style definitions is the following goal (with metavariables written as constants for simplicity):

$$\begin{array}{l} y \neq x, \quad x \neq y, \\ \mathit{perm}(\boxed{\mathit{delete}(a,\underline{u})}\,,v) \leftrightarrow \mathit{perm}(v,\mathit{delete}(a,u)) \\ \vdash \boxed{y \in (x :: m) \wedge \underline{\mathit{perm}(l, \boxed{x :: \mathit{delete}(y,\underline{m})}^{\downarrow}\,)}}^{\uparrow} \\ \quad \leftrightarrow \quad \boxed{x \in (y :: l) \wedge \underline{\mathit{perm}(m, \boxed{y :: \underline{\mathit{delete}(x,l)}}^{\uparrow}\,)}}^{\uparrow} \end{array}$$

Here *Dynamis* has failed to neutralise the context around *delete* on the left side. This is most likely a deficiency in the design or implementation of the neutralisation algorithm, and needs to be investigated further. If this had been done, the next stage of the proof from [Protzen, 1995] involves generating and weak fertilising with two new inductive hypotheses — one on each side of the iff — and continuing to ripple towards the initial inductive hypothesis. Hence *Dynamis* has clearly made as much progress as we can expect with its current single induction hypothesis restriction.

11.5 Conclusions of the Evaluation

Having looked at the reasons for the failure of the unassisted default strategy on 19 of the 38 test theorems (with complete failure for 11), we can assess how many of these failures can be attributed to our induction strategy, and how many to problems with its implementation.

To summarise, the following the shortcomings were found with the implementation:

1. The proposed side condition critic is not implemented.

2. Case splits are not performed during rewriting.

3. It can only generate a single step case with a single induction hypothesis.

4. Runtime memory errors occur.

5. Not all wavefronts that could be neutralised are removed.

The first three problems were known about before the evaluation, whereas (4) is an unanticipated problem with Teyjus λProlog, and (5) is a deficiency in the implementation of neutralisation. These shortcomings account for 19 of the 18 failures.

The remaining 5 failures — T2D, T5C/D, T7C and T8C — can be attributed to shortcomings of our induction strategy. Two of these examples have uncovered the potential for divergent applications of the speculation critic.

The purpose of the evaluation was to provide evidence for or against our three hypotheses. The first was that our induction strategy performed the task it was designed to, i.e. prove theorems not solvable with recursion analysis. The evidence of the test theorems supports this.

The second hypothesis was that our strategy proves theorems by non-destructor induction that destructor-only lazy induction cannot prove. Considering the constructor style examples, we have provided a collection of such theorems. It should be noted that many of these could be proved by lazy induction, given that translating the definitions to destructor style could be done automatically. However, this would not account for the theorems T1C, T15C, T16C and T17C, where a lemma (e.g. L27) was used to generate the constructor style induction. Lazy induction could not find these inductions with the given lemma, but would require a different lemma, e.g. L29, involving different functions, e.g. $chop$ and $last$ instead of $<>$. This was discussed in Chapter 3.

Furthermore, theorem T2 seems to have no satisfactory destructor style induction, and we cannot see how lazy induction could solve it, whereas the constructor style proof was successful. Hence, the evidence of the test theorems supports the second hypothesis, with the proviso that the lemmas supplied are a significant factor in the success of our approach.

Considering the third hypothesis, T5D, T14D, T18D, T19D are the only theorems cited as successes for lazy induction over recursion analysis in [Protzen, 1995]. None were planned by *Dynamis*. We have accounted for the failure of all but T5D, but further evaluation is required to verify whether our suggestions for overcoming these failures actually work. Even so, it would be a very small set of examples on which to base a comparison. The third hypothesis — that our strategy is strictly more successful than lazy induction — is not supported by the evaluation. Ideally, a reimplementaion of lazy induction in λ*Clam* would be used for a direct comparison over a larger problem set. It should be noted that for T5D and T14D we could not reconstruct from [Protzen, 1995] how the proofs were automatically found, and so we cannot account for why lazy induction performs better than our system on these examples.

Chapter 12

Case Studies

In this chapter we provide a collection of examples of the *Dynamis* system in action, constructing proof plans for theorems from Chapter 11. For clarity, the system traces are abridged and interspersed with explanatory comments. The abridgement omits a large amount of system output, but gives an accurate and readable presentation of the system's search for a proof plan.

We consider three examples which illustrate the techniques outlined in this thesis, and the range of the *Dynamis* system:

1. T6C (see §12.2), a constructor style problem which requires multiple speculations. *Dynamis* uses the speculation critic to justify the second speculative step.

2. T9D (see §12.3), a destructor style problem, for which recursion analysis chooses the wrong induction variable, but *Dynamis* creates the correct one. This also illustrates the use of creational rippling, where the conclusion is rewritten to match wave fronts in the induction hypothesis.

3. T16C (see §12.4), a constructor style problem which requires an induction with a non-trivial case structure.

12.1 Presenting *Dynamis* Output

As mentioned above, a number of changes have been made to the system output to make it shorter and more readable. This involved removing a large amount of λ*Clam* trace messages (e.g. "Attempting... dynamis_crit"), plan information (e.g. the address of the current node in plan tree) and some heuristic information (e.g. the wave measure of annotated goals). The layout of goals and formulae has also been tidied up. *Dynamis* displays embeddings as wave annotation, for example `[[...\\...//]](+)`. For clarity, this has been changed below to the standard rippling box-and-hole notation.

Renaming Variables

Another presentational change is the renaming of variables and meta-variables. λ*Clam* displays variables as λProlog constants: `<lc-0-1>`, `<lc-0-2>` etc. We have renamed these to more recognisable lowercase letters e.g. `x`, `y` etc.

More importantly, λ*Clam* displays meta-variables without consistent names – a shortcoming of Teyjus λProlog. In other words, two occurrences the same meta-variable may be displayed differently. *Dynamis* improves slightly on this by naming consistently within formulae. It displays them as constants surrounded by curly brackets e.g. {`<lc-0-1>`}. We have renamed meta-variables by hand with a unique name for each meta-variable. Uppercase letters are used, e.g. `A`, `B` etc. If a meta-variable `A` is partially instantiated then the meta-variables in its instantiater will be named `A′`, `A′′` etc. As a consequence of this, it is not clear below what variables a meta-variable is dependant on — but this is not essential to understanding the traces.

Pretty Printing

There are also differences which should be noted between pretty printing in λ*Clam* and *Dynamis*. Standard sequent goals (e.g. the root goal) are displayed by the original λ*Clam* code. Other meta-level goals introduced in this thesis, such as the schematic step case, are displayed using new *Dynamis* code. *Dynamis* tends to do more pretty printing, in order to reduce the size of goals. For instance, it displays functions like $plus$ and $oapp$ as $+$ and $<>$ in the examples below.

12.2 Case Study T6C: Speculation

Theorem T6 is stated as follows:

$$\forall x,y,z{:}nat.\, even(x+y) \wedge even(y+z) \rightarrow even(x+z)$$

In test T6C, both *even* and $+$ (sometimes displayed as $plus$) have constructor style definitions, and no lemmas were provided to the system.

This problem needs multiple speculation steps, as two such steps with the definition of $plus$ are required, in order to create the two wave fronts that can be rippled by the definition of *even*. Running *Dynamis* on this example illustrates the use of the speculation critic, where the second speculation is applied as a patch to the definite ripple method.

T6C: Initial Planning

We begin planning T6C with `dynamis_crit`, the default constructor style method. *Dynamis* loads the necessary function and datatype definitions:

```
This is Lambda-CLAM v4.0.0
Copyright (C) 2002 Mathematical Reasoning Group, University of Edinburgh
```

```
NOTE: this program uses the MRG patched version of Teyjus 1.0-b33.

lclam:
dynamis_plan dynamis_crit evenptrans 1 constructor.

Functions: zero :: s :: plus :: even :: nil

Eval Lemmas: nil
Wave Lemmas: nil

Loading Eval Rules: idty :: s_functional :: neq_s_zero :: neq_zero_s ::
   plus1 :: plus2 :: even1 :: even2 :: even3 :: nil
Loading Wave Rules: s_functional :: plus2 :: even3 :: nil

Planning:
evenptrans
>>> forall x:nat forall y:nat forall z:nat
       ((even plus (x, y) /\ even plus (y, z)) -> even plus (x, z))
```

Expanding `dynamis_crit` to `dynamis_main`, `schematic_induction` is then applied to the top-level goal.

```
Method application:  dynamis_crit
Method application:  dynamis_main (mo_step_case spec_critic_ripple) wellfound_strat
                                  case_strat (waterfall dynamis_crit)
Method application:  schematic_induction ...
```

A conjunction of five subgoals is produced: 1) the schematic step case:

```
allGoal nat (x\ allGoal nat (y\ allGoal nat (z\
caseSchema
A
(((even (B + C)) /\ (even (C + D))) -> (even (B + D)))
>>> (((even (E + F)) /\ (even (F + G))) -> (even (E + G))))))
```

2) the wellfounded step case goal:

```
** stepReduces
```

3) the exhaustive cases goal:

```
** allGoal tuple_type (nat :: nat :: nat :: nil) (u\
existsGoal (x\ existsGoal (y\ existsGoal (z\
caseGoal
Case: (trueP, u)
>>> (A /\ (u = E F G))))))
```

4) the unknown cases:

```
** maybeCases
```

5) the wellfounded rule goal:

```
** wfGoal
```

T6C: Step Case Plan

After splitting the conjunction and moving the goal quantification into the proof plan, the step case goal is considered:

```
caseSchema
A
(((even (B + C)) /\ (even (C + D))) -> (even (B + D)))
>>> (((even (E + F)) /\ (even (F + G))) -> (even (E + G)))
```

A single speculative ripple is tried to start the proof off, succeeding with the definition of *plus*. This creates two wave fronts in the conclusion, with the leftmost rippled out by this step:

```
Method application:  mo_step_case spec_critic_ripple
Method application:  embed_hypothesis
Method application:  spec_critic_ripple
Method application:  speculative_ripple plus2 (2 :: 1 :: 2 :: 1 :: 2 :: nil)

caseSchema
A
(((even (B + C)) /\ (even (C + D))) -> (even (B + D)))
```

$$\texttt{>>> (((even }\boxed{(\mathrm{s}\ \underline{(\mathrm{E}'+\mathrm{F})})}^{\uparrow}\texttt{) /\textbackslash{} (even (F + G))) -> (even (}\boxed{(\mathrm{s}\ \underline{\mathrm{E}'})}^{\uparrow}\texttt{ + G)))}$$

The rightmost wave front is now rippled out with the definition of *plus* $(+)$:

```
Method application:  patch_meth (definite_ripple plus2 (2 :: 2 :: 2 :: nil)) speculation_critic

caseSchema
A
(((even (B + C)) /\ (even (C + D))) -> (even (B + D)))
```

$$\texttt{>>> (((even }\boxed{(\mathrm{s}\ \underline{(\mathrm{E}'+\mathrm{F})})}^{\uparrow}\texttt{) /\textbackslash{} (even (F + G))) -> (even }\boxed{(\mathrm{s}\ \underline{(\mathrm{E}'+\mathrm{G})})}^{\uparrow}\texttt{))}$$

Rippling is blocked, but the failure of `definite_ripple` suggests the application of `speculation_critic`. The critic identifies missing wave fronts below the leftmost wave front — if inserted into the goal they would allow the definition of *even* to be applied. It inserts them, then attempts to ripple them inwards to find a suitable instantiation. This is successful:

```
Method application:  patch_meth (definite_ripple _273629 _273643) speculation_critic
speculation_critic succeeded

caseSchema
A
(((even (B + C)) /\ (even (C + D))) -> (even (B + D)))
```

>>> (((even (s $\boxed{(\mathtt{s}\ \underline{(\mathtt{E}'+\mathtt{F})})}^{\downarrow}$)) /\ (even (F + G))) -> (even (s (E′ + G))))

```
Method application:  ripple_in_and_speculate _
Method application:  forwards_ripple plus2 (2 :: 2 :: 1 :: 2 :: 1 :: 2 :: nil)
                                (definite_ripple plus2 (2 :: 2 :: 1 :: 2 :: 1 :: 2 :: nil) :: _)

caseSchema
A
(((even (B + C)) /\ (even (C + D))) -> (even (B + D)))
```

>>> (((even (s ($\boxed{(\mathtt{s}\ \underline{\mathtt{E}'})}^{\downarrow}$ + F))) /\ (even (F + G))) -> (even (s (E′ + G))))

```
Method application:  speculate_wavefronts (definite_ripple plus2 (2 :: 2 :: 1 :: 2 :: 1 :: 2 :: nil) :: _)
                                    (definite_ripple plus2 (2 :: 2 :: 1 :: 2 :: 1 :: 2 :: nil))

trueGoal!
Branch closed!
```

The speculation critic has found an instantiation and ripple patch that allows the definition of *even* to apply. It now applies this patch and the *even* ripple:

```
Method application:  ripple_patch (definite_ripple plus2 (2 :: 2 :: 1 :: 2 :: 1 :: 2 :: nil))
                              even3 (1 :: 2 :: 1 :: 2 :: nil) _
Method application:  redo_embeddings
Method application:  definite_ripple plus2 (2 :: 2 :: 1 :: 2 :: 1 :: 2 :: nil)
Method application:  definite_ripple even3 (1 :: 2 :: 1 :: 2 :: nil)

caseSchema
A
(((even (B + C)) /\ (even (C + D))) -> (even (B + D)))
```

>>> (((even (E′′ + F)) /\ (even (F + G))) -> (even $\boxed{(\mathtt{s}\ \underline{(\boxed{(\mathtt{s}\ \underline{\mathtt{E}''})}^{\uparrow}+\mathtt{G})})}^{\uparrow}$))

A side effect of the speculation critic's patch was to create another wave front on the right-hand side, which is now rippled out in two steps:

```
Method application:  patch_meth (definite_ripple plus2 (2 :: 2 :: 2 :: 2 :: nil)) speculation_critic

caseSchema
A
(((even (B + C)) /\ (even (C + D))) -> (even (B + D)))
```

>>> (((even (E′′ + F)) /\ (even (F + G))) -> (even $\boxed{(\mathtt{s}\underline{\boxed{(\mathtt{s}\underline{(\mathtt{E}''+\mathtt{G})})}^{\uparrow}\ )}}^{\uparrow}$))

```
Method application:  patch_meth (definite_ripple even3 (2 :: 2 :: nil)) speculation_critic

caseSchema
A
(((even (B + C)) /\ (even (C + D))) -> (even (B + D)))
>>> (((even (E’’ + F)) /\ (even (F + G))) -> (even (E’’ + G)))
```

Strong fertilisation completes the step case plan:

```
Method application:  mo_fertilise
Method application:  strong_fertilise

trueGoal!
Branch closed!
```

T6C: Wellfounded Step Case Plan

The planner returns to the four remaining goals:

```
stepReduces
   **
allGoal tuple_type (nat :: nat :: nat :: nil) (u\
existsGoal (x\ existsGoal (y\ existsGoal (z\
caseGoal
Case: (trueP, u)
>>>
(A /\ (u = (s (s E’’)) F G))))))
   **
maybeCases
   **
wfGoal
```

The next to be planned is the wellfounded step case goal. Having determined the form of the step case above, the `construct_wf_goals` method is now applied to explicitly construct its wellfoundedness goal, using the unknown wellfounded relation `H`:

```
stepReduces
Method application:  wellfound_strat
Method application:  construct_wf_goals (const_disj (measure 3 _ :: measure 2 _ :: measure 1 _ :: nil) :: _)

allGoal nat (x\ allGoal nat (y\ allGoal nat (z\
redGoal 1
>>> A -> H (E’’, (s (s E’’)))
   **
redGoal 2
>>> A -> H (F, F)
   **
redGoal 3
>>> A -> H (G, G))))
```

This three-part conjunction actually represents a disjunction, in that only one of the three induction positions *needs* to be proved to show the step case is wellfounded. However, in order to build up constraints on all the induction positions a proof of each subgoal is attempted, with failed subgoals being 'proved' by the `ignore_position` method.

Considering the first goal, the estimation strategy is applied. This produces two subgoals: the first states that some unknown difference equivalent `I` holds iff this induction position reduces under an unknown measure `J`; the second states that difference equivalent holds:

```
redGoal 1
>>> A -> H (E'', (s (s E'')))

Method application:  estimation_strat
Method application:  begin_estimation 1

estGoal
I <-> J(E'') < J((s (s E'')))
   **
>>> (trueP -> I)
```

The plan of both subgoals is straightforward, instantiating the measure function to the identity:

```
estGoal
I <-> J(E'') < J((s (s E'')))

Method application:  lower_estimate

estGoal
I' <-> id(E'') < id((s E''))

Method application:  lower_estimate

estGoal
I'' <-> id(E'') < id(E'')

Method application:  trivial_estimate

trueGoal!
Branch closed!

>>> (trueP -> (trueP \/ (trueP \/ falseP)))

Method application:  rewrite_equiv nil

trueGoal!
Branch closed!
```

The other two wellfoundedness goals are unsolvable, but the system can ignore them both as the first goal has been planned:

```
redGoal 2
>>> A -> H (F, F)
   **
redGoal 3
>>> A -> H (G, G)

Method application:  ignore_position 2

trueGoal!
Branch closed!

Method application:  ignore_position 3

trueGoal!
Branch closed!
```

This completes the step case wellfoundedness plan.

T6C: Exhaustive Cases Plan

Returning to the three remaining subgoals, the next goal to be planned is the exhaustive cases goal:

```
allGoal tuple_type (nat :: nat :: nat :: nil) (u\
existsGoal (x\ existsGoal (y\ existsGoal (z\
caseGoal
Case: (trueP, u)
>>> (A /\ (u = (s (s E’’)) F G))))))
   **
maybeCases
   **
wfGoal

Method application:  case_strat
Method application:  set_conditions

caseGoal
Case: (trueP, u)
>>> (trueP /\ (u = (s (s E’’)) F G))
```

First the redundant `trueP` is removed and the universal variable is split, in order to separate the three elements of the tuple:

```
Method application:  case_equiv (and3 :: nil)
Method application:  case_induction (tuple_split 3)
Method application:  case_equiv (tuple_eq_rec :: tuple_eq_rec :: tuple_eq_base :: nil)

caseGoal
Case: (trueP,  p q r)
>>> ((p = (s (s E’’))) /\ ((q = F) /\ (r = G)))
```

The case strategy goes through a waterfall of methods, eventually trying structural induction:

```
Method application:  case_induction nat_struct

caseGoal
Case: (trueP,  zero q r)
>>> ((zero = (s (s E''))) /\ ((q = F) /\ (r = G)))
   **
allGoal nat (v\
caseGoal
Case: (trueP,  (s v) q r)
((v = (s (s E''))) /\ ((q = F) /\ (r = G)))
```

`>>> ((`$\boxed{(s\ \underline{v})}^{\uparrow}$` = (s (s E''))) /\ ((q = F) /\ (r = G))))`

The base case `(trueP, zero q r)`} is reduced to `falseP`, and included as a missing case:

```
caseGoal
Case: (trueP,  zero q r)
>>> ((zero = (s (s E''))) /\ ((q = F) /\ (r = G)))

Method application:  remove_case_hyps
Method application:  case_equiv (neq_zero_s :: and1 :: nil)

caseGoal
Case: (trueP,  zero q r)
>>> falseP

Method application:  missing_case (case_abs nat (x\ case_abs nat (y\
                                  case trueP (_ x y) (tuple (zero :: y :: x :: nil)))))

trueGoal!
Branch closed!
```

The step case is simplified so that one of the two constructors is removed. Induction is applied again to remove the remaining constructor:

```
Method application:  remove_case_hyps
Method application:  case_equiv (s_functional :: nil)

caseGoal
Case: (trueP,  (s v) q r)
>>> ((v = (s E'')) /\ ((q = F) /\ (r = G)))

Method application:  case_induction nat_struct

caseGoal
Case: (trueP,  (s zero) q r)
>>> ((zero = (s E'')) /\ ((q = F) /\ (r = G)))
   **
allGoal nat (w\
caseGoal
Case: (trueP,  (s (s w)) q r)
((w = (s E’’)) /\ ((q = F) /\ (r = G)))

>>> (((s w)↑  = (s E’’)) /\ ((q = F) /\ (r = G))))
```

Again the base case is false, and this time the case `(trueP, (s zero) q r)` is added as a missing case:

```
caseGoal
Case: (trueP,  (s zero) q r)
>>> ((zero = (s E’’)) /\ ((q = F) /\ (r = G)))

Method application:  remove_case_hyps
Method application:  case_equiv (neq_zero_s :: and1 :: nil)

caseGoal
Case: (trueP,  (s zero) q r)
>>> falseP

Method application:
missing_case (case_abs nat (x\ case_abs nat (y\ case trueP (_ x y) (tuple (app s zero :: y :: x :: nil)))))

trueGoal!
Branch closed!
```

The step case is trivial:

```
caseGoal
Case: (trueP,  (s (s w)) p r)
((w = (s E’’)) /\ ((p = F) /\ (r = G)))
>>> (((s w)↑  = (s E’’)) /\ ((p = F) /\ (r = G)))

trueGoal!
Branch closed!
```

T6C: Base Case Plans

Dynamis now explicitly constructs the missing cases identified above:

```
maybeCases ** wfGoal

maybeCases

Method application:  construct_cases

allGoal nat (y\ allGoal nat (x\
>>> ((even plus (0, x) /\ even plus (x, y)) -> even plus (0, y))))
   **
allGoal nat (y\ allGoal nat (x\
>>> ((even plus (s 0, x) /\ even plus (x, y)) -> even plus (s 0, y))))
```

The first base case is simplified by rewriting, then solved by a nested induction, which we omit here:

```
Method application:  waterfall dynamis_crit
Method application:  rewrite
Method application:  normalise all_i_nf
Method application:  rewrite_equiv (plus1 :: plus1 :: nil)

>>> ((even x /\ even plus (x, y)) -> even y)

Method application:  normalise all_e_nf
Method application:  dynamis_crit
Method application:  dynamis_main (mo_step_case spec_critic_ripple) wellfound_strat
                                  case_strat (waterfall dynamis_crit)
[...]
Branch closed!
```

The second base case is planned in a similar way:

```
allGoal nat (y\ allGoal nat (x\
>>> ((even plus (s 0, x) /\ even plus (x, y)) -> even plus (s 0, y))))

Method application:  waterfall dynamis_crit
Method application:  rewrite
Method application:  normalise all_i_nf
Method application:  rewrite_equiv (plus2 :: plus1 :: plus2 :: plus1 :: nil)

>>> ((even s x /\ even plus (x, y)) -> even s y)

Method application:  normalise all_e_nf
Method application:  dynamis_crit
Method application:  dynamis_main (mo_step_case spec_critic_ripple) wellfound_strat
                                  case_strat (waterfall dynamis_crit)
[...]
Branch closed!
```

T6C: Final Plan

As a final step, the system plans the meta-level goal `wfGoal` that represents the well-foundedness of the rule. The `wellfounded` method solves the constraints on the rule's relation, and instantiates it accordingly:

```
wfGoal

Method application:  wellfounded (app select_induce (tuple (app s zero :: id :: nil)))

trueGoal!
Branch closed!

reached empty agenda
Plan Succeeded
```

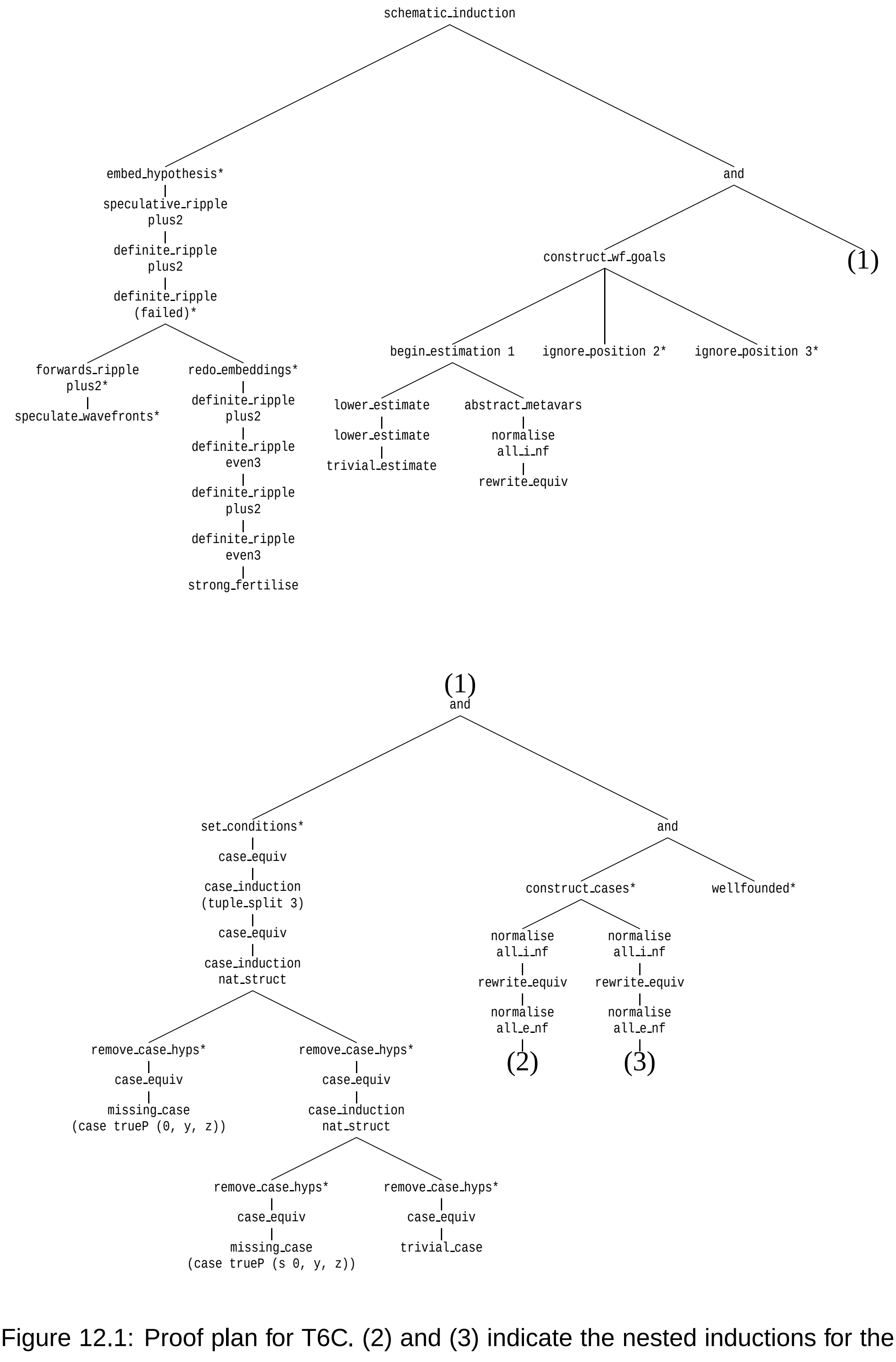

Figure 12.1: Proof plan for T6C. (2) and (3) indicate the nested inductions for the subgoals $even(x) \wedge even(x+y) \rightarrow even(y)$ and $even(s(x)) \wedge even(x+y) \rightarrow even(s(y))$. We omit these subplans. * indicates a purely meta-level plan step.

The proof plan for T6C is shown in Figure 12.1, with the subplans for the two nested inductions omitted. The plan has 129 nodes, including those subplans. *Dynamis* created and validated the following induction rule:

$$\frac{\begin{array}{rcl} & \vdash & \Phi(0,y,z) \\ & \vdash & \Phi(s(0),y,z) \\ \Phi(x,y,z) & \vdash & \Phi(s(s(x)),y,z) \end{array}}{\forall x{:}nat.\forall y{:}nat.\forall z{:}nat.\Phi(x,y,z)}$$

12.3 Case Study T9D: Destructor Style

In this section we give an example of *Dynamis* planning a destructor style problem. Theorem T9 is stated as follows:

$$\forall l,m{:}olist(nat).rotate(olength(l), l <> m) = m <> l$$

For test T9D the functions *rotate*, *olength* and <> (sometimes display *oapp*) have destructor style definitions. The following lemmas were provided:

$$\begin{array}{rcll} (x <> y) <> z & = & x <> (y <> z) & \text{(L17)} \\ head(l) :: tail(l) & = & l & \text{(L32)} \\ rotate(s(n), h :: t) & = & rotate(n, t <> (h :: nil)) & \\ (h :: t) <> l & = & h :: (t <> l) & \end{array}$$

The last two lemmas are constructor style definitions required for the destructor style proof.

T9D: Inital Planning

When planning is initiated, *Dynamis* first identifies the defining functions and lemmas pertaining to the theorem, then loads the appropriate symbolic evaluation and wave rules into the λ*Clam* database. It then begins planning with the top-level method `dynamis_lim1`, the default method for destructor style problems.

```
This is Lambda-CLAM v4.0.0
Copyright (C) 2002 Mathematical Reasoning Group, University of Edinburgh
NOTE: this program uses the MRG patched version of Teyjus 1.0-b33.

lclam:
dynamis_plan dynamis_lim1 gen_rotlen 1 destructor.

Functions: onil :: ocons :: tail :: zero :: head :: oapp :: s :: olength :: p :: rotate :: nil

Eval Lemmas: def rotate 3 :: oapp2 :: ass_app :: cons1 :: nil
Wave Lemmas: def rotate 3 :: oapp2 :: ass_app :: cons1 :: nil

Loading Eval Rules: idty :: cons_functional :: neq_nil_cons :: neq_cons_nil :: tail1 :: tail2 ::
   head1 :: head2 :: oapp1 :: s_functional :: neq_s_zero :: neq_zero_s :: olength1 :: p1 :: p2 ::
   def rotate 1 :: def rotate 2 :: def rotate 3 :: oapp2 :: ass_app :: cons1 :: nil
Loading Wave Rules: cons_functional :: tail2 :: head2 :: oapp3 :: s_functional :: olength3 ::
   p2 :: def rotate 4 :: def rotate 3 :: oapp2 :: ass_app :: cons1 :: nil

Planning:
gen_rotlen
>>> forall l:olist nat forall m:olist nat (rotate (olength l, oapp (l, m)) = oapp (m, l))
```

Dynamis expands the definition of `dynamis_lim1`, which gives `dynamis_main` parameterised by four methods. This in turn is expanded, and `schematic_induction` is applied to give a conjunction of five goals.

```
Method application:  dynamis_lim1
Method application:  dynamis_main (mo_step_case (n_spec_ripples 1)) wellfound_strat
                                  case_strat (waterfall (dynamis_lim 1))
Method application:  schematic_induction ...

allGoal olist nat (l\ allGoal olist nat (m\
caseSchema
A
((rotate  (olength B) (B <> C)) = (C <> B))
>>>
((rotate  (olength D) (D <> E)) = (E <> D))))
   **
stepReduces
   **
allGoal tuple_type (olist nat :: olist nat :: nil) (u\
existsGoal (l\ existsGoal (m\
caseGoal
Case: (trueP, u)
>>> (A /\ (u =  D E)))))
   **
maybeCases
   **
wfGoal
```

T9D: Step Case Plan

After splitting the goal conjunction and moving the goal quantification into the plan, the first goal is the schematic step case:

```
caseSchema
A
((rotate  (olength B) (B <> C)) = (C <> B))
>>> ((rotate  (olength D) (D <> E)) = (E <> D))
```

Planning the step case proceeds by embedding the induction hypothesis into the conclusion, and applying a single speculative ripple step with the definition of *olength* — only one such step is allowed by the step case method used here. This step instantiates the induction hypothesis, creating three wavefronts in this hypothesis, one of which is neutralised immediately. The step produces two subgoals, the first of which is a side condition.

```
Method application:  mo_step_case (n_spec_ripples 1)
Method application:  embed_hypothesis
Method application:  n_spec_ripples 1
Method application:  speculative_ripple olength3 (1 :: 2 :: 1 :: 2 :: nil)

sideCond
A
>>> (neg (D = onil))
   **
caseSchema
A
```

$$\texttt{((rotate (olength (tail B}'\texttt{)) (}\boxed{\texttt{(tail }\underline{\texttt{B}'}\texttt{)}}\texttt{ <> C)) = (C <> }\boxed{\texttt{(tail }\underline{\texttt{B}'}\texttt{)}}\texttt{))}$$

$$\texttt{>>> ((rotate }\boxed{\texttt{(s }\underline{\texttt{(olength (tail D)}}\texttt{))}}^{\uparrow}\texttt{ (D <> E)) = (E <> D))}$$

Dynamis first plans the side condition by applying the `simplify_sidecond` method. To discharge it the method assumes it as a condition on the step case, by instantiating `A` to `((neg (D = onil)) /\ A′)`:

```
sideCond
A
>>> (neg (D = onil))

Method application:  simplify_sidecond assume_cond

trueGoal!
Branch closed!
```

Returning to the main step case goal, three ripple steps are applied. First, a creational ripple with the definition of *oapp* (<>) which neutralises one of the two remaining hypothesis wavefronts:

```
Method application:  definite_rippling
Method application:  definite_ripple oapp3 (2 :: 2 :: 1 :: 2 :: nil)

caseSchema
((neg (D = onil)) /\ A′)
```

$$\texttt{((rotate (olength (tail B')) ((tail B') <> C)) = (C <> }\boxed{\texttt{(tail }\underline{\texttt{B'}}\texttt{)}}\texttt{))}$$

$$\texttt{>>> ((rotate }\boxed{\texttt{(s }\underline{\texttt{(olength (tail D))}}\texttt{)}}^{\uparrow}\ \boxed{\texttt{(ocons (head D) }\underline{\texttt{((tail D) <> E)}}\texttt{)}}^{\uparrow}\texttt{) = (E <> D))}$$

Next a ripple with the definition of *rotate*:

```
Method application:  definite_ripple (def rotate 3) (1 :: 2 :: nil)

caseSchema
((neg (D = onil)) /\ A′)
```

$$\texttt{((rotate (olength (tail B')) ((tail B') <> C)) = (C <> }\boxed{\texttt{(tail }\underline{\texttt{B'}}\texttt{)}}\texttt{))}$$

$$\texttt{>>> ((rotate (olength (tail D)) }\boxed{\texttt{(}\underline{\texttt{((tail D) <> E)}}\texttt{ <> (ocons (head D) onil))}}^{\uparrow}\texttt{) = (E <> D))}$$

And finally an inwards ripple with the associativity of *oapp* (<>):

```
Method application:  definite_ripple ass_app (2 :: 2 :: 1 :: 2 :: nil)

caseSchema
((neg (D = onil)) /\ A′)
```

$$\texttt{((rotate (olength (tail B')) ((tail B') <> C)) = (C <> }\boxed{\texttt{(tail }\underline{\texttt{B'}}\texttt{)}}\texttt{))}$$

$$\texttt{>>> ((rotate (olength (tail D)) ((tail D) <> }\boxed{\texttt{(}\underline{\texttt{E}}\texttt{ <> (ocons (head D) onil))}}^{\downarrow}\texttt{)) = (E <> D))}$$

Rippling is now blocked, and weak fertilisation is applied:

```
Method application:  mo_fertilise

Method application:  weak_fertilise 0

caseSchema
((neg (D = onil)) /\ A′)
>>> (((E <> (ocons  (head D) onil)) <> (tail D)) = (E <> D))

Method application:  replace_metavariables

>>> forall a:olist nat forall b:olist nat
      (oapp (oapp (a, ocons (head b, onil)), tail b) = oapp (a, b))
```

Rewriting with the associativity of *oapp* and function and datatype definitions completes the step case plan:

```
Method application:  waterfall (dynamis_lim 1)
Method application:  rewrite
Method application:  normalise all_i_nf

>>> (oapp (oapp (a, ocons (head b, onil)), tail b) = oapp (a, b))

Method application:  rewrite_equiv (ass_app :: oapp2 :: oapp1 :: cons1 :: idty :: nil)

trueGoal!
Branch closed!
```

T9D: Wellfounded Step Case Plan

Dynamis now returns to the four remaining induction subgoals. Notice that the exhaustive cases subgoal has become instantiated with the condition ((neg (D = onil)) /\ A′) by the step case planning.

```
stepReduces
   **
allGoal tuple_type (olist nat :: olist nat :: nil) (u\
existsGoal (x\ existsGoal (y\
caseGoal
Case: (trueP, u)
>>> (((neg (D = onil)) /\ A′) /\ (u =  D E)))))
   **
maybeCases
   **
wfGoal
```

After splitting the goal conjunction, the system tries to plan the wellfoundedness goal. The construct_wf_goals transforms the dummy meta-level goal stepReduces into the wellfoundedness goals for the step case that has just been found. There are two such subgoals — one for each universal variable in the original conjecture:

```
stepReduces

Method application:  wellfound_strat
Method application:  construct_wf_goals (const_disj (measure 2 _ :: measure 1 _ :: nil) :: _)

allGoal olist nat (x\ allGoal olist nat (y\
redGoal 1
>>> ((neg (D = onil)) /\ A′) -> F ((tail D), D)
   **
redGoal 2
>>> ((neg (D = onil)) /\ A′) -> F ((E <> (ocons  (head D) onil)), E)))
```

Selecting the first wellfoundedness goal, *Dynamis* applies the estimation strategy. Initially, this gives a conjunction of subgoals: an estimation goal which states that some unknown difference equivalent G holds iff the induction terms reduce under some unknown measure H; and a goal that states the step case conditions imply this difference equivalent.

```
redGoal 1
>>>
((neg (D = onil)) /\ A') -> F ((tail D), D)

Method application:  estimation_strat
Method application:  begin_estimation 1

estGoal
G <-> H((tail D)) < H(D)
   **
>>> ((˜ (D = onil) /\ trueP) -> G)
```

This goal is planned by estimating the list destructor `tail`, which instantiates the measure to `olength`:

```
estGoal
G <-> H((tail D)) < H(D)

Method application:  upper_estimate

estGoal
G' <-> olength(D) < olength(D)

Method application:  trivial_estimate

trueGoal!
Branch closed!
```

Moving on to the next subgoal, we see the difference equivalent has been instantiated by the estimation planning. Replacing the meta-variables with universal constants, the goal is passed to `rewrite`, which discharges it with a tautology checker:

```
>>> ((˜ (D = onil) /\ trueP) -> (˜ (D = onil) \/ falseP))

Method application:  abstract_metavars

>>> ((˜ (z = onil) /\ trueP) -> (˜ (z = onil) \/ falseP))

Method application:  rewrite
Method application:  normalise all_i_nf
Method application:  rewrite_equiv nil

trueGoal!
Branch closed!
```

The step case's second wellfoundedness goal arises from an induction position that was used to sink a wavefront. As before, *Dynamis* applies the estimation strategy:

```
redGoal 2
>>> ((neg (D = onil)) /\ A′) -> F (E <> (ocons  (head D) onil), E)

Method application:  estimation_strat
Method application:  begin_estimation 2

estGoal
I <-> J((E <> (ocons  (head D) onil))) < J(E)
   **
>>> ((˜ (D = onil) /\ trueP) -> I)
```

The estimation plan fails, as `ocons` cannot be upper estimated using any measure function. On backtracking the `ignore_position` method is applied instead, completing the step case wellfoundedness plan:

```
estGoal
I <-> J((E <> (ocons  (head D) onil))) < J(E)

Method application:  ignore_position 2

trueGoal!
Branch closed!
```

T9D: Exhaustive Cases Plan

The system now considers the two remaining subgoals from the original five goal conjunction:

```
allGoal tuple_type (olist nat :: olist nat :: nil) (u\
existsGoal (x\ existsGoal (y\
caseGoal
Case: (trueP, u)
>>> (((neg (D = onil)) /\ A′) /\ (u =  D E)))))
   **
maybeCases
   **
wfGoal
```

Splitting the conjunction, the first goal is the exhaustive cases goal. The case strategy is applied. The `set_conditions` method instantiates `A′`, the remaining meta-variable part of the step case condition, to `trueP`:

```
allGoal tuple_type (olist nat :: olist nat :: nil) (u\
existsGoal (x\ existsGoal (y\
caseGoal
Case: (trueP, u)
>>> (((neg (D = onil)) /\ A') /\ (u =  D E)))))

Method application:  case_strat
Method application:  set_conditions

caseGoal
Case: (trueP, u)
>>> (((neg (D = onil)) /\ trueP) /\ (u =  D E))
```

After removing the redundant `trueP`,`(case_induction (tuple_split 2))` is applied to the universal variable `u`. This ‘induction’ is actually a case split which allows the tuple to be broken up into individual terms:

```
Method application:  case_equiv (and4 :: nil)
Method application:  case_induction (tuple_split 2)
Method application:  case_equiv (tuple_eq_rec :: tuple_eq_base :: nil)

caseGoal
Case: (trueP, p q)
>>> ((neg (D = onil)) /\ ((p = D) /\ (q = E)))
```

The system goes through the `case_strat` waterfall of methods, eventually succeeding with `exists_casesplit`. This constructs a disjunct for each of the two possible instantiations of the meta-variable `D`. Simplification then removes the first disjunct:

```
Method application:  exists_casesplit list_struct

caseGoal
Case: (trueP,  p q)
>>> (((neg (onil = onil)) /\ ((p = onil) /\ (q = E)))
     \/
     ((neg ((ocons K L) = onil)) /\ ((p = (ocons K L)) /\ (q = E))))

Method application:  case_equiv (idty :: neg1 :: and1 :: or3 :: neq_cons_nil :: neg2 :: and3 :: nil)

caseGoal
Case: (trueP, p q)
>>> ((p = (ocons K L)) /\ (q = E))
```

The system goes through the method waterfall again, this time succeeding with structural induction on `p`:

```
Method application:  case_induction list_struct

caseGoal
Case: (trueP, onil q)
```

```
>>> ((onil = (ocons K L)) /\ (q = E))
   **
allGoal nat (v\ allGoal olist nat (w\
caseGoal
Case: (trueP, (ocons v w) q)
((w = (ocons K L)) /\ (q = E))

>>> (( (ocons v w)  = (ocons K L)) /\ (q = E))))
```

In the base case, rewriting reduces the conclusion to `falseP`. The `missing_case` method adds the `(case trueP _ (tuple [onil, _]))` to the list of missing cases, completing the plan branch:

```
caseGoal
Case: (trueP, onil q)
>>> ((onil = (ocons K L)) /\ (q = E))

Method application:  remove_case_hyps
Method application:  case_equiv (neq_nil_cons :: and1 :: nil)

caseGoal
Case: (trueP, onil q)
>>> falseP

Method application:  missing_case (case_abs (olist nat) (w1\ case trueP (_ w1) (tuple (onil :: w1 :: nil))))

trueGoal!
Branch closed!
```

Rippling fails in the step case, but simplification completes the plan branch:

```
allGoal nat (v\ allGoal olist nat (w\
caseGoal
Case: (trueP, (ocons v w) q)
((w = (ocons K L)) /\ (q = E))
>>> (( (ocons v w)  = (ocons K L)) /\ (q = E))))

Method application:  remove_case_hyps
Method application:  case_equiv (cons_functional :: solve_eq :: nil)

caseGoal
Case: (trueP,  (ocons v w) q)
>>> trueP

Method application:  trivial_case

trueGoal!
Branch closed!
```

T9D: Base Case Plan

There are now only two subgoals remaining: the first representing the missing cases of the induction, and the second that the rule is wellfounded. For the first goal, the missing base case determined by the exhaustive cases plan above is explicitly constructed:

```
maybeCases ** wfGoal

maybeCases

Method application:  construct_cases

allGoal olist nat (l\
>>> (rotate (olength onil, oapp (onil, l)) = oapp (l, onil)))
```

The goal is passed to the rewriting/generalisation/induction waterfall, where it is simplified by rewriting:

```
Method application:  waterfall (dynamis_lim 1)
Method application:  rewrite
Method application:  normalise all_i_nf
Method application:  rewrite_equiv (olength1 :: def rotate 1 :: oapp1 :: nil)

>>> (l = oapp (l, onil))
```

The waterfall now applies a nested induction using the `dynamis_lim` method. We omit the induction here, but it succeeds in completing the base case plan:

```
Method application:  normalise all_e_nf

>>> forall l:olist nat (l = oapp (l, onil))

Method application:  dynamis_lim 1
Method application:  dynamis_main (mo_step_case (n_spec_ripples 1)) wellfound_strat
                                  case_strat (waterfall (dynamis_lim 1))
[...]
Branch closed!
```

T9D: Final Plan

The last remaining goal represents the wellfoundedness of the entire induction rule. The `wellfounded` method discharges this by solving the wellfoundedness constraints built up during planning. This instantiates the wellfounded relation to one induced by measuring the first induction term by *olength*. The proof plan for T9D is complete:

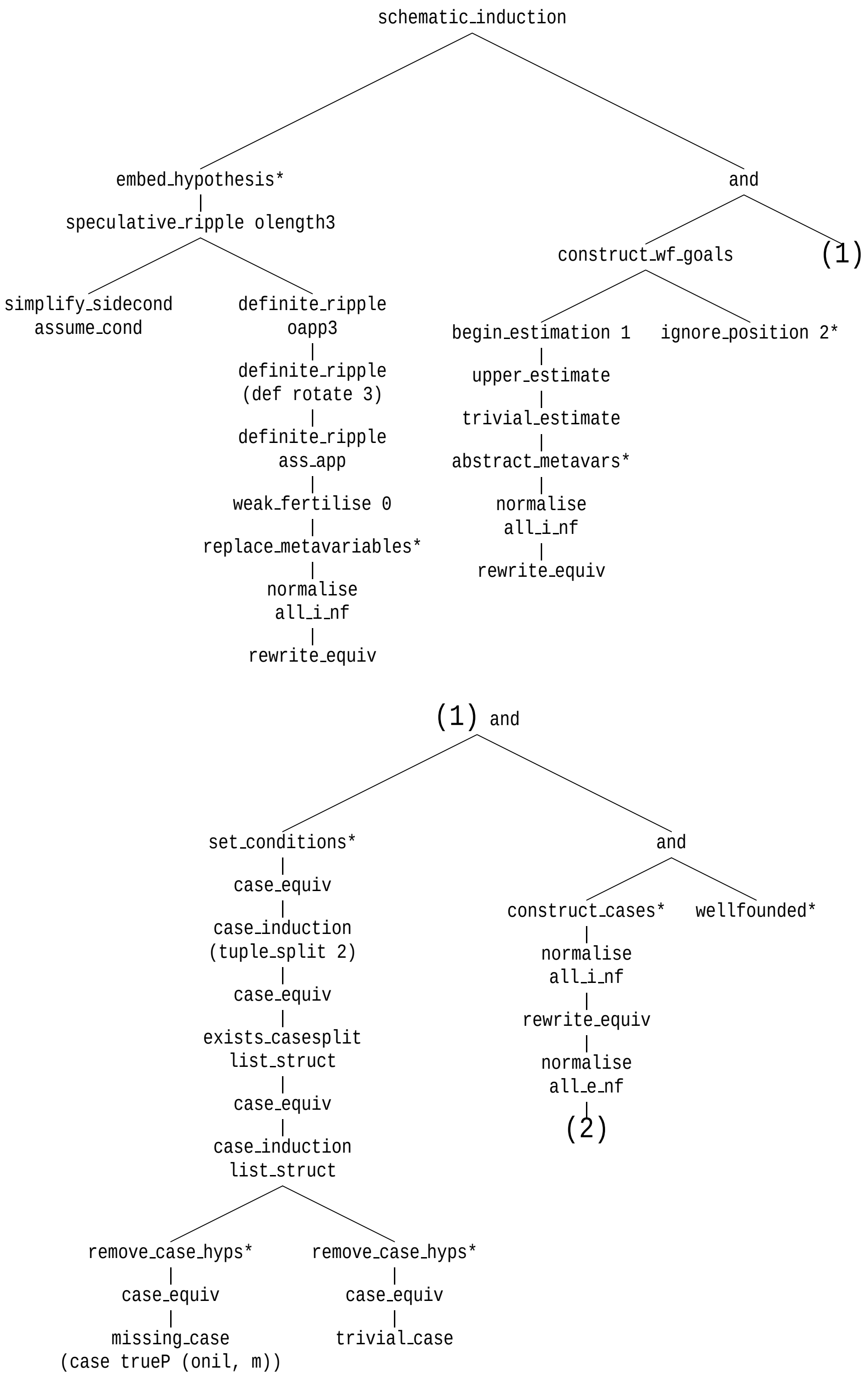

Figure 12.2: Proof plan for T9D. (2) indicates the nested induction for the subgoal $onil = l <> onil$. We omit this subplan. * indicates a purely meta-level plan step.

```
wfGoal

Method application:  wellfounded (app select_induce (tuple (app s zero :: olength :: nil)))

trueGoal!
Branch closed!

reached empty agenda
Plan Succeeded
```

The proof plan for T9D is shown in Figure 12.2, with the subplan for the nested induction in the base case omitted. Altogether, the plan has 67 nodes. *Dynamis* created and validated the following induction rule:

$$\frac{\begin{array}{rcl} & \vdash & \Phi(nil,m) \\ l \neq nil, \Phi(tail(l),(m <> (head(l) :: nil))) & \vdash & \Phi(l,m) \end{array}}{\forall l\colon list(nat).\forall m\colon list(nat).\Phi(l,m)}$$

12.4 Case Study T16C: Case Structure

We now look at a theorem which illustrates the creation of a non-trivial case structure for an induction rule. Theorem T16 is as follows:

$$\forall l{:}olist(nat).\, foldleft\_tr(\circ,el,l) = foldleft(\circ,el,orev(l))$$

The system was provided with the following lemmas:

$$\begin{array}{rcll} rev(l <> x :: nil) & = & x :: rev(l) & \text{(L25)} \\ foldright\_tr(f,x,l <> y :: nil) & = & f(y,foldright\_tr(f,x,l)) & \text{(L27)} \\ x = u \wedge y = v & \rightarrow & x \circ y = u \circ v & \text{(L31)} \end{array}$$

The induction is motivated by the either of the first two lemmas.

We use the top-level method `dynamis_lim1`, as the default constructor style method `dynamis_crit` causes a memory error with this example because of an error in the underlying implementation of λProlog (see §11.4.5).

T16C: Inital Planning

Dynamis begins by loading the appropriate functions and lemmas:

```
This is Lambda-CLAM v4.0.0
Copyright (C) 2002 Mathematical Reasoning Group, University of Edinburgh
NOTE: this program uses the MRG patched version of Teyjus 1.0-b33.

lclam:
dynamis_plan dynamis_lim1 foldleft_rev 1 constructor.

Functions: onil :: ocons :: oapp :: orev :: foldleft :: el1 :: op1 :: foldleft_tr :: nil

Eval Lemmas: foldltr_last :: rev_last :: oapp1 :: oapp2 :: nil
Wave Lemmas: foldltr_last :: rev_last :: oapp2 :: op1_functional :: nil

Loading Eval Rules: idty :: cons_functional :: neq_nil_cons :: neq_cons_nil :: oapp1 :: oapp2 ::
   orev1 :: orev2 :: foldleft1 :: foldleft2 :: foldleft_tr1 :: foldleft_tr2 :: foldltr_last ::
   rev_last :: oapp1 :: oapp2 :: nil
Loading Wave Rules: cons_functional :: oapp2 :: orev2 :: foldleft2 :: foldleft_tr2 ::
   foldltr_last :: rev_last :: oapp2 :: op1_functional :: nil

Planning:
foldleft_rev
>>> forall l:olist nat (foldleft_tr (op1, el1, l) = foldleft (op1, el1, orev l))
```

The top-level method is expanded to `dynamis_main`, and `schematic_induction` is applied, giving the usual five goal conjunction:

```
Method application:  dynamis_lim1
Method application:  dynamis_main (mo_step_case (n_spec_ripples 1)) wellfound_strat
                                  case_strat (waterfall (dynamis_lim 1))
Method application:  schematic_induction ...

allGoal olist nat (l\
caseSchema
A
((foldleft_tr  op1 el1 B) = (foldleft  op1 el1 (orev B)))
>>> ((foldleft_tr  op1 el1 C) = (foldleft  op1 el1 (orev C))))
   **
stepReduces
   **
allGoal tuple_type (olist nat :: nil) (u\
existsGoal (x\
caseGoal
Case: (trueP, <lc-0-2>)
>>> (A /\ (u = C))))
   **
maybeCases
   **
wfGoal
```

T16C: Step Case Plan

The step case begins with a speculative ripple with the definition of *foldright_tr*:

```
allGoal olist nat (l\
caseSchema
A
((foldleft_tr op1 el1 B) = (foldleft op1 el1 (orev B)))
>>> ((foldleft_tr op1 el1 C) = (foldleft op1 el1 (orev C))))

Method application:  mo_step_case (n_spec_ripples 1)
Method application:  embed_hypothesis
Method application:  n_spec_ripples 1
Method application:  speculative_ripple foldleft_tr2 (1 :: 2 :: nil)

caseSchema
A
((foldleft_tr op1 el1 B) = (foldleft op1 el1 (orev B)))
```

$$\texttt{>>> ((foldright\_tr op1 } \boxed{(\texttt{op1 } \underline{C'} \texttt{ el1})}^{\uparrow} \texttt{ C'') = (foldleft op1 el1 (orev } \boxed{(\texttt{ocons C' } \underline{C''})}^{\uparrow} \texttt{)))}$$

The speculative step on the left creates a wave front on the right which is now rippled out with the definition of *orev*:

```
Method application:  definite_rippling
Method application:  definite_ripple orev2 (3 :: 2 :: 2 :: 2 :: nil)

caseSchema
A
((foldleft_tr op1 el1 B) = (foldleft op1 el1 (orev B)))
```

$$\texttt{>>> ((foldright\_tr op1 } \boxed{(\texttt{op1 } \underline{C'} \texttt{ el1})}^{\uparrow} \texttt{ C'') = (foldleft op1 el1 } \boxed{(\underline{(\texttt{orev C''})} <> (\texttt{ocons C' onil}))}^{\uparrow} \texttt{))}$$

However, rippling is now completely blocked. The speculation critic is not being used in this example because of λProlog problems (see above), so rippling just fails. Fertilisation also fails, and *Dynamis* must backtrack over the initial speculative ripple. It tries again, this time with lemma (L27) instead of a function definition:

```
Method application:  mo_fertilise

Attempting...
strong_fertilise
Attempting...
weak_fertilise _
Attempting...
strong_fertilise_prop
backtracking over
mo_fertilise
backtracking over
definite_ripple orev2 (3 :: 2 :: 2 :: 2 :: nil)
backtracking over
definite_rippling
backtracking over
speculative_ripple foldleft_tr2 (1 :: 2 :: nil)
```

```
Method application:  speculative_ripple foldltr_last (1 :: 2 :: nil)

caseSchema
A
((foldleft_tr op1 el1 B) = (foldleft op1 el1 (orev B)))
```

$$\texttt{>>> (}\boxed{\texttt{(op1 C'' }\underline{\texttt{(foldright\_tr op1 el1 C')}}\texttt{)}}^{\uparrow}\texttt{ = (foldleft op1 el1 (orev }\boxed{\texttt{(}\underline{\texttt{C'}}\texttt{ <> (ocons C'' onil))}}^{\uparrow}\texttt{)))}$$

Again, the speculative ripple on the left creates a wave front on the right. This time the wave front is rippled out with the lemma (L25), and then with the definition of *foldleft*:

```
Method application:  definite_rippling
Method application:  definite_ripple rev_last (3 :: 2 :: 2 :: 2 :: nil)

caseSchema
A
((foldleft_tr op1 el1 B) = (foldleft op1 el1 (orev B)))
```

$$\texttt{>>> (}\boxed{\texttt{(op1 C'' }\underline{\texttt{(foldright\_tr op1 el1 C')}}\texttt{)}}^{\uparrow}\texttt{ = (foldleft op1 el1 }\boxed{\texttt{(ocons C'' }\underline{\texttt{(orev C')}}\texttt{)}}^{\uparrow}\texttt{))}$$

```
Method application:  definite_ripple foldleft2 (2 :: 2 :: nil)

caseSchema
A
((foldleft_tr op1 el1 B) = (foldleft op1 el1 (orev B)))
```

$$\texttt{>>> (}\boxed{\texttt{(op1 C'' }\underline{\texttt{(foldright\_tr op1 el1 C')}}\texttt{)}}^{\uparrow}\texttt{ = }\boxed{\texttt{(op1 C'' }\underline{\texttt{(foldleft op1 el1 (orev C'))}}\texttt{)}}^{\uparrow}\texttt{)}$$

The wave fronts on both sides are rippled out over the equality, allowing strong fertilisation to be applied. Rewriting completes the step case plan:

```
Method application:  definite_ripple op1_functional nil

caseSchema
A
((foldleft_tr op1 el1 B) = (foldleft op1 el1 (orev B)))
```

$$\texttt{>>> }\boxed{\texttt{((C'' = C'') /\textbackslash{} }\underline{\texttt{((foldright\_tr op1 el1 C') = (foldleft op1 el1 (orev C')))}}\texttt{)}}^{\uparrow}$$

```
Method application:  mo_fertilise
Method application:  strong_fertilise_prop

caseSchema
A
>>> ((C'' = C'') /\ trueP)

Method application:  replace_metavariables
Method application:  waterfall (dynamis_lim 1)
Method application:  rewrite
Method application:  normalise all_i_nf
Method application:  rewrite_equiv (and4 :: idty :: nil)

trueGoal!
Branch closed!
```

T16C: Wellfounded Step Case Plan

Dynamis now considers the four remaining induction subgoals, starting with the step case wellfoundedness goal. First, the goal is constructed explicitly:

```
stepReduces
   **
allGoal tuple_type (olist nat :: nil) (u\
existsGoal (x\
caseGoal
Case: (trueP, u)
>>> (A /\ (u = (C′ <> (ocons C′′ onil))))))
   **
maybeCases
   **
wfGoal

stepReduces

Method application:  wellfound_strat
Method application:  construct_wf_goals (const_disj (measure 1 _ :: nil) :: _)

allGoal olist nat (x\
redGoal 1
>>> A -> D (C′, (C′ <> (ocons C′′ onil))))
```

The estimation strategy is applied, producing two subgoals: the first stating that unknown difference equivalent `E` holds iff the induction terms reduce under some unknown measure function `F`; the second that `E` is true:

```
Method application:  estimation_strat
Method application:  begin_estimation 1

estGoal
E <-> F(C′) < F((C′ <> (ocons C′′ onil)))
   **
>>> (trueP -> E)
```

Considering the first goal, the lower estimation method is applied — it estimates the first argument of the $oapp$ ($<>$) function from the step case conclusion, instantiating the measure function to $olength$:

```
estGoal
E <-> F(C′) < F((C′ <> (ocons C′′ onil)))

Method application:  lower_estimate

estGoal
E′ <-> olength(C′) < olength(C′)
```

```
Method application:  trivial_estimate

trueGoal!
Branch closed!
```

The estimation of the first argument of *oapp* above had the side effect of instantiating the difference equivalent. The system now plans its rewriting proof, completing the wellfoundedness proof for the step case:

```
>>> (trueP -> (˜ (ocons (C′′, onil) = onil) \/ falseP))

Method application:  abstract_metavars
Method application:  rewrite
Method application:  normalise all_i_nf
Method application:  rewrite_equiv (imp3 :: or4 :: neq_cons_nil :: nil)

trueGoal!
Branch closed!
```

T16C: Exhaustive Cases Plan

The exhaustive cases goal comes next. The goal is non-trivial, as it contains the defined function *oapp* ($<>$), not just datatype constructors:

```
allGoal tuple_type (olist nat :: nil) (u\
existsGoal (x\
caseGoal
Case: (trueP, u)
>>> (A /\ (u = (C′ <> (ocons C′′ onil))))))
   **
maybeCases
   **
wfGoal

Method application:  case_strat
Method application:  set_conditions

caseGoal
Case: (trueP, u)
>>> (trueP /\ (u = (C′ <> (ocons C′′ onil))))
```

Even though there is only a single term being considered here, and not a tuple of terms, there is still a tuple 'wrapper' around `(C′ <> (ocons C′′ onil))` in the underlying syntax. This is removed by the `case_induction` method (rather confusingly, the generality of the method means we label the step as an induction without any inductive hypotheses and with only one case!):

```
Method application:  case_equiv (and3 :: nil)
Method application:  case_induction (tuple_split 1)
Method application:  case_equiv (tuple_eq_base :: nil)

caseGoal
Case: (trueP, p)
>>> (p = (C′ <> (ocons C′′ onil)))
```

Moving on, the case strategy goes through a waterfall of methods, eventually applying structural list induction to `v`:

```
Method application:  case_induction list_struct

caseGoal
Case: (trueP, onil)
>>> (onil = (C′ <> (ocons C′′ onil)))
   **
allGoal nat (v\ allGoal olist nat (w\
caseGoal
Case: (trueP, (ocons v w))
(w = (C′ <> (ocons C′′ onil)))

>>> ( (ocons v w)↑  = (C′ <> (ocons C′′ onil)))))
```

In the base case, an existential case split is applied to the first argument of *oapp* (<>), to allow its definition to be applied:

```
caseGoal
Case: (trueP, onil)
>>>
(onil = (C′ <> (ocons C′′ onil)))

Method application:  exists_casesplit list_struct

caseGoal
Case: (trueP,  onil)
>>> ((onil = (onil <> (ocons C′′ onil))) \/ (onil = ((ocons G H) <> (ocons C′′ onil))))
```

Rewriting reduces both disjuncts to `falseP`, and the base case is added to the list of missing cases:

```
Method application:  remove_case_hyps
Method application:  case_equiv (oapp1 :: neq_nil_cons :: or3 :: oapp2 :: neq_nil_cons :: nil)

caseGoal
Case: (trueP, onil)
>>> falseP

Method application:  missing_case (case trueP _ (tuple (onil :: nil)))

trueGoal!
Branch closed!
```

The step case also begins with an existential case split motivated by the definition of *oapp* (<>). The method reembeds the inductive hypothesis in the conclusion:

```
allGoal nat (v\ allGoal olist nat (w\
caseGoal
Case: (trueP, (ocons v w))
(w = (C′ <> (ocons C′′ onil)))
```

$$\texttt{>>> (}\ \boxed{(\texttt{ocons v }\underline{\texttt{w}})}^{\uparrow}\ \texttt{ = (C}'\texttt{ <> (ocons C}''\texttt{ onil)))))}$$

```
Method application:  exists_casesplit list_struct

caseGoal
Case: (trueP, (ocons v w))
(w = (C′ <> (ocons C′′ onil)))
```

$$\texttt{>>>}\ \boxed{(((\texttt{ocons v w}) = (\texttt{onil} <> (\texttt{ocons C}''\texttt{ onil}))) \quad \backslash/ \quad \underline{(\boxed{(\texttt{ocons v }\underline{\texttt{w}})}^{\uparrow} = (\boxed{(\texttt{ocons I }\underline{\texttt{J}})}^{\uparrow} <> (\texttt{ocons C}''\texttt{ onil})))})}^{\uparrow}$$

Rippling is tried before simplification, and succeeds in fully rippling out the wave fronts:

```
Method application:  case_ripple oapp2

caseGoal
Case: (trueP, (ocons v w))
(w = (C′ <> (ocons C′′ onil)))
```

$$\texttt{>>>}\ \boxed{(((\texttt{ocons v w}) = (\texttt{onil} <> (\texttt{ocons C}''\texttt{ onil}))) \quad \backslash/ \quad \underline{(\boxed{(\texttt{ocons v }\underline{\texttt{w}})}^{\uparrow} = \boxed{(\texttt{ocons I }\underline{(\texttt{J} <> (\texttt{ocons C}''\texttt{ onil}))})}^{\uparrow}\ )})}^{\uparrow}$$

```
Method application:  case_ripple cons_functional

caseGoal
Case: (trueP, (ocons v w))
(w = (C′ <> (ocons C′′ onil)))
```

$$\texttt{>>>}\ \boxed{(((\texttt{ocons v w}) = (\texttt{onil} <> (\texttt{ocons C}''\texttt{ onil}))) \quad \backslash/ \quad \underline{\boxed{((\texttt{v} = \texttt{I}) \quad /\backslash \quad \underline{(\texttt{w} = (\texttt{J} <> (\texttt{ocons C}''\texttt{ onil})))})}^{\uparrow}\ )}}^{\uparrow}$$

Fertilisation is now applied, which removes the defined function *oapp* (<>) from the goal. Simplification can now complete the plan:

```
Method application:  case_fertilisation
Method application:  remove_case_hyps

caseGoal
Case: (trueP, (ocons v w))
```

```
>>> (((ocons v w) = (onil <> (ocons C’’ onil))) \/ ((v = I) /\ trueP))

Method application:  case_equiv (and4 :: solve_eq :: or2 :: nil)

caseGoal
Case: (trueP,  (ocons v w))
>>> trueP

Method application:  trivial_case

trueGoal!
Branch closed!
```

T16C: Base Case Plan

Having found the missing base case above, *Dynamis* constructs the base case and discharges it with rewriting:

```
maybeCases ** wfGoal

maybeCases

Method application:  construct_cases

>>> (foldleft_tr (op1, el1, onil) = foldleft (op1, el1, orev onil))

Method application:  waterfall (dynamis_lim 1)
Method application:  rewrite
Method application:  normalise all_i_nf
Method application:  rewrite_equiv (foldleft_tr1 :: orev1 :: foldleft1 :: idty :: nil)

trueGoal!
Branch closed!
```

T16C: Final Plan

The last step is to solve the constraints on the rule’s wellfounded relation:

```
wfGoal

Method application:  wellfounded (app select_induce (tuple (app s zero :: olength :: nil)))

trueGoal!
Branch closed!

reached empty agenda
Plan Succeeded
```

The proof plan for T16C is shown in Figure 12.3. The plan has 37 nodes. *Dynamis*

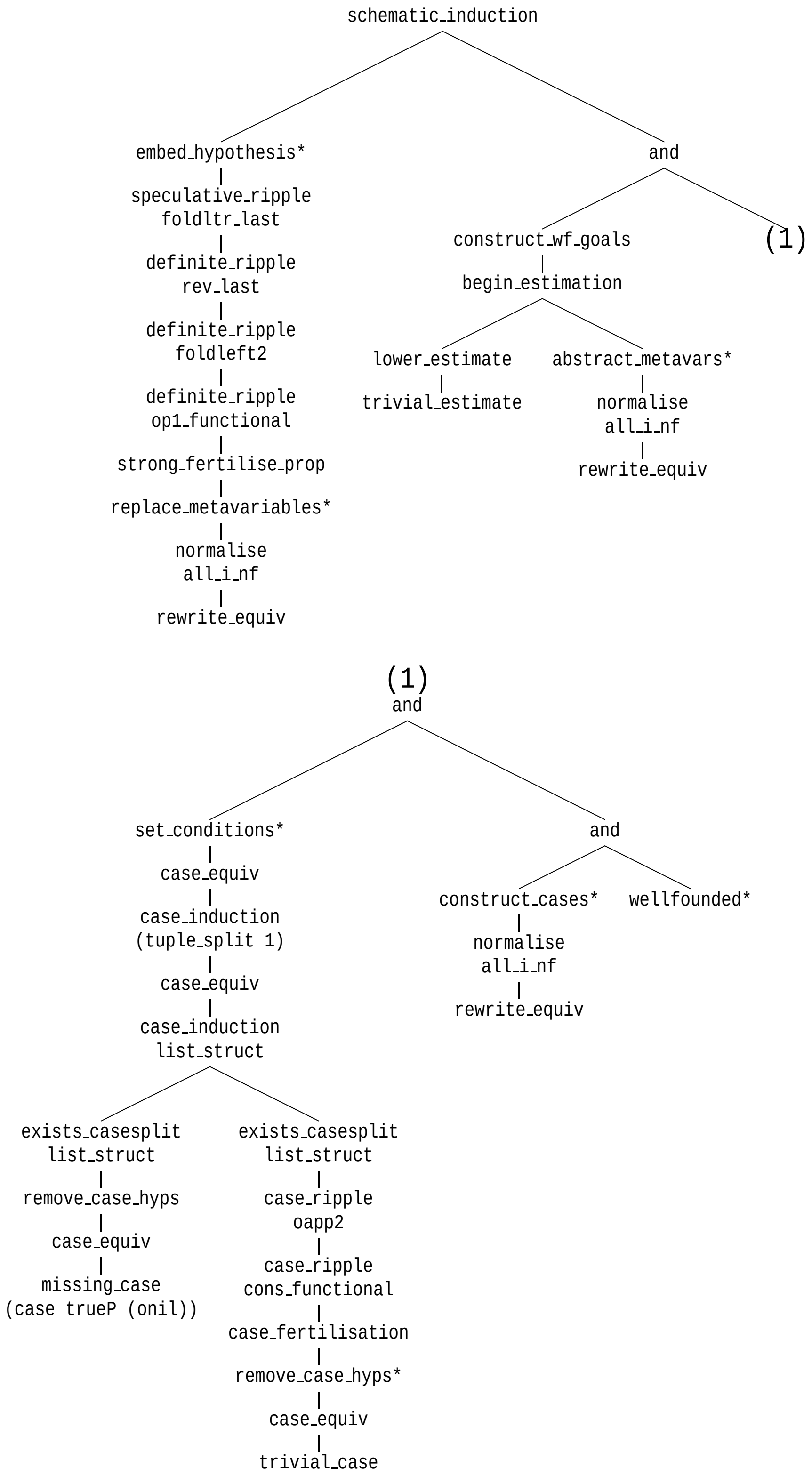

Figure 12.3: Proof plan for T16C. * indicates a purely meta-level plan step.

created and validated the following induction rule:

$$\frac{\begin{array}{rcl} & \vdash & \Phi(nil) \\ \Phi(l) & \vdash & \Phi(l <> (x :: nil)) \end{array}}{\forall l : list(nat).\Phi(l)}$$

As part of creating the rule, *Dynamis* also invented and validated the following case split:

$$\forall u{:}list(nat).(u = nil) \lor \exists l{:}list(nat).\exists x{:}nat.(u = (l <> (x :: nil)))$$

12.5 Summary

This chapter has provided in-depth details of three case studies where the *Dynamis* system has been used to automatically generate a proof plan. All these examples could not be automatically solved using recursion/ripple analysis. Furthermore, the case studies have demonstrated *Dynamis*'s ability to:

- Control problematic speculative steps using a critic.
- Handle both constructor and destructor style examples.
- Generate novel case structure for an induction rule.

Chapter 13

Related & Further Work

13.1 Introduction

In this chapter we compare in detail our strategy and some previous work on induction rule selection. Specifically:

- Recursion analysis and related approaches.
- Kraan's *Periwinkle* system [Kraan, 1994].
- Hutter's labelled fragments [Hutter, 1994].
- Protzen's lazy induction [Protzen, 1995].

These techniques were surveyed in §2.

13.2 Recursion Analysis

Recursion analysis can be considered to be a group of techniques, descended from the induction selection methods of [Boyer and Moore, 1979], which all select an induction rule derived from the relevant recursive functions. The techniques use various methods to combine and select induction rules, i.e. subsumption [Stevens, 1990], containment [Walther, 1993] and ripple analysis [Bundy et al., 1989]. They are surveyed in detail in §2.6. Most inductive theorem provers which automate induction selection use a form of recursion analysis.

As discussed in §2.7, these techniques have two significant disadvantages: that they must select a rule from a 'space' of induction rules which is predefined by the function definitions, and that they do not take the effect of the choice into account beyond the first rewriting of each induction term.

Previous work (e.g. [Protzen, 1995] has already addressed these problems to a limited extent. Our work also has clear theoretical advantages over recursion analysis in that it overcomes both these problems, and can prove a wider range of problems. The evaluation of Chapter 11 has also demonstrated pratical advantages of our strategy, as the *Dynamis* system planned proofs for a collection of theorems that cannot be solved by recursion analysis.

13.3 The Periwinkle System

Like our strategy, *Periwinkle* uses middle-out reasoning to determine a suitable step case for an inductive rule, i.e. a schematic step case goal becomes instantiated during its proof [Kraan, 1994] (see also §2.7.1). The goal schemas are similar in that they use second order meta-variables to represent unknown induction terms, and use rippling to guide the step case proof.

However, our work goes beyond Kraan's in three important respects: the dynamic construction of induction rules, the generality of the schema and speculation control.

Periwinkle used the step case obtained by middle-out reasoning to select an induction rule from a prestored set. In contrast, we use the step case as the basis for an induction rule which is constructed ‘from scratch’. Our approach lifts the restriction that all induction rules must be provided to the system beforehand from some outside source, e.g. generated from function definitions provided by the user.

Our step case schema is more general, as it can be instantiated to destructor style step cases. We have also suggested using Protzen’s heuristic (see Chapter 4.3.3) to allow multiple induction hypotheses, although this has not yet been implemented in *Dynamis*. Kraan’s step case schema did not allow any of these features, severely restricting the kind of inductive proofs it could perform.

The third key difference is the control of speculative rippling steps — the steps which instantiate meta-variables — and so determine the form of the induction rule. Kraan recognised that such steps made rippling potentially non-terminating, even when definite (non-speculative) steps were preferred. In other words, there is no limit on the complexity of the step case. *Periwinkle* overcame this problem by placing a finite limit on the number of such steps, typically a limit of one. This in turn limits step case proofs, and hence the induction proofs, that the system can find. Our strategy overcame this problem by allowing an initial speculative step, and requiring subsequent speculations to be licensed by a critic which analysed the failure of rippling. This was discussed in greater depth in Chapter 7.

Another difference between our work and Kraan’s is that she uses higher-order pattern unification, a decidable restricted form of high-order unification (HOU), whereas we use full HOU. Although decidability might be useful in some contexts, we have not experienced problems with termination in our work. Other authors on middle-out reasoning have also used full HOU [Hesketh, 1991, Ireland and Bundy, 1996].

13.4 Labelled Fragments

The first technique which dealt with constructing an induction rule entirely using information gleaned from proof was the use of labelled fragments in [Hutter, 1994] (see §2.8.1). The key difference between our approaches is that Hutter's work was aimed at proving existential theorems, whilst we have concentrated on universal theorems. However, it is worth comparing the methods, as there is the potential to extend each technique into the other's domain.

Hutter uses labelled fragments — basically an abstract representation of wave rules — to predict the induction terms which will lead to a successful ripple proof. This is done by performing a kind of 'abstract step case' using the rule fragments. However, the prediction can be incorrect, i.e. when the actual step case proof is performed it may fail. In contrast, our strategy determines the same information whilst actually performing the proof, and so avoids this risk. The technique was also restricted to generating destructor style induction rules, whilst our strategy is not.

13.5 Lazy Induction

Lazy induction [Protzen, 1995] is similar to our strategy in that it constructs an entire valid induction rule during a proof attempt, avoiding the need to rely on user provided rules or those generated from terminating function definitions. It takes the original conjecture as the step case conclusion, using rippling to guide the proof, definitional case splits to construct separate proof cases and lazily generating induction hypotheses whenever they can be used to rewrite the goal (see §2.8.2 for more details).

The technique uses steps which increase the ripple measure, and play the same rôle as the meta-variable instantiating speculative ripples of the schema-based approach.

We name these speculative steps by analogy, although one of our speculative steps may be equivalent to many lazy speculative steps.

There are three fundamental differences between our work and lazy induction: the restriction to destructor-style, the problem of mixed speculation and speculation control. As far as we know there are no relative disadvantages to our approach.

One key difference is that this method can only generate destructor style inductive proofs — Protzen's work is entirely based in a destructor style formalism. We argued in Chapter 3 why this is overly restrictive for inductive theorem proving: even if one only ever uses destructor style functions (which authors, in general, do not) then some useful induction rules are still suggested by 'constructor style lemmas', e.g.

$$\mathit{foldleft\_tr}(F,X,L <> [Y]) = F(\mathit{foldleft\_tr}(F,X,L),Y)$$

Unless an equivalent destructor style lemma is also present the proof will not be found, e.g.

$$L \neq \mathit{nil} \rightarrow \mathit{foldleft\_tr}(F,X,L) = F(\mathit{foldleft\_tr}(F,X,\mathit{chop}(L)),\mathit{last}(L))$$

(*chop* removes the last element of a list.) Converting between the two requires synthesing inverse functions, which is not a practical alternative to allowing constructor style inductions.

The other fundamental difference between our strategy and lazy induction is that the latter has no explicit representation of the as-yet-unknown step case throughout the proof. It does not construct an induction hypothesis until fertilisation, i.e. the point the hypothesis is applied. The drawback here is that in the middle of the proof, when some decisions have been made which correspond to a particular form of induction hypothesis, subsequent proof steps have no way of accessing this information, and may make inconsistent decisions. We call this problem *mixed speculation*, and it can significantly increase the size of the search space, as all the search paths where the speculative steps are not consistent must be explored. The problem arises because speculation is a local phenomenon, not accessible to the rest of the proof.

In contrast, speculation in our strategy is global. When a decision is made about the form of the step case, a meta-variable is instantiated, and so this decision is propagated throughout the proof. Future proof steps must be consistent with this choice of instantiation, and are prevented from making inconsistent choices. This prevents mixed speculation, and so cuts down the search space compared to lazy induction.

Mixed speculation occurs, for example, if we apply lazy induction to theorems T15, T16 or T17 from Chapter 11. However, in order to more clearly illustrate the phenomenon, we use a more concise, abstract example theorem:

$$f(x) = g(x) \tag{13.1}$$

where f and g are defined as (omitting base cases):

$$f(x) = f(p(x)) \tag{13.2}$$

$$g(x) = r(g(p(x))) \tag{13.3}$$

and the following lemmas are given:

$$f(x) = f(q(x)) \tag{13.4}$$

$$g(x) = g(q(x)) \tag{13.5}$$

Applying lazy induction to the goal (13.1), both (13.2) and (13.4) can be used to speculate on the LHS. The same is true for (13.3) and (13.5) on the RHS. Lazy induction will try all four possible combinations of these speculative steps, in the following order:

$$f(\boxed{p(\underline{x})}) \quad = \quad \boxed{r(\underline{g(\boxed{p(\underline{x})})})}^{\uparrow} \tag{13.6}$$

$$f(\boxed{p(\underline{x})}) \quad = \quad g(\boxed{q(\underline{x})}) \tag{13.7}$$

$$f(\boxed{q(\underline{x})}) \quad = \quad \boxed{r(\underline{g(\boxed{p(\underline{x})})})}^{\uparrow} \tag{13.8}$$

$$f(\boxed{q(\underline{x})}) \quad = \quad g(\boxed{q(\underline{x})}) \tag{13.9}$$

Of these, only the last (13.9) is successful, with the induction hypothesis $f(q(x)) = g(q(x))$. The first goal (13.6) almost matches the hypothesis $f(p(x)) = g(p(x))$, but the wavefront $r(\ldots)$ is blocked and the proof fails. Goals (13.7) and (13.8) can be weak fertilised, but both proofs fail because of mixed speculation — each has induction term $p(\ldots)$ on one side and $q(\ldots)$ on the other.

If instead we applying our induction strategy to the goal (13.1), we find that only two of the four combinations are generated (meta-variables are written as $A, B, \ldots$ for simplicity). The first is:

$$\begin{array}{rcl}
f(A) = g(A) & \vdash & f(B) = g(B) \\
f(p(A')) = g(\boxed{p(\underline{A'})}) & \vdash & f(p(B)) = g(B) \\
f(p(A')) = g(p(A')) & \vdash & f(p(B)) = \boxed{r(\underline{g(p(B))})}^{\uparrow}
\end{array}$$

This branch corresponds to (13.6), and is blocked. Backtracking over the first step we get the successful branch:

$$f(A) = g(A) \quad \vdash \quad f(B) = g(B) \tag{13.10}$$

$$f(q(A')) = g(\boxed{q(\underline{A'})}) \quad \vdash \quad f(q(B)) = g(B) \tag{13.11}$$

$$f(q(A')) = g(q(A')) \quad \vdash \quad f(p(B)) = g(q(B)) \tag{13.12}$$

Our strategy has avoided searching the two inconsistent branches. Although the additional search is not that great in this abstract example, it will increase with the number of alternative wave rules and reducible terms. In addition, the search required to establish that an inconsistent branch will fail could be arbitrarily large.

The third fundamental difference between our strategy and lazy induction, is that Protzen did not address the problem of non-terminating speculation in his thesis. In fact, he does not even recognise it, leaving his strategy as described highly prone to non-termination. Hence our work on speculation control (see §13.3) represents an advance over lazy induction.

13.6 Further Evaluation

We now discuss various ways in which the work described in this thesis could be continued. The most immediate area is to extend the implementation of *Dynamis* to reflect the full induction strategy set out in Chapter 4 to Chapter 8. This would require:

1. Case splits during rewriting;
2. Creation of multiple induction hypotheses, via Protzen's Heuristic, i.e. adding applicable instances of the inductive conjecture as hypotheses during the proof (see §4.3.3);
3. Creation of multiple step cases (see §6.3);
4. Side condition critic for failed estimation proofs (see §6.5.5);
5. $\pi\sigma$-rewriting (see §8.3.2).

Implementation of features (1) to (4) was suggested by the experimental evaluation in Chapter 11: they would allow a larger number of theorems to be planned by *Dynamis* using the default strategies. Feature (5) would reduce the rewriting search — although, as discussed below, its completeness has not yet been established. Features (3) to (5) are novel, and implementation would allow their effectiveness to be evaluated.

In addition, the description of (2) in [Protzen, 1995] does not provide much detail of its implementation or evaluation, and no implementation is currently available. Including it in *Dynamis* would allow Protzen's Heuristic to be assessed. Given our experience with speculative ripple steps, we anticipate that additional search heurisitics will have to be developed to make this effective.

Another possible direction for research is the implementation of lazy induction [Protzen, 1995] to allow experimental comparison with our schema-based approach. The evaluation in Chapter 11 failed to support or refute the hypothesis that our strategy is more powerful than lazy induction, because of shortcomings of the implementation. However, it did highlight the lack of data on lazy induction, and a working implementation is required to overcome this problem. It would also allow us to test the theoretical claim that using a schema reduces search compared to a lazy generation approach, by avoiding mixed speculation (see §13.5 above).

13.7 Developing the Strategy

As well as fully implementing and evaluating our induction strategy, there are a number of areas where we can see the strategy could be improved. We discuss these in turn below.

Extending the Speculation Critic

The speculation critic currently has two obvious shortcomings. Firstly, it can only create constructor style step cases. However, it may be possible to extend the critic patch to destructor style by instantiating a hypothesis meta-variable to match the fully rippled-in wave fronts in the conclusion. As an example, consider the following, where a required wave front has been rippled in to find an instantiation that would generate it (A etc. are meta-variables):

$$\ldots A+B\ldots \;\vdash\; \ldots \boxed{s(\underline{C+D})}^{\downarrow}\ldots$$

To find a destructor style step case, a destructor style critic needs to identify that the wave front can be generated from $C+D$ using the destructor style definition of $+$, providing A is instantiated to $p(A')$.

Secondly, the current speculation critic can suffer from non-termination, e.g. theorem T7C in §11.4.3. This could be prevented by imposing some kind of measure reduction on the critiqued goals to ensure that the proof has progressed since the last application of the critic. Further experimental work is required to obtain more examples of desirable and undesirable speculation, in order to formulate a suitable measure.

Formalising Neutralisation

Although the neutralisation procedure used by the step case strategy has been implemented in *Dynamis*, we have not formulated a clear, formal description. Doing this would allow a cleaner implementation, and probably help us to understand the bugs in *Dynamis* that allowed certain corresponding wave-fronts in hypothesis and conclusion to remain unneutralised (see Chapter 11). We have already provided a specification that any neutralisation procedure must meet (see Definition 3). The procedure as implemented is suitable for formulation as a set of rules, similar to those that defined embeddings [Smaill and Green, 1996].

Instantiation Selection

We can see two shortcomings with the way in which the strategy searches the possible instantiations of the meta-variables in a given goal. Firstly, it is prone to searching the same instantiation multiple times if it is generated by different speculative ripples. A more efficient strategy would be to identify all possible speculative steps and the instantiations they generate, and only try each instantiation once.

Interleaving Rule Validation

The induction strategy currently constructs a step case, and then constructs a wellfoundedness plan for it. In contrast, [Protzen, 1995] interleaves the two processes, by only allowing wave fronts that contain lower argument bounded functions to be moved towards induction positions, where they may be incorporated into induction hypotheses. Hence the wellfoundedness checks are integrated into rewriting. This is more efficient, as non-wellfounded step cases are pruned at an early stage, rather than after they have been completed.

A similar approach could be taken with our strategy, using a planner that is capable of prioritising open subgoals. By giving a partially instantiated estimation goal a higher priority than its step case, the strategy could ensure that any instantiation of the induction terms is immediately validated before the step case continues.

Existential Problems

All the example theorems considered in this thesis have been purely universally quantified. The application to existential theorems — and hence program synthesis — would be a fruitful research direction, given that one significant problem in this work is the need to generate novel inductions rules that cannot be generated from the function definitions provided, i.e. the existential witness is a program with a novel recursive structure [Hutter, 1994].

Object Level Proofs

The λ*Clam* system, and hence the *Dynamis* system built on top of it, does not have any facility for constructing object level proofs from the proof plans it produces. Indeed, λ*Clam* has been deliberately designed to avoid commitment to a particular logic — this is entirely the decision of the method designer, who may be as specific about the logic as she chooses.

Previous work on proof planning has established that inductive proof plans can be used by a variety of tactic-based systems to generate proofs in a variety of logics, e.g. [Bundy et al., 1991, Boulton et al., 1998]. However, our proof plans are substantially different from those in previous work, in that they also include a proof that the induction rule is valid. Further work is required to validate these plans by execution to object level proofs.

13.8 Exploring πσ-Rewriting

Chapter 8 set out a novel technique for controlling our induction strategy's search during rewriting/rippling. However, the technique is more widely applicable to any non-confluent rewriting system. Several areas of future research suggest themselves. Firstly, identifying other applications which use such rewriting. Secondly, implementation and evaluation of the technique, to assess how effective it is in reducing search for various applications.

Another direction is the proof of completeness for σ-rewriting, which we presume could be used with the existing π-completeness result to prove completeness for full $\pi\sigma$-rewriting.

13.9 Research on Proof Planning

Another area closely related to our thesis is proof planning, and the λ*Clam* proof planner in particular. On a prosaic level, our evaluation uncovered bugs in the λProlog implementation underlying the λ*Clam* proof planner. These need to be addressed in some way.

More interestingly, in Chapter 10 we found the lack of a cut methodical in λ*Clam* to be a very significant factor in the design of our methods. Search during proof plan construction can be made impractical when key choices are preceded by a large number of unimportant ones — unless the previous choices points have been cut. Without this ability, we were sometimes forced to use a less clear method formulation than we would have chosen, in order to avoid constant backtracking over such unimportant choices, e.g. equivalence preserving rewriting. The design could have been much cleaner if a cut methodical was available. A design for a proof planner which handles cut using explicit OR branches has been proposed[1], but not yet implemented.

In Chapter 9 we gave a design for a critics planner which integrates critics with the methodical-based approach of λ*Clam*. Further research could assess whether this is, in general, a suitable proof planning architecture, by perhaps investigating whether it allows superior reimplementations of previous critic work, e.g. with more declarative formulations.

Our induction strategy is a case study in delaying search choices using metavariables: they are used for induction terms, hypotheses, measure functions etc. Fur-

[1] Julian Richardson, personal communication.

ther work is needed to establish how is this related to previous proof planning research that uses similar techniques e.g. [Cheikhrouhou and Siekmann, 1998], and whether such work can be included in a common reasoning framework.

13.10 Summary

In this chapter we have compared our induction strategy to four pieces of closely related research in automated induction rule selection. It had clear advantages over all of them.

Compared to Kraan's *Periwinkle* system our strategy has three significant advantages:

- It constructs induction rules dynamically, rather than relying on a prestored set.
- Induction rules may be destructor style and, in theory, have multiple induction hypotheses.
- The speculation critic allows speculative rippling to be flexibly controlled, rather than setting a fixed limit on the search.

Compared to Protzen's lazy induction our strategy has three advantages:

- Induction rules need not be destructor style. Recall that constructor style rules are required even if all function definitions are destructor style.
- Our strategy does not suffer from mixed speculation, because meta-variables are used to explicitly represent the developing induction rule.
- The speculation critic controls speculative rippling — a problem not addressed at all by Protzen.

We have also outlined a number of future search directions based on our work, which include:

- Further implementation and evaluation of our strategy with the *Dynamis* system.
- Designing a destructor style speculation critic, and finding a measure that forces the critic to terminate.
- Ensuring that each meta-variable instantiation is considered only once.
- Interleaving the step case proof with the wellfoundedness proof.
- Synthesising programs with novel recursive structures.
- Executing *Dynamis*'s plans to object level proofs.
- Implementing and proving completeness of $\pi\sigma$-rewriting.
- Evaluating our proof planner design.

Chapter 14

Conclusions

14.1 Introduction

In this chapter we review the contributions made by the thesis, and assess whether our work has met the aims laid out at the beginning.

14.1.1 Contributions of the Thesis

Our thesis contributes specifically to the understanding of inductive theorem proving in four key ways:

1. It identifies the significance of restricting induction rules to constructor style or destructor style.
2. It describes improved search control and coverage for induction rule creation using a schema-based approach.
3. It includes a novel procedure for generating missing proof cases.
4. It gives a modular induction strategy for creating induction rules during proof.

We expand on each of these below.

Rule Structure Firstly, in Chapter 3 we have explained the relationship between constructor and destructor style induction rules and function definition, and explained why neither style of rule is totally sufficient for inductive proof. This prompted the definition of simple induction rules as a suitable class for automated proof, and a novel formulation of creational rippling in order to provide search control for this class.

Search Control and Coverage Secondly, we have shown in Chapter 4 how a step case schema can be used to delay key choices until the middle of the step case proof, giving better choice of induction rule than recursion/ripple analysis, and which unlike previous work [Kraan, 1994, Protzen, 1995] is not just restricted to either constructor or destructor style step cases. Search control is also improved: non-terminating speculative steps are controlled using a critic on the ripple method (Chapter 7), and using a meta-variable schema avoids the problem of mixed speculation that arises with a lazy generation approach [Protzen, 1995] (Chapter 13).

Case Synthesis Thirdly, a procedure for generating the missing cases of an induction rule was given in Chapter 5, based on trying to prove that the existing cases are exhaustive. The failed proof is patched by adding missing cases to the conjecture, following previous work on correcting faulty conjectures [Protzen, 1995, Monroy, 2000]. We identified that non-equivalence preserving steps — in particular instantiating free variables — are incompatible with such corrective techniques. The equivalence preserving *existential case splits* was proposed instead.

Modular Strategy Lastly, in Chapter 6 an induction strategy for generating induction rules during the inductive proof was described. It is modular with respect to three sub-strategies: one for step case generation, another for ensuring step case well-foundedness, and a third for generating missing proof cases. A restricted version of this strategy has been implemented in the *Dynamis* system, using our schematic step case strategy, Walther's estimation method and our case generation procedure as the three sub-strategies 'modules'.

The thesis also makes more general contributions to automated theorem proving. Some of the techniques mentioned above have potential applications outside induction rule creation. Our strategy for generating missing cases of a case analysis, which connects the problem to research in correcting faulty conjectures. We proposed $\pi\sigma$-rewriting in Chapter 8 as a way to further reduce the proof search in inductive proof, and we have proved the completeness of π-rewriting, a useful restriction of this technique. It is a technique that could be applied to other non-confluent rewriting systems. Furthermore, our arguments for the superiority of a schema approach over lazy generation has implications for any delayed commitment strategy.

Finally, we have described a novel proof planning architecture for specifying critics and combining them with method expressions.

14.1.2 Have We Achieved Our Aims?

The aims set out at the beginning of this thesis were to design a practical, delayed choice induction rule creation strategy, which improved on previous research with better search control for speculation steps and a wider range of coverage of induction rules, and hence theorems.

We have demonstrated that our strategy is a practical approach to induction rule creation by implementing it in the *Dynamis* system and evaluating it on a range of test problems. Three of the contributions above improve the search control for the crucial step case proof, and of these two have been implemented and evaluated. The evaluation also supported the hypotheses that the strategy can construct a wider range of induction rules than previous work, which has been restricted to constructor style [Kraan, 1994] or destructor style [Protzen, 1995] induction rules.

However, a few of our aims have not been met. The claim that our induction strategy is *strictly* better than lazy induction — i.e. it can prove *any* theorem lazy induction can — has not been backed up with *experimental* evidence. This is partly because of the lack of available data for lazy induction. We have shown in Chapter 13 that our strategy is theoretically superior to lazy induction, although experimental evidence could not be gathered because no working implementation exists. We hope further work will be able to gather this evidence. Furthermore, some parts of the strategy still have to be implemented and evaluated experimentally: notably creating induction rules with multiple induction hypotheses and multiple step cases, and $\pi\sigma$-rewriting.

In conclusion, further implementation and evaluation work is required to provide conclusive experimental evidence that our full induction strategy meets *all* our aims, but the majority of our original aims have been met. We have demonstrated experimentally that even a partial version already exceeds the state of the art in automated inductive theorem proving in several important respects.

Our work also has implications beyond inductive theorem proving. The induction strategy presented in this thesis is perhaps one of the most complex yet implemented using proof planning, both because it brings several complex proof strategies together in order to construct an inductive proof, and because of its extensive use of delayed commitment with meta-variables. This demonstrates that proof planning is a viable framework for developing such complex automated proof strategies. We anticipate that the techniques employed here can be used to improve automated theorem proving in a variety of domains, and that a better understanding of proof planner design —

such as improved support for critics and meta-variables — would be of great benefit to automated reasoning.

Appendix A

Glossary

[n] The finite set $\{1,2,\ldots,n\}$.

[n,m] The finite set $\{n,n+1,\ldots,m\}$.

Base Case An *induction case* with no *induction hypothesis*.

Case Complete Covering all possible cases. For example, an complete set of *induction cases*, or complete *recursive definition*.

Case Conditions A hypothesis of an *induction case* which is *not* a variant of the rule's conclusion.

Case Formula A formula which expresses the *case completeness* of a set of cases.

Constructor Style Of an *induction rule*: having *induction terms* which are compound in conclusions of each step case, and non-compound in the hypotheses. Of a *recursive definition*: having a head with compound arguments and recursive calls with non-compound arguments.

Context In *rippling*: parts of term which do not belong to the term's *skeleton*.

Creational Rippling An extension of *rippling* which can handle *wave fronts* in the *induction hypothesis* by having a common *skeleton* for two different terms. See §3.4.

Destructor Style Induction Of an *induction rule*: having *induction terms* which are non-compound in conclusions of each step case, and compound in the hypotheses. Of a *recursive definition*: having a head with non-compound arguments and recursive calls with compound arguments.

Domain Of a *substitution* σ: the set of variables $\{x : x/t \in \sigma\}$ replaced by the substitution, written as $Dom(\sigma)$.

Dual Induction Of a *recursive function*: an *induction rule* with the same recursive structure as the function: cases of the definition map to *induction cases*; the head of a defining equation maps to a case's conclusion; recursive calls map to *induction hypotheses*; function arguments map to *induction terms*; conditions on an equation map to *case conditions*.

Embedding A mapping of a term tree into another term tree where function symbols and constants are mapped to copies of themselves and which preserves ordered ancestor-descendant relationships. Can be used in *rippling* to map a *skeleton* into another term.

Estimation A proof technique used to prove inductions and definitions *wellfounded*.

Existential Case Split An equivalence preserving proof step which proves an existential statement by proving a disjunct of instantiations of the existential variable. The set of instantiations must be *case complete*.

Induction A proof which establishes a statement by using some variants of a statement to prove another variant. See *induction rule* and §2.2.

Induction Case The premise of an *induction rule*.

Induction Hypothesis A hypothesis in an *induction case* which is a variant of the rule's conclusion.

Induction Term A term substituted into an *induction position* in the premises of an *induction rule*.

Induction Position A universally quantified variable in the conclusion of an *induction rule*. Also the corresponding subterm in the variants of the conclusion in the rule's premises.

Induction Rule A rule of inference which represents an *induction* argument. This thesis deals with *simple induction rules*.

Lazy Induction A technique for automating *destructor style induction* by rewriting a conjecture to create and remove *context*, using *Protzen's heuristic* to generate *induction hypotheses*. See §2.8.2.

Multiset An unordered collection of objects, in which each object may appear more than once.

Neutralisation In *creational rippling*: the process of finding corresponding *wave fronts* in two terms and making this syntax part of their common *skeleton*.

Noetherian Induction The most general form of *induction*. All *induction rules* are instances of the Noetherian induction rule. See §2.2.1.

Protzen's Heuristic In automated *induction,* generating an *induction hypothesis* when the goal can be rewritten with an instance of the original conjecture.

Recursive Function A function defined in terms of itself. The definition must be *wellfounded* to be valid.

Rippling A heuristic rewriting technique which removes the differences between a term and its *skeleton*. Used to automate *step case* proofs by removing the differences between the induction conclusion and one or more induction hypotheses. See §2.5.

Simple Induction Rule A syntactic restriction on *induction rules* where the rule's conclusion is of the form $\forall x_1 \ldots . \forall x_n . \Phi$. The rule's premises are all sequents that have a conclusion which is an instance of Φ and a list of hypotheses which are either:

- *induction hypotheses* which are instances of Φ with optional universal quantification, or
- *case conditions*.

See §3.3 for a formal definition. For a simple induction rule to be valid it is sufficient that it is *wellfounded* and *case complete*.

Skeleton A term formed by removing some of the structure from another term.

Step Case An *induction case* with at least one *induction hypothesis*.

Substitution A function from terms to terms, defined by a set of variable/term pairs x/t. A substitution σ replaces all occurences of x with t for all $x/t \in \sigma$. See *domain*.

Var The free variables of a term.

Wave Front A syntactic difference between a term and its *skeleton* in *rippling*.

Wellfounded Definition A *recursive definition* where each recursive call is smaller than the head of the definition, by some *wellfounded relation*.

Wellfounded Induction An *induction* where the each *induction hypothesis* is smaller than its corresponding conclusion by some *wellfounded relation*.

Wellfounded Relation A relation $\succ$ with no infinite descending chains $x_1 \succ x_2 \succ x_3 \succ \ldots$.

Appendix B

Datatype & Function Definitions

This appendix collects together all of the definitions for the datatypes and functions mentioned in this thesis. For simplicity all functions, including datatype destructors, are total. If a function is defined under a alternative name in *Dynamis* this is given.

Datatype: *bool*

The boolean datatype simply has two base constructors:

$$\begin{array}{rcl} true & : & bool \\ false & : & bool \end{array}$$

We define the usual propositional functions $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$ of type $bool \rightarrow bool \rightarrow bool$.

Datatype: *nat*

The Peano natural numbers *nat* has two constructors 0 (zero) and s (successor) and the destructor p (predecessor):

$$\begin{array}{rcl} 0 & : & nat \\ s & : & nat \longrightarrow nat \end{array}$$

$$\begin{array}{rcl} p & : & nat \longrightarrow nat \\ p(0) & = & 0 \\ p(s(X)) & = & X \end{array}$$

Datatype: $list(\tau)$

The $list(\tau)$ (`olist` in *Dynamis*) datatype has two constructors *nil* (the empty list, `onil`) and :: (`ocons`) and one destructor $tail$:

$$\begin{array}{rcl} nil & : & list(\tau) \\ :: & : & \tau \longrightarrow list(\tau) \longrightarrow list(\tau) \end{array}$$

$$\begin{array}{rcl} tail & : & list(\tau) \longrightarrow list(\tau) \\ tail(nil) & = & nil \\ tail(H :: T) & = & T \end{array}$$

For each type τ we define a second destructor $head_\tau$:

$$\begin{array}{rcl} head_\tau & : & list(\tau) \rightarrow \tau \\ head_\tau(H :: T) & = & H \\ head_{bool}(nil) & = & true \\ head_{nat}(nil) & = & 0 \\ head_{list(\alpha)}(nil) & = & nil \\ head_{card}(nil) & = & red \\ & \vdots & \end{array}$$

We write omit the subscript τ when this is obvious from the context. An alternative would be to use partial functions and define a single generic *head*.

Datatype: *card*

The *card* datatype is defined for the Gilbreath Card Trick (see Chapter 11), and is isomorphic to *bool*.

$$\begin{array}{rcl} red & : & card \\ black & : & card \end{array}$$

Function: $<>$ (*append*)

`oapp` in *Dynamis*.

$$\begin{array}{rrcll} & <> & : & list(\tau) \rightarrow list(\tau) \rightarrow list(\tau) & \\ & nil <> M & = & M & \\ & H :: T <> M & = & H :: (T <> M) & \text{(C)} \\ L \neq nil \rightarrow & L <> M & = & head(L) :: (tail(L) <> M) & \text{(D)} \end{array}$$

Function: *even*

$$\begin{array}{rrcll}
 & even & : & nat \rightarrow bool & \\
 & even(0) & = & true & \\
 & even(s(0)) & = & false & \\
 & even(s(s(X))) & = & even(X) & \text{(C)} \\
X \neq 0 \wedge X \neq s(0) \rightarrow & even(X) & = & even(p(p(X))) & \text{(D)}
\end{array}$$

Function: *evenelems*

$$\begin{array}{rcll}
evenelems & : & list(\tau) \rightarrow list(\tau) & \\
evenelems(nil) & = & nil & \\
evenelems(X :: nil) & = & nil & \\
evenelems(X :: Y :: L) & = & Y :: evenelems(L) & \text{(C)} \\
L \neq nil \wedge tail(L) \neq nil \rightarrow evenelems(L) & = & head(tail(L)) :: evenelems(tail(tail(L))) & \text{(D)}
\end{array}$$

Function: *foldleft*

$$\begin{array}{rrcll}
 & foldleft & : & (\alpha \rightarrow \beta \rightarrow \alpha) \rightarrow \alpha \rightarrow list(\beta) \rightarrow \alpha & \\
 & foldleft(F,A,nil) & = & A & \\
 & foldleft(F,A,H :: T) & = & F(foldleft(F,A,T),H) & \text{(C)} \\
L \neq nil \rightarrow & foldleft(F,A,L) & = & F(foldleft(F,A,tail(L)),head(L)) & \text{(D)}
\end{array}$$

Function: *foldleft_tr*

$$\begin{array}{rrcll}
 & foldleft\_tr & : & (\alpha \rightarrow \beta \rightarrow \alpha) \rightarrow \alpha \rightarrow list(\beta) \rightarrow \alpha & \\
 & foldleft\_tr(F,A,nil) & = & A & \\
 & foldleft\_tr(F,A,H :: T) & = & foldleft\_tr(F,F(A,H),T) & \text{(C)} \\
L \neq nil \rightarrow & foldleft\_tr(F,A,L) & = & foldleft\_tr(F,F(A,head(L)),tail(L)) & \text{(D)}
\end{array}$$

Function: $foldright$

$$\begin{array}{rrcll}
 & foldright & : & (\beta \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow list(\beta) \rightarrow \alpha & \\
 & foldright(F,A,nil) & = & A & \\
 & foldright(F,A,H::T) & = & F(H, foldright(F,A,T)) & \text{(C)} \\
L \neq nil \rightarrow & foldright(F,A,L) & = & F(head(L), foldright(F,A,tail(L))) & \text{(D)}
\end{array}$$

Function: $foldright\_tr$

$$\begin{array}{rrcll}
 & foldright\_tr & : & (\beta \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha \rightarrow list(\beta) \rightarrow \alpha & \\
 & foldright\_tr(F,A,nil) & = & A & \\
 & foldright\_tr(F,A,H::T) & = & foldright\_tr(F,F(H,A),T) & \text{(C)} \\
L \neq nil \rightarrow & foldright\_tr(F,A,L) & = & foldright\_tr(F,F(head(L),A),tail(L)) & \text{(D)}
\end{array}$$

Function: $half$

$$\begin{array}{rrcll}
 & half & : & nat \rightarrow nat & \\
 & half(0) & = & 0 & \\
 & half(s(0)) & = & 0 & \\
 & half(s(s(X))) & = & s(half(X)) & \text{(C)} \\
X \neq 0 \wedge X \neq s(0) \rightarrow & half(X) & = & s(half(p(p(X)))) & \text{(D)}
\end{array}$$

Function: len

`olength` in *Dynamis*.

$$\begin{array}{rrcll}
 & len & : & list(\tau) \rightarrow nat & \\
 & len(nil) & = & 0 & \\
 & len(H::T) & = & s(len(T)) & \text{(C)} \\
L \neq nil \rightarrow & len(L) & = & s(len(tail(L))) & \text{(D)}
\end{array}$$

Function: $\leq$

`leq` in *Dynamis*.

$$
\begin{array}{rrcll}
 & \leq & : & nat \rightarrow nat \rightarrow bool & \\
 & 0 \leq Y & = & true & \\
 & s(X) \leq 0 & = & false & \\
 & s(X) \leq s(Y) & = & X \leq Y & \text{(C)} \\
X \neq 0 \wedge Y \neq 0 & X \leq Y & = & p(X) \leq p(Y) & \text{(D)}
\end{array}
$$

Function: $+$ ($plus$)

$$
\begin{array}{rrcll}
 & + & : & nat \rightarrow nat \rightarrow nat & \\
 & 0 + Y & = & Y & \\
 & s(X) + Y & = & s(X + Y) & \text{(C)} \\
X \neq 0 \rightarrow & X + Y & = & s(p(X) + Y) & \text{(D)}
\end{array}
$$

Function: odd

$$
\begin{array}{rrcll}
 & odd & : & nat \rightarrow bool & \\
 & odd(0) & = & false & \\
 & odd(s(0)) & = & true & \\
 & odd(s(s(X))) & = & odd(X) & \text{(C)} \\
X \neq 0 \wedge X \neq s(0) \rightarrow & odd(X) & = & odd(p(p(X))) & \text{(D)}
\end{array}
$$

Function: *oddelems*

$$\begin{aligned}
oddelems &: list(\tau) \to list(\tau) \\
oddelems(nil) &= nil \\
oddelems(X :: nil) &= X :: nil \\
oddelems(X :: Y :: L) &= X :: oddelems(L) && \text{(C)} \\
L \neq nil \wedge tail(L) \neq nil \to oddelems(L) &= head(L) :: oddelems(tail(tail(L))) && \text{(D)}
\end{aligned}$$

Function: *quot*

$$\begin{aligned}
& quot &&: nat \to nat \to nat \\
& quot(X,0) &&= 0 \\
Y \neq 0 \wedge \neg(Y \leq X) \to \quad & quot(X,Y) &&= 0 \\
Y \neq 0 \to \quad & quot(X+Y,Y) &&= s(quot(X,Y)) && \text{(C)} \\
Y \neq 0 \wedge leq(Y,X) \to \quad & quot(X,Y) &&= s(quot(X-Y,Y)) && \text{(D)}
\end{aligned}$$

Function: *rev*

`orev` in *Dynamis*.

$$\begin{aligned}
& rev &&: list(\tau) \to list(\tau) \\
& rev(nil) &&= nil \\
& rev(H :: T) &&= rev(T) <> (H :: nil) && \text{(C)} \\
L \neq nil \to \quad & rev(L) &&= rev(tail(L)) <> (head(L) :: nil) && \text{(D)}
\end{aligned}$$

Function: *rotate*

$$\begin{array}{rrcll}
 & rotate & : & nat \rightarrow list(\tau) \rightarrow list(\tau) & \\
 & rotate(0,L) & = & L & \\
 & rotate(X,nil) & = & nil & \\
 & rotate(s(X),H :: T) & = & rotate(X,T <> (H :: nil)) & \text{(C)} \\
X \neq 0 \wedge L \neq nil \rightarrow & rotate(X,L) & = & rotate(p(X),tail(L) <> (head(L) :: nil)) & \text{(D)}
\end{array}$$

Function: *sum*

$$\begin{array}{rrcll}
 & sum & : & list(nat) \rightarrow nat \rightarrow nat & \\
 & sum(nil,X) & = & X & \\
 & sum(H :: T,X) & = & sum(T,X+H) & \text{(C)} \\
L \neq nil \rightarrow & sum(L,X) & = & sum(tail(L),X+head(L)) & \text{(D)}
\end{array}$$

Function: $\times$ (*times*)

$$\begin{array}{rrcll}
 & \times & : & nat \rightarrow nat \rightarrow nat & \\
 & 0 \times Y & = & 0 & \\
 & s(X) \times Y & = & (X \times Y) + Y & \text{(C)} \\
X \neq 0 \rightarrow & X \times Y & = & (p(X) \times Y) + Y & \text{(D)}
\end{array}$$

Appendix C

Dynamis Documentation

This appendix documents several aspects of the *Dynamis* system: how to run it, the lower level methods that were not fully covered in Chapter 10 and the λProlog predicates used in method pre- and postconditions. We also detail the minor changes that were made to the main λ*Clam* source code in order to integrate *Dynamis*'s code.

C.1 Running *Dynamis*

There are two predicates that can be used at the *Dynamis* command line to plan theorems. Both are built on top of λ*Clam*'s `claudio_plan` (version 4.0).

```
plan_and_display:  meth -> query -> o
```

Initiates planning of the given query with the given method, and displays the plan if one is found.

```
dynamis_plan:  meth -> query -> int -> style -> o
```

Loads a predetermined configuration of rewrite and wave rules, then initiates planning of the given query with the given method. If successful it displays the plan.

The configuration is determined by the style, which is the function definition style to be used (constructor or destructor), and an integer indicating the lemma set to be loaded. This information must be hard-coded beforehand. For example, the command:

```
dynamis_plan dynamis_crit comp 1 constructor.
```

relies on the following two hard-coded facts:

```
defn_rules plus constructor [plus1, plus2] [plus2].
needs comp 1 constructor [plus_right1, plus_right2] [plus_right2].
```

C.2 Step Case Methods

Method: embed_hypothesis

The embed_hypothesis method, shown in Figure C.1, takes an unannotated step case goal with a single hypothesis and adds embeddings for hypothesis and conclusion.

The preconditions embed the skeleton Skel into both the hypothesis IndHyp and conclusion Conc, with embeddings EH1 and EC1 respectively. The postconditions merge and orient wave-fronts to give embeddings EH2 and EC2, and then weigh them using the number of wave-fronts (HW) and the wave measure (Out, In) respectively. The method's subgoal has a single annotated induction hypothesis NewAnnHyp.

Method: redo_embeddings

Shown in Figure C.1 is the redo_embeddings, which recomputes the conclusion embedding during the middle of the step case. It is required after the speculation critic

Method: `embed_hypothesis`

Goal: `(caseSchema Cond Hyps (preRippleHyps Skel [IndHyp]) Conc)`

Pre:

```
(once (embedding Skel EH1 IndHyp,
       embedding Skel EC1 Conc))
```

Post:

```
(tidy_hyp_context EH1 EH2 HW,
 tidy_conc_context EC1 outward EC2 Out In,
 AnnHyp = (annHyp IndHyp Skel EH2 HW EC2 Out In))
```

SubGoal: `(caseSchema Cond Hyps (rippleHyps [AnnHyp]) Conc)`

Method: `redo_embeddings`

Goal: `(caseSchema Case Hyps (rippleHyps [AnnHyp]) Conc)`

Pre:

```
(AnnHyp = (annHyp Hyp Skel EH HW _ _ _),
 embedding Skel EC1 Conc)
```

Post:

```
(tidy_conc_context EC1 outward EC2 Out In,
 NewAnnHyp = (annHyp Hyp Skel EH HW EC3 Out In))
```

SubGoal: `(caseSchema Case Hyps (rippleHyps [NewAnnHyp]) Conc)`

Figure C.1: The embedding methods: embed_hypothesis and redo_embeddings.

Method: `(definite_ripple Rule Ad)`

Goal: `(caseSchema Case Hs (rippleHyps [AnnHyp]) Conc)`

Pre:

```
(AnnHyp = (annHyp Hyp Skel EH HW EC1 Out In),
 wave_rule_list Rules,
 rewrite_inner (rewr_list Rules rewr_match) Rule _ Conc NewC Cond Ad,
 not (rulestyle Rule destructor),
 reverse Ad At,
 subterm_embed undir Rule At [] EC1 EC2 Skel NewC bool,
 tidy_conc_context EC2 anydir EC3 NewOut NewIn,
 measure_less Out In NewOut NewIn)
```

Post:

```
(NewAnnHyp = (annHyp Hyp Skel EH HW EC3 NewOut NewIn),
 Main = (caseSchema Case Hs (rippleHyps [NewAnnHyp]) NewC),
 condition_goal Cond Case Hs
               (c\ (caseSchema Case Hs sideCond c)) Main SubGoal)
```

Subgoal: `SubGoal`

Figure C.2: Clause 1 of the `definite_ripple` method.

has instantiated a meta-variable, and the embedding in the main step case plan branch needs updating to reflec this.

The method works in a similar way to `embed_hypothesis`, but leaves the hypothesis embedding untouched.

Method: `definite_ripple`

The method has two clauses, shown in Figure C.2 and Figure C.3: the first for wave-measure decreasing ripples, the second for creational ripples that remove hypothesis wave-fronts. In both clauses, the conclusion is rewritten with the relation `rewr_match`, which does not instantiate metavariables. The rewritten subterm is reembedded with

Method: `(definite_ripple Rule Ad)`

Goal: `(caseSchema Case Hs (rippleHyps [AnnHyp]) Conc)`

Pre:

```
(AnnHyp = (annHyp Hyp Skel EH1 HW EC1 _ _),
 wave_rule_list Rules,
 rewrite_inner (rewr_list Rules rewr_match) Rule _ Conc NewC Cond Ad,
 not (rulestyle Rule constructor),
 reverse Ad At,
 subterm_embed undir Rule At [] EC1 EC2 Skel NewC bool,
 cancel_context 0 At Skel NewSkel Hyp EH1 EH2 NewC EC2 EC3,
 reembed NewSkel bool Hyp bool EH2 EH3,
 tidy_hyp_context EH3 EH4 NewHW,
 NewHW < HW)
```

Post:

```
(tidy_conc_context EC3 outward EC4 Out In,
 NewAnnHyp = (annHyp Hyp NewSkel EH4 NewHW EC4 Out In),
 Main = (caseSchema Case Hs (rippleHyps [NewAnnHyp]) NewC),
 condition_goal Cond Case Hs
                (c\ (caseSchema Case Hs sideCond c)) Main SubGoal)
```

SubGoal: `SubGoal`

Figure C.3: Clause 2 of the `definite_ripple` method.

the corresponding subterm of the skeleton *Skel*.

In the first clause, the preconditions check the wave-measure is reduced. In the second, neutralisation is performed to give an expanded skeleton `NewSkel`, and new embeddings for the hypothesis (`EH2`) and the conclusion (`EC3`). The weight of the hypothesis `NewHW` (the number of wave-fronts) is measured — it must be less than the old weight `HW`.

Both clauses disallow certain rewrite rule styles in order to prevent unwanted ripple steps. Destructor style rules are typically used in creational steps, and so the first

Method: `meta_ripple`

Goal: `(caseSchema Cond Hyps (rippleHyps [AnnHyp1]) Conc)`

Pre:
```
(AnnHyp1 = (annHyp Hyp Skel EH HW EC1 Out In),
 embedding Skel EC2 Conc,
 tidy_conc_context EC2 anydir EC3 Newout In,
 measure_less Out In NewOut In)
```

Post: `(AnnHyp2 = (annHyp Hyp Skel EH HW EC3 NewOut In)`

Goal: `(caseSchema Cond Hyps (rippleHyps [AnnHyp2]) Conc)`

Figure C.4: The `meta_ripple` method.

clause excludes these rules. Constructor style rules are typically used in wave measure reducing steps, and so the second clause excludes them. These restrictions only affect definitional rewrites — lemmas are always allowed.

Method: `meta_ripple`

The `meta_ripple` method is shown in Figure C.3. A meta-ripple step reduces the wave measure of the embedding without rewriting the underlying term. The preconditions simply reembed the step case skeleton in the conclusion. The new embedding `EC3` must be less than the original embedding `EC1`.

Method: `forwards_ripple`

The `forwards_ripple` method is used after the speculation critic has been applied. It ripples inwards the ‘missing’ wave fronts inserted by the critic, so that a suitable instantiation that unblocks the main ripple proof can be found with the `speculate_wavefronts`

Method: `(forwards_ripple Rule Ad Ripples)`

Goal: `(caseSchema Case Hyps (blockedGoal Skel E1 In) Conc)`

Pre:

```
(wave_rule_list Rules,
 rewrite_inner (rewr_list Rules rev_rewr_match)
               Rule _ Conc NewC trueP Ad,
 reverse Ad At,
 subterm_embed undir (backwards Rule) At [] E1 E2 Skel NewC bool,
 tidy_conc_context E2 inward E3 nil NewIn,
 measure_less nil In nil NewIn)
```

Post: `(varadd (definite_ripple Rule Ad) Ripples)`

SubGoal: `(caseSchema Case Hyps (blockedGoal Skel E3 NewIn) NewC)`

Figure C.5: The `forwards_ripple` method.

method (see below). This strategy is implemented in the `ripple_in_and_speculate` method (see Figure 10.12).

The `forwards_ripple` method is shown in Figure C.5. Its preconditions are similar to the first clause of the `definite_ripple` method (see Figure C.2). The conclusion `Conc` is rewritten to `NewC`, and the embedding is updated from `E1` to `E3`. The key difference from standard rippling is that we are constructing the proof in reverse, as we are looking for an instantiation *earlier in the proof* which would have unblocked the current ripple goal. Confusingly, proof search in λ*Clam* is normally backwards (from theorem to axioms) so by reversing the proof direction we are now going forwards (from axioms to theorems). Hence the name of the method.

Because its a *reverse* ripple method:

- the method ripples wave fronts inwards.

Method: `(speculate_wavefronts Ripples RipplePlan)`

Goal: `(caseSchema _ _ (blockedGoal Goal E _) Conc)`

Pre:

```
(fully_rippled_subs Conc E [] Subs,
 speculate_subs Subs)
```

Post: `(compose_plan_steps Ripples RipplePlan)`

SubGoal: `trueGoal`

Figure C.6: The `speculate_wavefronts` method.

- the rewrite relation is used *backwards*: `rev_rewr_match` instead of `rewr_match`.

- The wave measure must *increase*.

Method: `speculate_wavefronts`

The `speculate_wavefronts` method is applied when the `ripple_in_and_speculate` method (see Figure 10.12) has exhaustively rippled in the ‘missing’ wave fronts so that they surround meta-variables. It is shown in Figure C.6. The method tries to find an instantiation of the goal’s meta-variables which would produce the fully rippled-in wave fronts.

Its preconditions find the pairings of wave front/meta-variable `Subs`, and then computes a set of instantiations if one exists. The postconditions instantiate `RipplePlan` with the ripple steps that lead to the instantiation, so that they can be applied in reverse in the main step case plan branch.

Method: `strong_fertilise`

Goal: `(caseSchema _ _ (rippleHyps [AnnHyp]) Conc)`

Pre:

```
(AnnHyp = (annHyp Hyp _ _ 0 _ _ _),
 rewrite_match_with_hyp equiv Hyp trueP Conc trueP [],
 Conc = Hyp)
```

Post: `true`

Subgoal: `trueGoal`

Figure C.7: The `strong_fertilise` method.

Method: `strong_fertilise`

Strong fertilisation comes in two forms. Firstly, where the hypothesis and conclusion are unified, via the `strong_fertilise` method. Secondly, where the hypothesis appears as a subterm of the conclusion (see the `strong_fertilise_prop` method in the next section). The `strong_fertilise` method is shown in Figure C.7.

The preconditions first check that the induction hypothesis `Hyp` contains no wave fronts, i.e. the hypothesis measure equals zero. The conclusion is then rewritten to `trueP` using the rewrite rule `Hyp` ⇒ `trueP`, without instantiating the conclusion's meta-variables. The method does this before it unifies the two propositions as a check that they are unifiable. The check is made because higher order unification often diverges if hypothesis and conclusion are non-unifiable. If the rewrite succeeds then they are unified.

Method: `strong_fertilise_prop`

Goal: `(caseSchema Case Hyps (rippleHyps AnnHyp) Conc)`

Pre:

```
(not (Conc = (app F _), (F = eq; F = iff)),
 AnnHyp = (annHyp Hyp _ _ 0 (econtext _ _ EC) _ _),
 fully_rippled_in EC,
 rewrite_match_with_hyp equiv Hyp trueP Conc NewConc _)
```

Post: `true`

Subgoal: `(caseSchema Case Hyps postRippleHyps NewConc)`

Figure C.8: The `strong_fertilise_prop` method.

Method: `strong_fertilise_prop`

The `strong_fertilise_prop` method is shown in Figure C.8. It performs the second form of strong fertilisation, namely where the induction hypothesis matches a subterm of the conclusion. The preconditions are similar to `strong_fertilise`, except that a check is made that all wave fronts are fully rippled out or in. A residual conclusion remains as a subgoal.

Method: `weak_fertilise`

The `weak_fertilise` method, shown in Figure C.9, performs weak fertilisation, where the induction hypothesis is used to rewrite one side of a binary predicate conclusion — either `eq` or iff. The method is parameterised by a flag indicating whether the induction hypothesis has been ‘flipped’ before being applied. A residual subgoal is left after fertilisation.

Method: `(weak_fertilise Swap)`

Goal: `(caseSchema Case Hyps (rippleHyps [AnnHyp]) (app F (tuple [A, B])))`

Pre:

```
((F = eq; F = iff),
 swap A B Swap A2 B2,
 AnnHyp = (annHyp Hyp _ EH _ EC _ _),
 Hyp = (app F (tuple [X, Y])),
 EH = (eapp [] (ebase [1]) (etuple [2] [XH, YH])),
 EC = (eapp [] (ebase [1]) (etuple [2] [XC, YC])),
 swap X Y Swap X2 Y2,
 swap XH YH Swap XH2 _,
 swap XC YC Swap XC2 _,
 hyp_weight XH2 0 0,
 fully_rippled XC2,
 rewrite_match_with_hyp equiv X2 Y2 A2 NewA2 _)
```

Post: `(swap NewA2 B2 Swap A3 B3)`

Subgoal: `(caseSchema Case Hyps postRippleHyps (app F (tuple [A3, B3])))`

Method: `replace_metavariables`

Goal: `(caseSchema _ Hyps _ Conc)`

Pre:

```
(metavars Conc [] MVs bool,
 abstract_meta_vars Conc MVs [] NewConc)
```

Post: `true`

Subgoal: `(seqGoal (Hyps >>> NewConc))`

Figure C.9: The `weak_fertilise` and `replace_metavariables` methods.

Method: `(construct_wf_goals Consts)`

Goal: `(stepReduces Hyps KB)`

Pre:

```
(dkb_cases KB Cases,
 dkb_constraints KB Consts,
 dkb_types KB Types,
 length Types N,
 setup_constraints N Consts,
 list_to_goal Cases (wellfound_goals Hyps Consts) RedGoals)
```

Post: `true`

Subgoal: `RedGoals`

Figure C.10: The `construct_wf_goals` method.

Method: `replace_metavariables`

The `replace_metavariables` method is shown in Figure C.9. It finds the metavariables in a schematic goal and replaces them with universally quantifed variables. The quantifiers appear at the top of the conclusion.

C.3 Wellfoundedness Methods

The theory and implementation of the wellfoundedness strategy was discussed in §6.5 and §10.4. This section briefly describes the low-level methods that did not appear in §10.4.

Method: `(ignore_position N)`

Goal: `(redGoal N Consts _ _ _ _ _)`

Pre:

```
(varadd (ignore N) Consts,
 check_satisfiable Consts)
```

Post: `true`

Subgoal: `trueGoal`

Figure C.11: The `ignore_position` method.

Method: `construct_wf_goals`

The `construct_wf_goals` method is shown in Figure C.10. It transforms the dummy meta-level `stepReduces` goal to a conjunction of wellfoundedness goals for a step case. If this is the first step case then the method also posts `measure` constraints on the step case, indicating the measure function used for each induction position. At this point the unknown measures are represented by meta-variables.

Method: `ignore_position`

The `ignore_position` method is shown in Figure C.11. It can be applied to any wellfoundedness goal, irrespective of its validity, providing this leaves at least one wellfoundedness goal that has not been ignored. The method's preconditions post an `ignore` constraint for the corresponding induction position, and checks the current constraints are still satisfiable, indicating a valid position remains.

Method: `(begin_estimation N)`

Goal: `(redGoal N Consts Hyps Cond A (induce M) B)`

Pre:

```
(varmemb (const_disj MConsts) Consts,
 varmemb (measure N M) MConsts,
 not (varmemb (ignore N) Consts))
```

Post: `(extract_condition Cond Cond2)`

Subgoal:

```
((estGoal M A B Diff)
 ** (seqGoal (Hyps >>> (app imp (tuple [Cond2, Diff])))))
```

Figure C.12: The `begin_estimation` method.

C.3.1 Estimation Methods

This section describes the low-level methods used to implement Walther's estimation method, employed by our strategy to discharge wellfoundedness goals. These methods are organised into a strategy by the `estimation_strat` method (see Figure 10.17).

Method: `begin_estimation`

The `begin_estimation` method is shown in Figure C.12. It takes a wellfoundedness goal and sets up an estimation proof, consisting of two subgoals. Firstly, an estimation goal, which claims i) that one term is equal to or smaller than another under a measure and ii) that it being strictly smaller is equivalent to `Diff`. Secondly, a goal stating that the difference equivalent `Diff` is implied by the corresponding case conditions.

Method: `lower_estimate`

Goal: `(estGoal M L R (app or (tuple [DiffLit, DiffEquiv])))`

Pre:

```
(not (headvar_osyn R),
 R = (app F Args),
 lower_arg_bound F N M DiffPred)
```

Post:

```
(nth_arg Args N Arg,
 DiffPred Args DiffLit)
```

Subgoal: `(estGoal M L Arg DiffEquiv)`

Method: `upper_estimate`

Goal: `(estGoal M L R (app or (tuple [DiffLit, DiffEquiv])))`

Pre:

```
(not (headvar_osyn L),
 L = (app F Args),
 upper_arg_bound F N M DiffPred)
```

Post:

```
(nth_arg Args N Arg,
 DiffPred Args DiffLit)
```

Subgoal: `(estGoal M Arg R DiffEquiv)`

Figure C.13: The `lower_estimate` and `upper_estimate` methods.

Method: `trivial_estimate`

Goal: `(estGoal M X Y falseP)`

Pre:

```
(not (headvar_osyn M),
 (headvar_osyn X; (not (headvar_osyn X), obj_atom X)),
 (headvar_osyn Y; (not (headvar_osyn Y), obj_atom Y)),
 X = Y)
```

Post: `true`

Subgoal: `trueGoal`

Figure C.14: The `trivial_estimate` method.

Method: `lower_estimate`

The `lower_estimate` method is shown in Figure C.13. It applies the lower estimation rule (see §6.5.4), i.e. it takes an estimation goal where the 'smaller' term has a top functor f that is lower argument bounded, and removes this functor to form the subgoal. The difference equivalent is instantiated to a disjunction of the difference literal for f and a fresh meta-variable.

Method: `upper_estimate`

The `upper_estimate` method, shown in Figure C.13, implements Walther's original form of estimation. It is analogous to the `lower_estimate` method, except an upper argument bounded function is stripped off the 'larger' term of the inequality.

Method: `abstract_metavars`

Goal: `Goal`

Pre:

```
(Goal = (seqGoal (Hyps >>> Conc)),
 fold_left (v1\ t\ v2\ (metavars t v1 v2 bool)) [] [Conc|Hyps] Vars,
 abstract_goal Goal Vars [] AbsGoal)
```

Post: `true`

Subgoal: `AbsGoal`

Figure C.15: The `abstract_metavars` method.

Method: `trivial_estimate`

The `trivial_estimate` method is shown in Figure C.14. The method discharges trivial estimation goals. The preconditions check that each side of the inequality is either a meta-variable or an atom, and unifies the two sides. The difference equivalent is instantiated to `falseP`, indicating that this inequality is not strict.

Method: `abstract_metavars`

The `abstract_metavars` method is shown in Figure C.15. The method collects together the meta-variables in a sequent goal and replaces them with variables. These variables are univerally quantified in the subgoal.

The purpose of this method is two remove any meta-variables from the difference equivalent goal before rewriting is applied.

Method: `set_conditions`

Goal: `(caseGoal C M Hs F)`

Pre: `(inst_var_literal F)`

Post: `true`

Subgoal: `(caseGoal C M Hs F)`

Figure C.16: The `set_conditions` method.

C.4 Case Synthesis Methods

The implementation of the wellfoundedness strategy was discussed in §10.4. This section briefly describes the low-level methods that did not appear there.

Method: `set_conditions`

The `set_conditions` method is shown in Figure C.16. It instantiates the meta-variable representing any unknown case conditions, so that the subsequent case strategy does not accidently instantiate it.

Method: `case_equiv`

Figure C.17 shows the `case_equiv` method. This method simplifies the case formulae during the case exhaustiveness proof using equivalence preserving steps, i.e. rewriting with certain rules or removing ‘solved’ disjuncts. These steps do not need to be backtracked over, so several are applied together within the `equiv_case` predicate, which prevents this happening.

Method: `(case_equiv Rules)`
Goal: `Goal`

Pre: `(equiv_case Goal NewGoal Rules)`

Post: `true`

Subgoal: `NewGoal`

Figure C.17: The `case_equiv` method.

This is an unelegant way to represent these proof steps — individual method applications would have been better. However, without a cut methodical available in λ*Clam* this is the only way to prevent needless backtracking through such sequences of simplification (see also §10.3.4).

Method: `exists_casesplit`

The `exists_casesplit` method, shown in Figure C.18, applies the existential case split method described in §5.4.2. Although its preconditions seem quite complex, they implement the heuristics described in full in §5.6.

Method: `case_induction`

The `case_induction` method is shown in Figure C.19. It applies induction to the case formula during the case exhaustiveness proof, following the heuristics described in §5.6. The method consists of two clauses, corresponding to the two different contexts in which it was determined induction could be applied.

After induction, rippling and fertilisation are applied to the case formula, using the `case_ripple` and `case_fertilisation` methods shown in Figure C.20.

Method: `(exists_casesplit Scheme)`

Goal: `(aseGoal CasePair Missing Hyps Conc)`

Pre:

```
(not (rewritable Conc),
 metavars Conc [] Vars bool,
 memb (otype_of Var Type) Vars,
 not (headvar_osyn Type),
 once (junctive or Disjunct Conc,
       junctive and Conjunct Conc,
     ((Conjunct = (app eq (tuple [UTerm, (app F Args)])),
       defined_function F (ATypes arrow _),
       contains_metavar Args ATypes Var Type,
       not (is_univ_var UTerm _));
      (not (Conjunct = (app eq _)),
       contains_metavar Conjunct bool Var Type))),
       exhaustive Scheme Type ExCases,
       for_each ExCases (some_case (c\
         (sigma t\
           (sigma r\ (not (not (case_term c Var, rewritable Conc))))))))
```

Post:

```
(map_junction or (split_exist Var ExCases) Conc NewConc,
 mappred Hyps (reembed_casehyp NewConc) NewHyps)
```

Subgoal: `(caseGoal CasePair Missing NewHyps NewConc)`

Figure C.18: The `exists_casesplit` method.

Method: `(case_induction Scheme)`
Goal: `Goal`

Pre:

```
(Goal = (caseGoal _ _ _ Conc),
 not (rewritable Conc),
 junctive or Disjunct Conc,
 universal_vars Disjunct [] UVars,
 subset [] IndSet UVars,
 mappred2 IndSet (x\ y\ z\ (x = (otype_of y z))) IndVars Types,
 case_scheme Scheme Types IndVars Goal SubGoals,
 for_each_goal SubGoals (g\ (sigma c\ (get_conc g c, rewritable c))))
```

Post: `(map_goal SubGoals rename_and_embed NewSubGoals)`

Subgoal: `NewSubGoals`

Method: `(case_induction Scheme)`

Goal: `Goal`

Pre:

```
(Goal = (caseGoal _ _ _ Conc),
 not (rewritable Conc),
 junctive or Disjunct Conc,
 universal_vars Disjunct [] UVars,
 subset [] IndSet UVars,
 mappred2 IndSet (x\ y\ z\ (x = (otype_of y z))) IndVars Types,
 case_scheme Scheme Types IndVars Goal SubGoals,
 not (for_each_goal SubGoals (g\ (sigma c\
                                 (get_conc g c, rewritable c)))),
 once (memb (otype_of UVar Type) IndSet,
       junctive and (app eq (tuple [UVar, (app F _)])) Disjunct,
       not (headvar_osyn F),
       defined_function F (_ arrow Type)))
```

Post: `(map_goal SubGoals rename_and_embed NewSubGoals)`

Subgoal: `NewSubGoals`

Figure C.19: The two clauses of the `case_induction` method.

Method: `(case_ripple Rule)`

Goal: `(caseGoal Case Missing Hyps Conc)`

Pre:

```
(wave_rule_list Rules,
 rewrite_outer (rewr_list Rules rewr_match) Rule _ Conc NewConc trueP Ad,
 nth Hyps N (caseHyp Hyp E1 Out1) Rest,
 reverse Ad At,
 subterm_embed undir Rule At [] E1 E2 Hyp NewConc bool,
 tidy_conc_context E2 outward E3 Out3 nil,
 measure_less Out1 nil Out3 nil)
```

Post: `(nth NewHyps N (caseHyp Hyp E3 Out3) Rest)`

SubGoal: `(caseGoal Case Missing NewHyps NewConc)`

Method: `case_fertilisation`

Goal: `(caseGoal Case Missing Hyps Conc)`

Pre:

```
(memb (caseHyp Hyp _ _) Hyps,
 junctive or Disj Hyp,
 junctive and Conj Disj,
 case_fert Conj Conc NewConc)
```

Post: `true`

SubGoal: `(caseGoal Case Missing Hyps NewConc)`

Figure C.20: The `case_ripple` and `case_fertilisation` methods.

Method: `remove_case_hyps`

Goal: `(caseGoal C M Hs F)`

Pre: `true`

Post: `(remove_casehyps Hs NewHs)`

SubGoal: `(caseGoal C M NewHs F)`

Figure C.21: The `remove_case_hyps` method.

Method: `remove_case_hyps`

The `remove_case_hyps` method is shown in Figure C.21. It strips the case formula of any inductive hypotheses, if rippling and/or fertilisation fail. This happens when induction was used to achieve a case split, rather than a genuine inductive argument (see §5.6).

Method: `trivial_case`

The `trivial_case` method is shown in Figure C.22. It discharges trivially case formulae during the case exhaustiveness proof.

Method: `missing_case`

The `missing_case` method is shown in Figure C.22. The method discharges trivially false case formulae during the case exhaustiveness proof, and identifies the missing proof case that corresponds to this failed subgoal. The case is added to the list of missing cases `Missing`.

Method: `trivial_case`

Goal: `(caseGoal _ _ _ trueP)`

Pre: `true`

Post: `true`

Subgoal: `trueGoal`

Method: `(missing_case AbsCase)`

Goal: `(caseGoal (case Cond _ Term) Missing _ falseP)`

Pre: `true`

Post:

```
(univ_vars Cond [] Vars,
 univ_vars Term Vars Vars2,
 abstract_case Vars2 [] (case Cond _ Term) AbsCase,
 varadd AbsCase Missing)
```

Subgoal: `trueGoal`

Figure C.22: The `trivial_case` and `missing_case` methods.

C.5 Base Case Methods

The base case strategy was described in §10.6, and this section provides definitions for the low-level methods which were not given there. These are: `rewrite`, `rewrite_equiv`, `rewrite_nonequiv` and `normalise` (see Figure C.23 and Figure C.24).

Method: `rewrite`

```
(then_meth (normalise all_i_nf)
(then_meth (some_meth rewrite_equiv)
           (try_meth
              (repeat_meth
                 (then_meth (some_meth rewrite_nonequiv)
                            (some_meth rewrite_equiv))))))
```

Method: `(rewrite_equiv Rules)`
Goal: `Goal`

Pre:

```
(equiv_simplification Goal SubGoal Rules)
```

Post: `true`

SubGoal: `SubGoal`

Method: `(rewrite_nonequiv Rule)`
Goal: `(seqGoal (H >>> C))`

Pre:

```
(sym_eval_rewrites_list Rules,
 rewrite_outer (rewr_list Rules rewr_unif) Rule Dir C NewC trueP _,
 not (Dir = equiv))
```

Post:

```
(condition_goal Cond trueP H (c\ (seqGoal (H >>> c)))
                (seqGoal (H >>> NewC)) SubGoal)
```

SubGoal: `SubGoal`

Figure C.23: The `rewrite`, `rewrite_equiv` and `rewrite_nonequiv` methods.

Method: `(normalise NF)`
Goal: `G`

Pre: `(NF G G2)`

Post: `true`

SubGoal: `G2`

Figure C.24: The `normalise` method.

Appendix D

Dynamis Traces

This online appendix provides in the full trace files for the evaluation theorems of Chapter 11. It can be found at `http://homepages.inf.ed.ac.uk/s9362054/thesis`

Bibliography

[Allen et al., 2000] Allen, S., Constable, R., Eaton, R., Kreitz, C., and Lorigo, L. (2000). The Nuprl open logical environment. In McAllester, D. A., editor, *Automated Deduction - CADE-17: 17th International Conference on Automated Deduction, Pittsburgh, PA, USA, June 2000, Proceedings*, volume 1831 of *Lecture Notes in Artificial Intelligence*, pages 170–176. Springer.

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

[Baader and Nipkow, 1998] Baader, F. and Nipkow, T. (1998). *Term Rewriting and All That*. Cambridge University Press.

[Bachmair, 1991] Bachmair, L. (1991). *Canonical Equational Proofs*. Birkhauser.

[Bajcsy, 1993] Bajcsy, R., editor (1993). *Proceedings of the 13th International Joint Conference on Artificial Intelligence, Chambéry, France*, volume 1. Morgan Kaufmann.

[Basin and Walsh, 1993] Basin, D. and Walsh, T. (1993). Difference unification. In [Bajcsy, 1993], pages 116–122. Also available as Technical Report MPI-I-92-247, Max-Planck-Institut für Informatik.

[Basin and Walsh, 1996] Basin, D. and Walsh, T. (1996). A calculus for and termination of rippling. *Journal of Automated Reasoning*, 16(1–2):147–180.

[Benzmüller et al., 1997] Benzmüller, C., Cheikhrouhou, L., Fehrer, D., Fiedler, A., Huang, X., Kerber, M., Kohlhase, M., Konrad, K., Meier, A., Melis, E., Schaarschmidt, W., Siekmann, J. H., and Sorge, V. (1997). Ωmega: Towards a mathematical assistant. In McCune, W., editor, *Automated Deduction — CADE-14: 14th International Conference on Automated Deduction, Townsville, North Queensland, Australia, July 1997, Proceedings*, volume 1249 of *Lecture Notes in Artificial Intelligence*, pages 252–255. Springer.

[Bouhoula et al., 1992] Bouhoula, A., Kounalis, E., and Rusinowitch, M. (1992). SPIKE, an automatic theorem prover. In [Voronkov, 1992], pages 460–462.

[Boulton et al., 1998] Boulton, R., Slind, K., Bundy, A., and Gordon, M. J. C. (1998). An interface between *Clam* and HOL. In Grundy, J. and Newey, M. C., editors, *Theorem Proving in Higher Order Logics, 11th International Conference, TPHOLs'98, Canberra, Australia, September 1998, Proceedings*, volume 1479 of *Lecture Notes in Computer Science*, pages 87–104. Springer.

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

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

[Boyer and Moore, 1992] Boyer, R. S. and Moore, J. S. (1992). On the difficulty of automating inductive reasoning. In *Proceedings of the Workshop on Mathematical Induction, CADE-11*.

[Bundy, 1988] Bundy, A. (1988). The use of explicit plans to guide inductive proofs. In [Lusk and Overbeek, 1988], pages 111–120.

[Bundy, 1991] Bundy, A. (1991). A science of reasoning. In Lassez, J. L. and Plotkin, G., editors, *Computational Logic: Essays in Honor of Alan Robinson*, pages 178–198. MIT Press.

[Bundy, 1994] Bundy, A., editor (1994). *Automated Deduction — CADE-12: 12th International Conference on Automated Deduction, Nancy, France, June 1994, Proceedings*, volume 814 of *Lecture Notes in Artificial Intelligence*. Springer.

[Bundy, 2001] Bundy, A. (2001). The automation of proof by mathematical induction. In Robinson, A. and Voronkov, A., editors, *Handbook of Automated Reasoning*, volume 1, pages 845–911. Elsevier Science.

[Bundy and Green, 1996] Bundy, A. and Green, I. (1996). An experimental comparison of rippling and exhaustive rewriting. Research paper 836, Department of Artificial Intelligence, University of Edinburgh.

[Bundy et al., 1990a] Bundy, A., Smaill, A., and Hesketh, J. (1990a). Turning eureka steps into calculations in automatic program synthesis. In Clarke, S. L. H., editor, *Proceedings of UK IT 90*, pages 221–226.

[Bundy et al., 1993] Bundy, A., Stevens, A., van Harmelen, F., Ireland, A., and Smaill, A. (1993). Rippling: A heuristic for guiding inductive proofs. *Artificial Intelligence*, 62:185–253.

[Bundy et al., 1991] Bundy, A., van Harmelen, F., Hesketh, J., and Smaill, A. (1991). Experiments with proof plans for induction. *Journal of Automated Reasoning*, 7(3):303–324.

[Bundy et al., 1989] Bundy, A., van Harmelen, F., Hesketh, J., and Stevens, A. (1989). A rational reconstruction and extension of recursion analysis. In Sridharan, N. S., editor, *Proceedings of the 11th International Joint Conference on Artificial Intelligence, Detroit, USA*, pages 359–365. Morgan Kaufmann.

[Bundy et al., 1990b] Bundy, A., van Harmelen, F., Horn, C., and Smaill, A. (1990b). The Oyster-Clam system. In [Stickel, 1990], pages 647–648.

[Burstall, 1969] Burstall, R. (1969). Proving properties of programs by structural induction. *The Computer Journal*, 12(1):41–48.

[Burton, 1988] Burton, D. M. (1988). *The History of Mathematics - An Introduction*. William C. Brown, Dubuque, Iowa.

[Castaing, 1985] Castaing, J. (1985). How to facilitate the proof of theorems by using the induction-matching, and by generalization. In [Joshi, 1985], pages 1208–1213.

[Cheikhrouhou and Siekmann, 1998] Cheikhrouhou, L. and Siekmann, J. H. (1998). Planning diagonalisation proofs. In Giunchiglia, F., editor, *Artificial Intelligence: Methodology, Systems, and Applications, 8th International Conference, AIMSA '98, Sozopol, Bulgaria, September 1998, Proceedings*, volume 1480 of *Lecture Notes in Computer Science*, pages 167–180. Springer.

[Comon, 2001] Comon, H. (2001). Inductionless induction. In Robinson, A. and Voronkov, A., editors, *Handbook of Automated Reasoning*, volume 1, pages 913–962. Elsevier Science.

[Comon and Nieuwenhuis, 2000] Comon, H. and Nieuwenhuis, R. (2000). Induction = I-axiomatization + first-order consistency. *Information and Computation*, 159(1–2):151–186.

[Dennis and Brotherston, 2002] Dennis, L. and Brotherston, J. (2002). *λClam v4.0.1: User/Developer's Manual*. Mathematical Reasoning Group, Division of Informatics, University of Edinburgh.

[Dennis et al., 2000] Dennis, L., Bundy, A., and Green, I. (2000). Making a productive use of failure to generate witness for coinduction from divergent proof attempts. *Annals of Mathematics and Artificial Intelligence*, 29:99–138. Also available as paper No. RR0004 in the Informatics Report Series.

[Dershowitz, 1987] Dershowitz, N. (1987). Termination of rewriting. *Journal of Symbolic Computation*, 3:69–115.

[Dershowitz and Hoot, 1993] Dershowitz, N. and Hoot, C. (1993). Topics in termination. In Kirchner, C., editor, *Rewriting Techniques and Applications, 5th International Conference, RTA-93, Montreal, Canada, June 1993, Proceedings*, volume 690 of *Lecture Notes in Computer Science,* pages 198–212. Springer.

[Free, 1992] Free, N. (1992). Summing series using proof plans. Master's thesis, Department of Artificial Intelligence, University of Edinburgh.

[Giesl, 1995a] Giesl, J. (1995a). Automated termination proofs with measure functions. In Ipke Wachsmuth, Claus-Rainer Rollinger, W. B., editor, *KI-95: Advances in Artificial Intelligence, 19th Annual German Conference on Artificial Intelligence, Bielefeld, Germany, September 1995, Proceedings*, volume 981 of *Lecture Notes in Artificial Intelligence,* pages 149–160. Springer.

[Giesl, 1995b] Giesl, J. (1995b). Generating polynomial orderings for termination proofs. In [Hsiang, 1995], pages 426–431.

[Giesl, 1995c] Giesl, J. (1995c). Termination analysis for functional programs using term orderings. In Mycroft, A., editor, *Static Analysis, Second International Symposium, SAS'95, Glasgow, UK, September 1995, Proceedings*, volume 983 of *LNCS,* pages 154–171.

[Gordon and Melham, 1993] Gordon, M. J. C. and Melham, T. F., editors (1993). *Introduction to HOL: A theorem proving environment for higher order logic*. Cambridge University Press.

[Gow and Bundy, 2000] Gow, J. and Bundy, A. (2000). The λ*Clam* system: Response to challenge problems. In Schurmann, C., editor, *Proceedings of the 9th Workshop on Automation of Proof by Mathematical Induction, CADE-17.*

[Gow et al., 1999] Gow, J., Bundy, A., and Green, I. (1999). Extensions to the estimation calculus. In Ganzinger, H., McAllester, D. A., and Voronkov, A., editors, *Logic for Programming and Automated Reasoning, 6th International Conference, LPAR'99, Tbilisi, Georgia, September 1999, Proceedings,* volume 1705 of *Lecture Notes in Artificial Intelligence*, pages 258–272. Springer.

[Gramlich, 1990] Gramlich, B. (1990). UNICOM: a refined completion based inductive theoremprover. In [Stickel, 1990], pages 655–656.

[Harrison, 1996] Harrison, J. (1996). Formalized mathematics. Technical Report 36, Turku Centre for Computer Science (TUCS), Finland.

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

[Hsiang, 1995] Hsiang, J., editor (1995). *Rewriting Techniques and Applications, 6th International Conference, RTA-95, Kaiserslautern, Germany, April 1995, Proceedings*, volume 914 of *Lecture Notes in Computer Science*. Springer.

[Huang et al., 1995] Huang, X., Kerber, M., and Cheikhrouhou, L. (1995). Adapting the diagonalization method by reformulations. In Levy, A. and Nayak, P., editors, *Proceedings of the 2nd Symposium on Abstraction, Reformulation, and Approximation*. Ville d'Estérel, Canada.

[Huet, 1975] Huet, G. (1975). A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57.

[Huet, 1991] Huet, G. (1991). The Gilbreath card trick: A case study in axiomatization and proof development in the COQ proof assistant. Technical Report 1511, INRIA.

[Huet et al., 1997] Huet, G., Kahn, G., and Paulin-Mohring, C. (1997). The Coq proof assistant — A tutorial, version 6.1. Technical Report 204, INRIA.

[Hummel, 1990] Hummel, B. (1990). *Generation of induction axioms and generalisation*. PhD thesis, Universität Karlsruhe.

[Hutter, 1990] Hutter, D. (1990). Guiding inductive proofs. In [Stickel, 1990], pages 147–161.

[Hutter, 1994] Hutter, D. (1994). Synthesis of induction orderings for existence proofs. In [Bundy, 1994], pages 29–41.

[Hutter, 1997] Hutter, D. (1997). Colouring terms to control equational reasoning. *Journal of Automated Reasoning*, 18:399–442.

[Hutter, 2000] Hutter, D. (2000). Annotated reasoning. *Annals of Mathematics and Artificial Intelligence*, 29:183–222.

[Hutter and Kohlhase, 1997] Hutter, D. and Kohlhase, M. (1997). Managing structural information by higher-order colored unification. *Journal of Automated Reasoning*, 18(3):399–442.

[Hutter and Sengler, 1996] Hutter, D. and Sengler, C. (1996). INKA: the next generation. In [McRobbie and Slaney, 1996], pages 288–292.

[Ireland, 1992] Ireland, A. (1992). The use of planning critics in mechanizing inductive proof. In [Voronkov, 1992], pages 178–189.

[Ireland and Bundy, 1996] Ireland, A. and Bundy, A. (1996). Productive use of failure in inductive proof. *Journal of Automated Reasoning*, 16:79–111.

[Ireland and Bundy, 1999] Ireland, A. and Bundy, A. (1999). Automatic verication of functions with accumulating parameters. *Journal of Functional Programming*, 9:225–245.

[Ireland et al., 1999] Ireland, A., Jackson, M., and Reid, G. (1999). Interactive Proof Critics. *Formal Aspects of Computing: The International Journal of Formal Methods*, 11(3):302–325. Longer version available from Dept. of Computing and Electrical Engineering, Heriot-Watt University, Research Memo RM/98/15.

[Jackson, 1999] Jackson, M. (1999). *Interacting with Semi-automated Theorem Provers via Interactive Proof Critics*. PhD thesis, School of Computing, Napier University.

[Jamnik et al., 2002] Jamnik, M., Kerber, M., and Pollet, M. (2002). Automatic learning in proof planning. In van Harmelen, F., editor, *Proceedings of the 15th Eureopean Conference on Artificial Intelligence, ECAI 2002, Lyon, France*, pages 282–286. IOS Press.

[Joshi, 1985] Joshi, A. K., editor (1985). *Proceedings of the 9th International Joint Conference on Artificial Intelligence, Los Angeles, CA, USA*. Morgan Kaufmann.

[Jouannaud and Kounalis, 1989] Jouannaud, J.-P. and Kounalis, E. (1989). Automatic proofs by induction in theories without constructors. *Information and Computation*, 82(1):1–33.

[Kapur and Sakhanenko, 2003] Kapur, D. and Sakhanenko, N. A. (2003). Automatic generation of generalization lemmas for proving properties of tail-recursive definitions. In Basin, D. A. and Wolff, B., editors, *Theorem Proving in Higher Order Logics, 16th International Conference, TPHOLs 2003, Rome, Italy, September 2003, Proceedings*, volume 2758 of *Lecture Notes in Computer Science*, pages 136–154. Springer.

[Kapur and Subramaniam, 1996] Kapur, D. and Subramaniam, M. (1996). Lemma discovery in automating induction. In [McRobbie and Slaney, 1996], pages 538–552.

[Kapur and Zhang, 1995] Kapur, D. and Zhang, H. (1995). An overview of rewrite rule laboratory (RRL). *Journal of Computer Mathematics with Applications*, 29(2):91–114.

[Kaufmann and Moore, 1996] Kaufmann, M. and Moore, J. S. (1996). ACL2: An industrial strength version of Nqthm. In *Proceedings of the Eleventh Annual Conference on Computer Assurance, 1996. COMPASS '96*, pages 23–34, Gaithersburg, Maryland. National Institute of Standards and Technology.

[Kerber, 1998] Kerber, M. (1998). Proof planning – A practical approach to mechanised reasoning in mathematics. In Bibel, P. H. S. W., editor, *Automated Deduction – A Basis for Applications*, chapter Vol. III, pages 77–95. Kluwer.

[Knuth and Bendix, 1970] Knuth, D. and Bendix, P. (1970). Simple word problems in universal algebra. In Leech, J., editor, *Computational Problems in Abstract Algebras*, pages 263–297. Pergamon Press.

[Kraan, 1994] Kraan, I. (1994). *Proof Planning for Logic Program Synthesis*. PhD thesis, Univeristy of Edinburgh.

[Kraan et al., 1996] Kraan, I., Basin, D., and Bundy, A. (1996). Middle-out reasoning for synthesis and induction. *Journal of Automated Reasoning*, 16(1–2):113–145.

[Kreisel, 1965] Kreisel, G. (1965). *Lectures on Modern Mathematics*, volume 3, chapter Mathematical Logic, pages 95–195. John Wiley and Sons.

[Lusk and Overbeek, 1988] Lusk, E. L. and Overbeek, R. A., editors (1988). *9th International Conference on Automated Deduction, Argonne, Illinois, USA, May 1988, Proceedings*, volume 310 of *Lecture Notes in Computer Science*. Springer.

[Maclean, 1999] Maclean, E. (1999). Generalisation as a critic to the induction strategy. Master's thesis, Department of Artificial Intelligence, University of Edinburgh.

[McAllester and Arkoudas, 1996] McAllester, D. and Arkoudas, K. (1996). Walther recursion. In [McRobbie and Slaney, 1996], pages 643–657.

[McCarthy, 1963] McCarthy, J. (1963). A basis for a mathematical theory of computation. In Braffort, P. and Hirschberg, D., editors, *Computer Programming and Formal Systems*, pages 33–70. North-Holland, Amsterdam. Available online from `http://www-formal.stanford.edu/jmc/basis.html`.

[McCune, 1990] McCune, W. (1990). The Otter user's guide. Technical Report ANL/90/9, Argonne National Laboratory.

[McRobbie and Slaney, 1996] McRobbie, M. A. and Slaney, J. K., editors (1996). *Automated Deduction — CADE-13: 13th International Conference on Automated Deduction, New Brunswick, NJ, USA, August 1996, Proceedings*, volume 1104 of *Lecture Notes in Artificial Intelligence*. Springer.

[Melis and Meier, 2000] Melis, E. and Meier, A. (2000). Proof planning with multiple strategies. In Lloyd, J. W., Dahl, V., Furbach, U., Kerber, M., Lau, K.-K., Palamidessi, C., Pereira, L. M., Sagiv, Y., and Stuckey, P. J., editors, *Proceedings of the First International Conference on Computational Logic (CL 2000)*, volume 1861 of *Lecture Notes in Computer Science*, pages 644–659. Springer.

[Melis et al., 2000] Melis, E., Zimmer, J., and Müller, T. (2000). Extensions of constraint solving for proof planning. In Horn, W., editor, *ECAI 2000, Proceedings of the 14th European Conference on Artificial Intelligence, Berlin, Germany*, pages 229–233. IOS Press.

[Monroy, 2000] Monroy, R. (2000). The use of abduction and recursion-editor techniques for the correction of faulty conjectures. In Alexander, P. and Flener, P., editors, *Proceedings of ASE-2000: The 15th IEEE Conference on Automated Software Engineering, Grenoble, France*, pages 91–100. IEEE CS Press.

[Nadathur and Miller, 1998] Nadathur, G. and Miller, D. (1998). Higher-order logic programming. In Gabbay, D. M., Hogger, C. J., and Robinson, J. A., editors, *Handbook of Logics for Artificial Intelligence and Logic Programming*, volume 5, pages 499–590. Clarendon Press, Oxford.

[Nederpelt et al., 1994] Nederpelt, R. P., Geuvers, J. H., and de Vrijer, R. C., editors (1994). *Selected Papers on Automath*, volume 133 of *Studies in Logic and the Foundations of Mathematics*. North-Holland.

[Newell et al., 1956] Newell, A., Shaw, C., and Simon, H. (1956). The logic theory machine. *IRE Transactions Information Theory*, 2(3):61–79.

[Owre et al., 1996] Owre, S., Rajan, S., Rushby, J. M., Shankar, N., and Srivas, M. K. (1996). PVS: Combining specification, proof checking, and model checking. In Alur, R. and Henzinger, T. A., editors, *Computer Aided Verification, 8th International Conference, CAV '96, New Brunswick, NJ, USA, August 1996, Proceedings*, volume 1102 of *Lecture Notes in Computer Science*, pages 411–414. Springer.

[Pascal, 1665] Pascal, B. (1665). *Traité du triangle arithmétique avec quelques autres petits traités sur la même matière*. In Brunschvicg, L., Boutroux, P. and Gazier, F, editors, *Oeuvres de Blaise Pascal*, 1978.

[Paulson, 1989] Paulson, L. C. (1989). The foundation of a generic theorem prover. *Journal of Automated Reasoning*, 5:363–397.

[Paulson, 1991] Paulson, L. C. (1991). *ML for the Working Programmer*. Cambridge University Press, Cambridge, UK.

[Pientka and Kreitz, 1999] Pientka, B. and Kreitz, C. (1999). Automating inductive specification proofs. *Fundamenta Informaticae*, 39(1-2):189–208.

[Protzen, 1994] Protzen, M. (1994). Lazy generation of induction hypotheses. In [Bundy, 1994], pages 42–56.

[Protzen, 1995] Protzen, M. (1995). *Lazy Generation of Induction Hypotheses and Patching Faulty Conjectures*. PhD thesis, Technical University of Darmstadt, Germany.

[Rashed, 1994] Rashed, R. (1994). *The development of Arabic mathematics : between arithmetic and algebra*. London.

[Richardson et al., 2000] Richardson, J., Dennis, L., Gow, J., and Jackson, M. (2000). *User/programmer manual for the λClam proof planner (version 2.0.0)*. Mathematical Reasoning Group, Division of Informatics, University of Edinburgh.

[Richardson and Smaill, 2001] Richardson, J. and Smaill, A. (2001). Continuations of proof strategies. In Goré, R., Leitsch, A., and Nipkow, T., editors, *IJCAR 2001 — Short Papers, International Joint Conference on Automated Reasoning*, pages 130–139.

[Richardson et al., 1998] Richardson, J., Smaill, A., and Green, I. (1998). System description: Proof planning in higher-order logic with λ*Clam*. In Kirchner, C. and Kirchner, H., editors, *Automated Deduction — CADE-15: 15th International Conference on Automated Deduction, Lindau, Germany, July 1998, Proceedings*, volume 1421 of *Lecture Notes in Artificial Intelligence*, pages 129–133. Springer.

[Schreye and Decorte, 1994] Schreye, D. D. and Decorte, S. (1994). Termination of logic programs: The never ending story. *Journal of Logic Programming*, 19/20:199–260.

[Sengler, 1996] Sengler, C. (1996). Termination of algorithms over non-freely generated datatypes. In [McRobbie and Slaney, 1996], pages 121–135.

[Shankar, 1994] Shankar, N. (1994). *Metamathematics, Machines, and Godel's Proof*. Cambridge University Press.

[Smaill and Green, 1996] Smaill, A. and Green, I. (1996). Higher-order annotated terms for proof search. In von Wright, J., Grundy, J., and Harrison, J., editors, *Theorem Proving in Higher Order Logics, 9th International Conference, TPHOLs'96, Turku, Finland, August 1996, Proceedings*, volume 1125 of *Lecture Notes in Computer Science*, pages 399–413. Springer.

[Steinbach, 1995] Steinbach, J. (1995). Automatic termination proofs with transformation orderings. In [Hsiang, 1995], pages 11–25.

[Stevens, 1988] Stevens, A. (1988). A rational reconstruction of Boyer & Moore's technique for constructing induction formulas. In Kodratoff, Y., editor, *8th European Conference on Artificial Intelligence, ECAI 88, Munich, Germany, August 1988, Proceedings*, pages 565–570. Pitmann.

[Stevens, 1990] Stevens, A. (1990). *An Improved Method for the Mechanisation of Inductive Proof*. PhD thesis, Department of Artificial Intelligence, University of Edinburgh.

[Stickel, 1990] Stickel, M. E., editor (1990). *10th International Conference on Automated Deduction, Kaiserslautern, Germany, July 1990, Proceedings*, volume 449 of *Lecture Notes in Artificial Intelligence*. Springer.

[Trybulec and Blair, 1985] Trybulec, A. and Blair, H. (1985). Computer assisted reasoning with Mizar. In [Joshi, 1985], pages 26–28.

[van der Waerden, 1961] van der Waerden, B. L. (1961). *Science Awakening*. New York.

[van Harmelen, 1996] van Harmelen, F. (1996). *The Clam proof planner: user manual and programmer manual (version 2.5)*. Mathematical Reasoning Group, Division of Informatics, University of Edinburgh.

[Voronkov, 1992] Voronkov, A., editor (1992). *Logic Programming and Automated Reasoning, International Conference, LPAR'92, St. Petersburg, Russia, July 1992, Proceedings*, volume 624 of *Lecture Notes in Artificial Intelligence*. Springer.

[Walsh, 1996] Walsh, T. (1996). A divergence critic for inductive proof. *Journal of Artificial Intelligence Research*, 4:209–235.

[Walther, 1988] Walther, C. (1988). Argument-bounded algorithms as a basis for automated termination proofs. In [Lusk and Overbeek, 1988], pages 602–621.

[Walther, 1992] Walther, C. (1992). Computing induction axioms. In [Voronkov, 1992], pages 381–392.

[Walther, 1993] Walther, C. (1993). Combining induction axioms by machine. In [Bajcsy, 1993], pages 95–101.

[Walther, 1994a] Walther, C. (1994a). Mathematical induction. In Gabbay, D. M., Hogger, C. J., and Robinson, J. A., editors, *Handbook of Logic in A.I. and Logic Programming*, volume 2, pages 127–228. Oxford University Press.

[Walther, 1994b] Walther, C. (1994b). On proving termination of algorithms by machine. *Artificial Intelligence*, 71(1):101–157.

[Yoshida et al., 1994] Yoshida, T., Bundy, A., Green, I., Walsh, T., and Basin, D. A. (1994). Coloured rippling: an extension of a theorem proving heuristic. In Cohn, A. G., editor, *Proceedings of the Eleventh European Conference on Artificial Intelligence, Amsterdam, The Netherlands*, pages 85–89. John Wiley and Sons.